August 2013

Master's Degree Thesis

A High Speed LT Codec Processor Design Using ASIP Implementation Tools

Graduate School of Chosun University

Department of Information and Communication Engineering

S. M. Shamsul Alam

A High Speed LT Codec Processor Design Using ASIP Implementation Tools

ASIP 구현툴들을이용한고속 LT Codec 프로세서설계

August 23, 2013

Graduate School of Chosun University

Department of Information and Communication Engineering

S. M. Shamsul Alam

A High Speed LT Codec Processor Design Using ASIP Implementation Tools

Thesis Advisor: GoangSeog Choi, PhD

This thesis is submitted to the Graduate School of Chosun University in partial fulfillment to the requirements for a Master's degree in Engineering

April 2013

Graduate School of Chosun University

Department of Information and Communication Engineering

S. M. Shamsul Alam

Graduate School of Chosun University

Gwangju, South Korea

CERTIFICATE OF APPROVAL

MASTER'S THESIS

This is to certify that the master's thesis of

S. M. ShamsulAlam

has been approved by examining committee for the thesis requirement for the Master's degree in Engineering

Committee Chairperson ……..
Prof. Jae-Young Pyun

Committee Member ……..
Prof. Young-Sik Kim

Committee Member ……..
Prof. GoangSeog Choi

알람 샴술 석사학위 논문을 인준함

위원장 조선대학교 교수 변 재 영 印

위 원 조선대학교 교수 김 영 식 印

위 원 조선대학교 교수 최 광 석 印

2013년 5월

조 선 대 학 교 대 학 원

Contents

List of Figures

Figure Title Page

List of Tables

List of Abbreviations and Symbols

ADF	Architecture Definition File.	t_{cycle}	cycle time
ADL	Architecture Description Language	*#Instr*	Number of instruction
ALU	Arithmetic Logical Unit	AC_{compl}	architectural complexity
ARQ	Automatic Repeat Request	DP_{compl}	data path complexity
ASIC	Application Specific Integrated Circuit	ε	Erased probability
ASIP	Application Specific Instruction-set Processor	*	Erasure
ASM	Assembly Instruction Set	μ	binary message domain
ASP	Application Specific Processor	r	likelihood ratio
BEC	Binary Erasure Channel	l	log-likelihood ratio
BEM	Binary Encoding Map	δ	failure probability
BIMS	Binary-Input, Memoryless and Symmetric	E	Excess Packet
		$\mathbf{G}$	Generator Matrix
BP	Bypass	$\rho(d)$	degree distribution (ISD)
BP	Belief Propagation	$\tau(d)$	degree distribution (RSD)
BTB	Branch Target Buffer	R	Information rate
CFG	Control Flow Graphs	$\oplus$	XOR Operation
CG	Connectivity Graph	*.cfg*	Configuration file
CISC	Complex Instruction Set Computer	*.log*	Log file
CMOS	Complementary Metal Oxide Semiconductor	*.vmem*	Memory image file
		$\mathbf{H}$	parity check matrix
CNU	Check Node Unit	$Cap(C)$	Channel Capacity
CPI	Cycles Per Instruction	M_{par}	amount of parallelism

DC	Decode
DDG	Degree Distribution Generator
DLPP	Data Level Parallel Processors
DSE	Design Space Exploration
DSP	Digital Signal Processor
DSP	Digital Signal Processing
EICT	Exploiting ILP at Compile Time
ELF	Executable and Linkable Format
EX	Execution Stage
FLIX	Flexible Instruction Extensions
FPGA	Field Programmable Gate Array
FU	Function Unit
GPP	General Purpose Processor
HDB	Hardware Database
HDL	Hardware Description Language
HW	Hardware
IDE	Integrated Development Environment
IF	Instruction Fetch
ILP	Instruction-Level Parallelism
ILPP	Instruction level parallel processor
INU	Input node Processing Unit
IP	Intellectual Property
ISD	Ideal Soliton Distribution
ISS	Instruction Set simulator
IU	Immediate Unit
LDPC	Low Density Parity Check
LLR	Log Likelihood Ratio

#Trans	number of transistors per chip
O	number of operations
I	issue rate
D	number of operand
S	superpipelining degree
$\mathbf{e}$	Error vector
$\mathbf{x}$	Input message vector
$\mathbf{y}$	Transmitted code vector
$\mathbf{z}$	Received code vector

LLVM	Low Level Virtual Machine
LSU	Load/Store Unit
LT Code	Luby Transform Code
LUT	Look Up Table
LDW	Load Word
MAP	Maximum A Posteriori
MBIST	Memory Built In Self Test
MD	Multiple Data
MIMD	Multiple Instruction Multiple Data
MISC	Multiple Instruction Stream Computers
MO	Multiple Operation
ONU	Output Node Processing Unit
OR 1200	OpenRISC 1200
PIC	Programmable Interrupt Controller
PIG	Program Image Generator
PM	Power Management
RC	Read Connection Port
RF	Register File
RISC	Reduced Instruction Set Computing
RNG	Random Number Generator
ROM	Read Only Memory
RR	Read Register
RSD	Robust Soliton Distribution
RTL	Register Transfer Level
SIMD	single instruction multiple data
SIP	Silicon Intellectual Property
SISC	Single Instruction Stream Computers

SoC	System on chip
STW	Store Word
TCE	TTA based Co-design Environment
TCECC	TCE C Compiler
TIE	Tensilica Instruction Extension
TPEF	TTA Program Exchange Format
TSMC	Taiwan Semiconductor Manufacturing Company
TTA	Transport Triggered Architecture
VCD	value change dump
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
VNU	Variable Node Unit
WB	Write Back
WP	Write connection Port
XX	Xtensa Xplorer

논문초록

ASIP 구현툴들을이용한고속 LT Codec 프로세서설계

알람샴술
논문지도교수: 최광석
공동지도교수: 권구락
정보통신공학과
조선대학교대학원

오늘날 루비 변환코드는 분수 부호 영역에서중요한 역할로 사용되고 있다. 이 논문은 응용 특정 명령어 세트프로세서 (ASIP) 디자인 툴을 사용하여 LT 코덱의 구현을 위한 다양한 기술을 알려준다. ASIP 디자인에서 프로세서의 성능을 향상할 수 있는 일반적인 방법은 동시 작동을 보장하기 위한 기능들을 향상하는 것이다. 이러한 이유 때문에, 지난 몇 년간의 연구에 응용 프로그램 특정 도메인에서 프로세서의 작동을 직접하였다. 따라서이 연구 논문에서 LT 코덱과 같은 이러한 응용 프로그램 특정 작업은 서로 다른 프로세서 플랫폼을 사용하여 구현되었다.하드웨어의 성능과 프로세서의 구조에 따라 달라질 뿐만 아니라 입력 응용 프로그램 LT Codec의 구조에 따라 달라진다. 따라서입력 설계 전략, 프로세서 및 컴파일러 아키텍처와같은 최적화는 응용하는 특정 프로세서의성능을 향상시킬 수 있는 매우 유용한 현상이다. 지난 몇 년 동안, 프로세서 아키텍처는RISC 가족의 영역에서 발전되어 왔다. 교육 수준 병렬 처리 (ILP), 우회 기법 및 여러 강좌와 같은 몇 가지 주요 개념은 RISC 프로세서의 운영에 포함되어 있다. 따라서 운송 실

행 아키텍처(TTA)는 응용 프로그램 특정 프로세서 디자인에 스타일을 기본으로 한다.

이 논문은 LT 코덱을위한 고속 TTA 프로세서를 설계 할 몇 가지 기술을 분석한다. 이외에도 TTA 아키텍처, LT 인코더와 디코더의 설계 수정이 쉽고 효율적인 코덱 생산을 하기 위해 수정되어야 한다. 따라서이 논문은 복구 목적으로 소프트 디코딩으로 알려진 여러개의 제품 알고리즘을 사용하고, 그리고매우 적은 반복작업으로 AWGN 채널을 통해 인코딩 된 비트 스트림에서 디코딩 된 신호를 생성하였다.

TTA 기반의 병행설계 환경 (TCE) 툴(tool)은 LT코덱을 실행하는 프로세서의 다양한 범주(category)를 개발하기 위해서 사용돼왔다. 게다가, 이 결과를 다른것들의 응답과 비교하기 위해 LT 코덱을 실행하기 위한 Tensilica 와 OpenRISC 툴들을 사용했다. TCE와 마찬가지로, Tensilica 툴에서 프로세서의 성능을 극대화 하기위해 몇몇 환경설정들이 선택(설정)되고 수정되었다. 이러한 활동들을 기반으로, 몇몇 유용한 결과들이 생성되었고 TTA의 LT_CODEC.adf 아키텍쳐가 TCE와 Tensilica 툴의 아키텍쳐와 비교했을때 LT 엔코더와 디코더를 실행함에 있어서 최소의 사이클을 차지함을 보여주었다.이런 프로세서에서 Decoder.adf 나 Decoder_llr.adf 그리고 마지막으로 LT_CODEC.adf라 이름지어진 일반적인 기능 유닛(function unit)들은 TTA 프로세서의 기능을 점진적으로 개선하기 위해서 사용되었다. LT_CODEC.adf는 LT코덱을 시뮬레이팅 하는데 오직 4466 사이클과 43ms를 차지 했는데, 이는 Tensilica 툴에 비하면 매우 적다.
그럼에도 불구하고 Tensilica의 시뮬레이션 스피드는 TCE 툴에 비하면 매우 빠르다. 이런 시뮬레이션 결과로부터 100MHz 클럭을 사용하여 초당 거

의 100K 사이클을 수행한다고 볼 수 있다. 그러나 Tensilica는 ConnX D2 엔진을 이용하여 초당 1M 사이클을 수행한다. LT 코덱의 디코딩 기술은 반복 방식으로서 수행되었고다른 프로세서 아키텍쳐의 변화 때문에 이런 디코딩 반복 방식은 TCE와 Tensilica 툴을 이용하여 연구되었다. 이런 결과로부터 TCE 툴의 LT_CODEC.adf는 디코더 된 신호를 발생시키기 위해 오직 단일 반복만을 취했다. 그러나 Tensilica의 XRC_D2MR 환경은 성공적인 디코딩을 위해서 9 사이클을 취했다. 나중에 이 학위논문은 TCE, Tensilica 와 OpenRICS 사이의 비교를 나타낸다. 결과는 Tensilica 툴은 OpenRISC 보다 더 많은 사이클을 취하고 TCE의 성능은 다른것들과 비교했을때 더 좋다는 것을 보여준다. 그러나 이 비교에서 OpenRISC 프로세서의 제한 때문에 오직 LT 엔코더만 시뮬레이션 되었다. Tensilica와 마찬가지로 OpenRISC는 명령을 실행하기 위해서 몇몇 사이클을 사용하는데이는 OTA 클레스 프로세서 툴의 일반적인 행동이다.

ABSTRACT

A High Speed LT Codec Processor Design Using ASIP Implementation Tools

S. M. Shamsul Alam
Advisor: Prof. GoangSeog Choi, Ph.D.
Department of Information and Communication Engineering
Graduate School of Chosun University

Luby Transform code nowadays plays an important role in the area of fountain code. This thesis reports the various techniques for implementation of LT codec using the application specific instruction set processor (ASIP) design tools. In ASIP design, a common approach to increase the performance of processors is to boost the number of function units for ensuring concurrent operation. Due to this reason, in past few decades researches had been carried out to dedicate the operation of processor on application specific domain. Therefore, in this research paper, such an application specific work like LT codec was implemented using different processor platforms. The performance of the hardware not only depends on architecture of the processor but also depends on structure of the input application i.e. LT codec for this thesis. Therefore, optimizations like strategy of input design, processor and compiler architecture are very useful phenomenon to enhance the performance of the application specific processor. In past few years, processor architectures had been evolved in the area of RISC family. Some key concepts like instruction level parallelism (ILP), bypassing technique, and multiple instruction executions are included with the operation of the RISC processor. Hence transport triggered architecture (TTA) is promising style in application

specific processor design. This thesis analyzes some techniques to design a high speed TTA processor for LT codec. Besides this modification of TTA architecture, the design of the LT encoder and decoder should be modified to make a simple and computationally efficient codec processor. Therefore, in this thesis, sum product algorithm known as soft decoding had been used for message recovery purpose and this algorithm took very less iterations for generating error free decoded signal from encoded bit streams through AWGN channel.

TTA based co-design environment (TCE) tool has been used for developing various category of processors in this LT codec implementation. Moreover, to compare this result with other's response, Tensilica and OpenRISC tools are taken for implementing this LT codec. Like TCE, in Tensilica tool several configurations are chosen and modified for optimizing the performance of the processor. Based on these activities some useful results are produced and it shows that LT_CODEC.adf architecture under TTA takes minimum cycles compared to other architectures of TCE and Tensilica tools for implementing LT encoder and decoder. In this processor, some processor architectures named as Decoder.adf and Decoder_llr.adf and finally LT_CODEC.adf are generated for gradually improving the performance of the TTA processor. LT_CODEC.adf took only 4466 cycles and 43 ms for simulating LTcodec, which is very less compared to the Tensilica tool. Nevertheless, the simulation speed of Tensilica is very high compared to the TCE tool. From this simulation result, it can be shown that TCE executes almost 100 K cycles per second using 100 MHz clock. However, Tensilica runs 1 M cycles per second using ConnX D2 engine. It is shown that the decoding technique of LT codec has been performed as iterative manner and the manner of this decoding iteration due to the change of different processor architectures was investigated using TCE and Tensilica tools. From this experiment, LT_CODEC.adf of TCE tool took only single iteration for generating decoded signal. However, XRC_D2MR configuration of Tensilica took 9 cycles for successful decoding. Later, this thesis

portrays a comparison between TCE, Tensilia and OpenRISC tool. Result shows that Tensilica tool takes more cycles than OpenRISC and the performance of the TCE is very good compared to others. But, in this comparison, only LT encoder was simulated due to the limitations of OpenRISC processor. Like Tensilica, OpenRisc takes separate cycles for executing the instructions, which is a common behavior of the operation triggered architecture (OTA) class processor tools. On the other hand, for TCE tool it is occurred as the side effect of data transport. Moreover, to determine the efficiency of the LT Codec architecture, the encoder and decoder are implemented with a core area of 9 mm^2 in TSMC 180-nm 1-poly 6-metal and Samsung 130-nm complementary metal–oxide–semiconductor (CMOS) technology. Therefore, an efficient trade off is required between all these observations to design an excellent processor based on the specific input application.

Chapter 1

Introduction

System on chip (SoC) is a great revolution in modern era. Like integrated circuit, SoC includes many components of digital, analog or mixed signal electronic system in a single chip. Therefore, SoC plays a vital role in the area of embedded system. As a result, new design tools and methodologies are required to address the design, test and verification for SoC. In today's SoC design, programmability, reusability and concurrent operation ability are the most exigent challenges and these force the design work from Register Transfer Level (RTL) to a higher abstraction level. Silicon Intellectual Property (SIP) or Silicon IPs are used as components in silicon chip design since mid-1990s. The important constrains for quality design of SIP became higher after the year 2000. After that time, SIP has been accepted widely and used in large scale [1]. Figure 1 shows the design complexity using SIP. As shown in Fig. 1, around the middle to late of 1980s, RTL components were optimized as the lowest level component of system design. In this stage, RTL components took a certain degree of design complexity from the system design so that the system could be relatively more advanced compare to the system designed on a transistor level. During 1990s, the system design became more advanced and complicated that programmable IP has to be used as the lowest level component to relax the system design complexity [1]. After 2005, the component design complexity was dramatically increasing which was handled by SoC platform.

1.1 Design Goal or Motivation

The traditional RTL design and SoC design differ from the size of their basic building blocks. Designers can use complete blocks instead of logic gates and registers. In order to increase the productivity, hardware design reuse is vital factor in SoC system. To build complex systems, designers can integrate the pre-designed and

pre-verified intellectual property (IP) blocks to save the time to market of a product. Designers are working hard to meet the requirements of embedded system design constrains like enhanced performance, less area, low power and less time to market. General Purpose Processors (GPPs), Digital Signal processors (DSPs) and Application Specific Integrated Circuits (ASICs) are trying to solve the SoC design problem partially. Because of wide variety of applications, GPPs are not suitable for application specific embedded system. Here, designers think of ultimate performance and flexibility. Since the application and programmer's behavior are unknown, the instruction set must be general. As a result, for different embedded system devices, GPPs cannot provide good performance at low power. Similarly, in ASIC there is no post-programmable opportunity, so its reusability is very limited. On the other hand, in spite of programmability DSPs cannot achieve high performance with low power dissipation. Because of that, in order to get an optimistic solution for SoC design, there is a recent interest in new flexible architectures with programmability and instruction parallelism and probably now a days it is known as Application Specific Instruction-set Processor (ASIP). These ASIP architectures can replace multiple chip designs implemented as ASIC architecture [2]. Sometimes ASIP is known as SIP. More SoC solutions use ASIP IP. For ASIP designers, the biggest challenge is the efficiency issue. Based on the coverage of full functionality of input application, the main target of ASIP design is to gain the highest performance over silicon and the highest performance over power consumption as well as the highest performance over design cost [1]. For this reason ASIP gives more impression to solve all the constrains in SoC scheme and looks very good solution for application specific embedded systems design. Recently, ASIPs provide enhanced performance and flexibility and keep the benefit of post-programmability compared to custom ASICs. The extensible use of programmable processor platforms brings the advantage of Time-to-market. ASIPs are optimized to execute a single application or a set of applications for focusing on the specific purpose. In ASIP, it is possible to get higher performance if

the processor resources like registers, function units and computational units are exactly matched with the input application. For example, in input application there is no left or right shift operation, so this shift operation can be removed from the processor architecture. As a result, this specific processor will take less power and area compared to general-purpose processors those include all operation instructions. For this reason, the main instigation of ASIPs is to increase the performance of application without implementing fixed function hardware components. On the other hand, manual IP block design sometimes time consuming and expensive. It requires a tradeoff between GPP software implementation and pure hardware implementation in terms of area, power and time. ASIP implementation is perfect for this trade off and it is capable for scalable operation in terms of performance per area and power consumption factors [3]. In ASIP, a platform (next it is known as custom function unit in specific processor architecture) is a partly designed application specific system that is used to adjust to a custom design with minimum cost. Therefore, this platform based system design requires minimum design cost during the plugging a programmable IP on the platform.

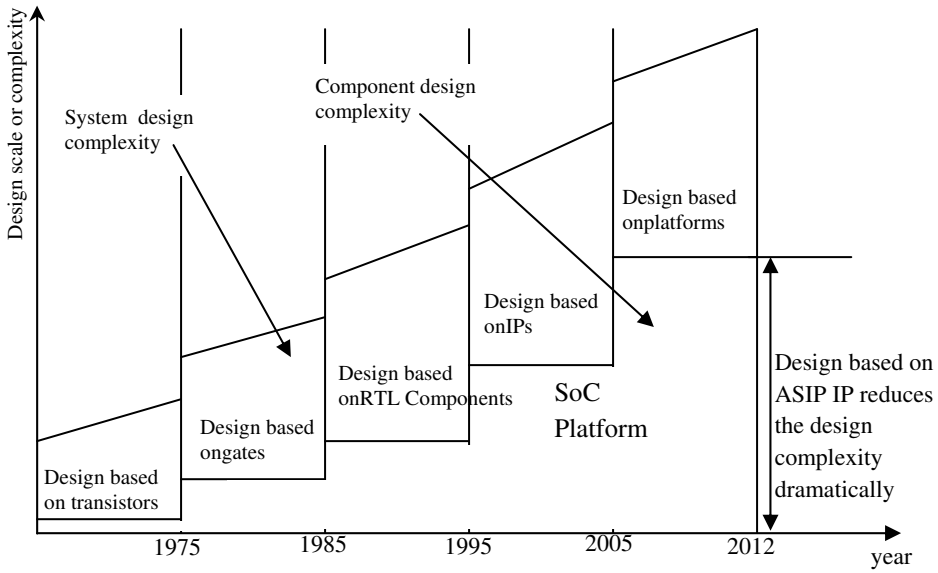


Figure 1: Dealing complexity of the design using Silicon IP and SoC platform [1].

Processor design is not an easy task. Without the help of advanced design flow diagram, it is very difficult to design processor in time and even not possible to maintain high quality. For complicated system such as ASIP, therefore the design flow is very much essential. Figure 2 shows the state of the art ASIP design flow adopted from ref. [1]. This ASIP design flow is divided into three parts: architecture design, design of programming tools, and firmware design, as depicted in Fig. 2 (a).

In this thesis, I will focus on the application specific processor design techniques by using ASIP design tools. I have selected this input application as Luby Transform codec (LT code). The reasons for selecting this LT code as a class of fountain codes have been discussed in the chapter 4. Now a days the ASIP design is very promising technique due to its tremendous demands in daily applications. In order to reduce the time to market and to improve the excellency of the processor, there are many automated design tools developed in this area. This thesis will describe the processor design techniques using different tools for specific application.

The instruction set design is most important step in this design flow and this is the first step of ASIP design process. This design stage is complicated and cannot be claimed that a certain instruction set is the best. There should be a tradeoff of instruction set among different parameters like performance, functional coverage, flexibility, power consumption, silicon cost, and design time etc. Figure 2(b) represents a basic design flow for the design of an instruction set architecture. As shown in Fig. 2(b), at the starting stage, first the input application should be specified and then translate to functional coverage. Under functional coverage, it is required to collect the relevant standard specifications and knowledge in order to add extra features for future usage.

After getting the input application specification, the partitioning of hardware/software should be decided through profiling of the source code. It is required to meet the performance constraint by defining the functions boosted by application specific instructions and the functions accelerated by software using

conventional instructions. This is an important design concept known as 10% - 90% code locality. That means 10% of the instructions run by 90% of the time and 90% of the instructions run by 10% of time. Therefore, ASIP design required to find the best instruction-set architecture optimized for the 10% frequently used instructions and to avoid the instructions among the 90% those are not frequently used. The next step is to implement the instruction set that include instruction-coding, design of the instruction set simulator, and benchmarking. Therefore, the compiler takes the instruction set and converts into the assembly syntax and the design of the binary machine codes. Then the Instruction Set simulator (ISS) implements the instruction-set in forms of assembly and binary codes.

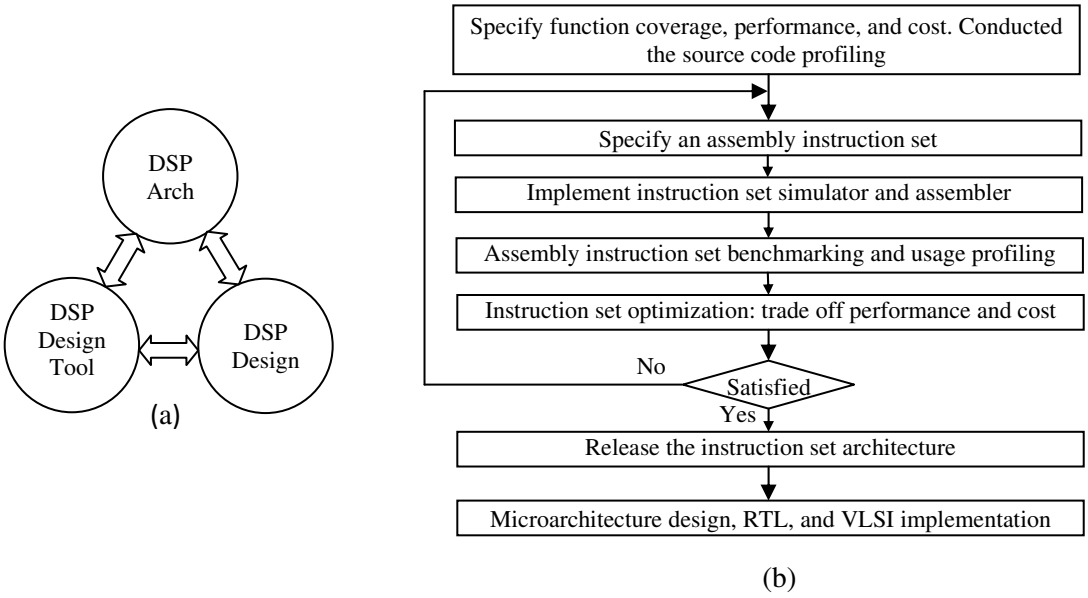


Figure 2: Hierarchy of ASIP design flow. (a) Different sections of ASIP design. (b) Basic flow of ASIP design [1].

Finally benchmarking is applied to evaluate the performance of instruction-set. Moreover, the performance of instruction-set can be modified and the usage of each instruction will be exposed for further optimization. The ASIP design flow takes the specific design requirements as input and deliveries the microarchitecture design as

output. The design of an ASIP is based mostly on experience, and it is essential to minimize the cost of design iteration. This microarchitecture in form of RTL coding is known as the tiny processor of the specific input application. This RTL design is ready to use for chip design.

1.2 Thesis Organization

Chapter 2 describes the evolution of different processor architectures. It presents the improvement of the RISC processor and explains how ILP, bypassing techniques, and FUs as well as RFs are added in the processor architectures.

Chapter 3: After discussing the basic architecture of typical processors in chapter 2, in chapter 3, an efficient architecture of processor has been discussed. It shows why transport Triggered architecture is more suitable for designing the custom function unit. Finally, it represents the hardware structure of TTA.

The main ideology of this thesis is to design an efficient processor for LT encoder and decoder. For this reason after selecting the suitable processor, we need to discuss about Luby Transform code. Chapter 4 will be discussing about the LT codec. There are many issues for implementing the encoder and decode of LT codec. Chapter 4 includes the basic algorithm of encoding and decoding procedure, degree distribution and background study of the LT codec implementation. Next chapter will show the proposed architecture of input design as well as processor design.

For this reason chapter 5 surely discusses about the ASIP design tools. Moreover, our proposed LT codec architectures are explained in this chapter. This architectures have been implemented by using ASIP design tool.

In this thesis work TCE, Tensilica and OpenRisc processor tools have studied and proposed processor of LT codec has been developed. This chapter represents the basic theories for developing processor using these tools. By using these concepts, LT codec program has been simulated which is shown in simulation chapter.

Chapter 6 shows these simulation results generated by three tools. First of all individual result generated by specific tool has been displayed. Here mainly cycle counts and simulation time are taken as reference parameters for comparing the performance of the tool. After simulating using all these thee tools, then a comparison table is portrayed to get the overall scenario of these tools.

Finally, in chapter 7, the whole work of this thesis will be summarized including the limitations as well as different diversified optimization levels of these tools. Besides this, an effective discussion are reported to make trade off between input design and optimization level of the processors. However to get the ultimate goal, some future works have been proposed with few ideas.

Chapter 2

Evolution of RISC Processors

In the previous section, we have discussed about the general concept of ASIP design goal that includes the instruction-set generation, ISS execution and microarchitecture formation technique. In this chapter, we will be discussing about the step-by-step evolution of processors. The main theme for selecting processor platform is to take a processor class that has concurrent operation strategy, good flexibility in terms of use and more automated working functionality. Therefore, design automation is very important to reduce the time to market. For complex input deign, it is difficult to design the instruction-set manually. So, in order to make the ASIP work user-friendly it requires an automated design tool for improving the efficiency and reducing the research time. For this reason for selecting ASIP design tool, that exploits good design automation.

2.1 Design Automation

Figure 3 (a) shows the overview of ASIP design automation in point of research view. This design automation is divided into three major parts: architecture exploration, modeling and generation-verification. In first step, architecture and assembly instructions are generated according to the input application analyses. Here, researchers design different profiles like control flow graph. The tool will merge different control flow graphs. Architecture Description Language (ADL) is required to model the instruction-set and architecture which is shown in second stage of Fig. 3 (a). ADL is little bit difficult to understand. It should have sufficient information regarding the modeling of instruction-set, data path, control path and microarchitecture. If the ADL carries sufficient information for generating tools and architectures, the ADL will not be readable and cannot be used by ASIP designers [1]. The third stage includes the generation and verification of processor. Some ASIP

design tools like Xtensa, LISA, OpenrRISC, TCE etc are extensively used for this generation and verification purposes. However, in designer's point of view this ASIP design flow is different compared to research point. The designer's should give focus on on how to use the tool to generate instruction set, architecture, and assembly programming tools, as well as support for design verifications. Figure 3 (b) shows the ASIP design flow in point of design view. Architecture and assembly instruction set exploration are first and most important part in ASIP design flow in point of design view. Because there is a huge gap between CFGs (control flow graphs) of multiple applications and an ASM (assembly instruction set) and many choices are possible to select different instruction set architectures. So to reduce the effect of this large gap another design step (constraint specification) might be needed. Designers will propose the instruction-set architecture of a processor and this instruction set and architecture will be the inputs of processor modeling. The processor model will be used for generating the instruction set simulator, the compiler, assembler, and the architecture behavior model. After benchmarking of the instruction set and architecture, RTL code will be finally generated by the ASIP automation design tools [1].There are many kinds of ASIP design tools developed by different research institutes and universities over the years. Those are MIMOLA, Cathedral-II, Target, ARC, Xtensa, LISA, MESCAL, PEAS-III, NOGAP, TCE, OpenRisc etc. In this thesis, I have used Xtensa, OpenRisc and TCE tools to simulate my input application. Xtensa configurable processor is used as ASIP design tool that is developed under Tensilica IP core company in Silicon Valley. Similarly, OpenRisc is developed under the project of OpenCores community. It's purpose is to develop a series of general purpose open source RISC CPU architectures. TTA based Co-design Environment (TCE) tool developed by Tampere University of Technology, Finland. TCE is a toolset for designing ASIP based on the Transport Triggered Architecture (TTA). This toolset provides a complete design flow from C program to synthesizable Hardware Description Language (HDL) and parallel program binaries. Besides the discussion on

Xtensa and OpenRisc, this thesis mainly focused the extensive use of TCE tool. After getting the design automation idea, we need to give focus for selection of processor class based on some benchmarks like cycle counts, simulation time, architecture structure etc.

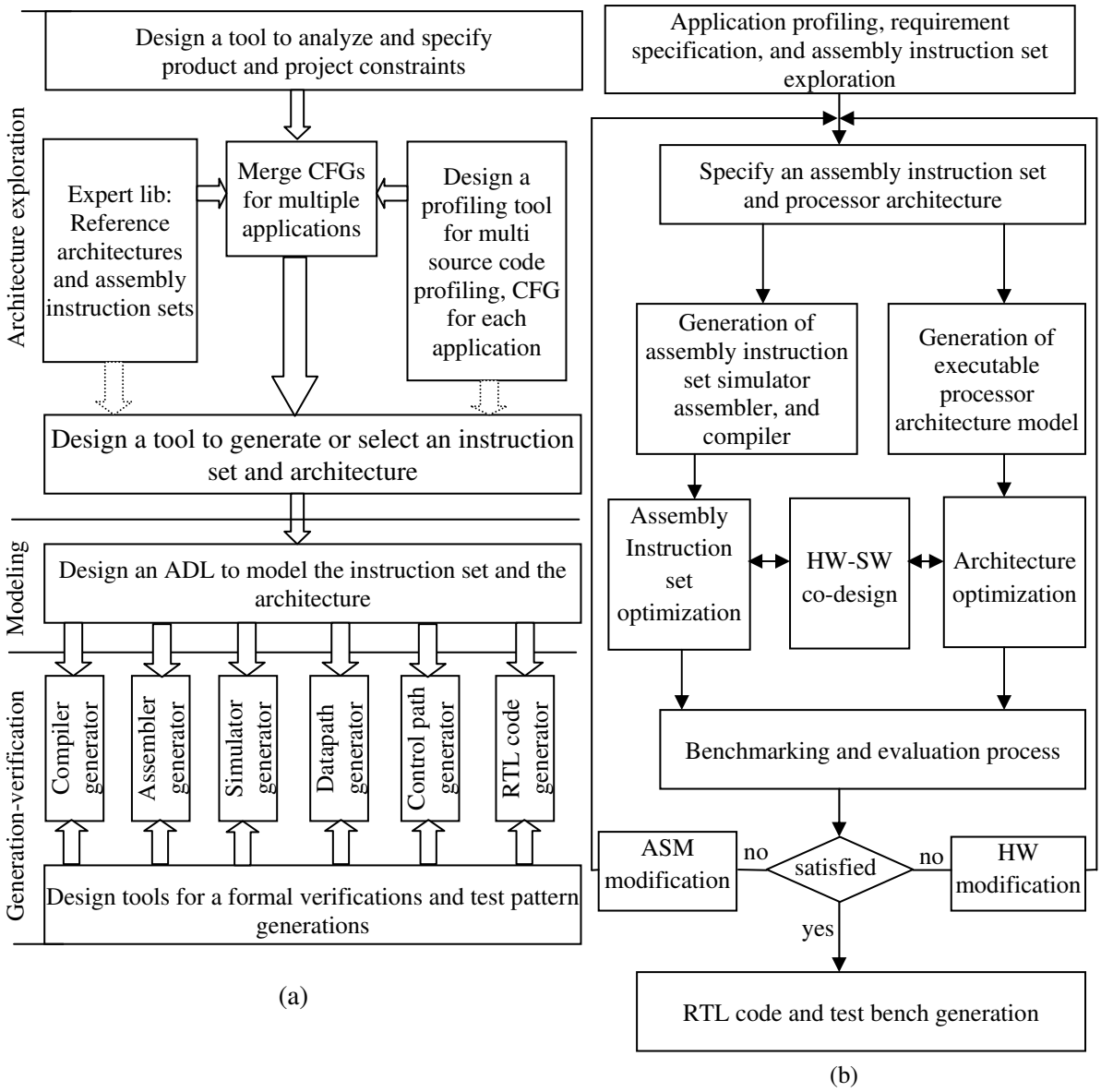


Figure 3: Automatic ASIP design flow (a) Tool researcher's view. (b) Designer's view [1].

2.2 Performance of Computer System

The performance of the computer system depends on the real time taken to accomplish a certain task or application by the system. This time is known as the elapsed or wall clock time. This elapsed time includes 1) the user time, 2) the system time, and 3) the time swapping and executing other processes [4]. In user time, the system executes instruction specified by the application and the system time required to handle operating system calls as requested by the application. In this thesis, I mainly interested to decrease the user time. There are some standard benchmarks like Dhrystone, SPECint and SPECfp used to estimate the performance of a computer. The performances of different GPPs are listed in ref [4]. If we see the performance of SPECint and SPECfp, a tremendous improvement in terms if issue rate was found in these benchmarks [4]. This was happening because of the factors determine user time of an application. So this time can be calculated from [4]

$$t_{user} = \#Instr \times CPI \times t_{cycle} \quad (1)$$

where *#Instr* is the number of instructions executed, *CPI* is the average number of cycles per instruction (*CPI*) and t_{cycle} is the cycle time. So order to increase the performance we need to decrease the factors contributing to the user time: *#Instr, CPI* and or t_{cycle}. There are three main developments which influences these factors [4]:

1. The improvement of VLSI technology, decreasing t_{cycle} and increasing number of transistors per chip.
2. There should be developments in pipelining instructions, instruction level parallelism, influencing t_{cycle}, #*Instr* and *CPI*.
3. Compiler developments, especially the exploitation of instruction level parallelism which influences #*Instr* and *CPI*.

These kind of developments are strongly related to VLSI improvements. The gradual progressive manner of VLSI revolution offers the possibility to put much more hardware on a single chip. This was allowing the implementation of multiple function units (FU) on a single chip. As a result, the CMOS feature size scaled down almost

20% per year. Therefore, chips are getting larger and the number of transistors per chip #*Trans* is increasing more than 50% annually. The achievable cycle time t_{cycle}, which is determined by the critical timing path of circuit and roughly estimated by [4]:

$$t_{cycle} \approx t_{gate} \times \#\, gate\_levels + wiring\_delay + pad\_delay \quad (2)$$

The *pad_delay* can be avoided by using the single chip fabrication and depending on the dimensions of mask-layers scale with the minimal feature size (*mfs*), the switching time of a gate t_{gate} reduces at least linearly with *mfs* [4]. It is possible to reduce the effective number of gate levels, #*gate_levels* using pipelining.

2.3 Overview of Architecture Developments

In the evaluation of VLSI technology, CISC was dominating in the decade of seventies. It is necessary for computer architecture to maximize the performance, or performance-cost ratio, through the perfect exploitation of VLSI capabilities. As it is mentioned in Eq. (1) that the performance of architecture will be improved by reducing the parameters of right hand side of Eq. (1). So, there are three techniques to improve the performance of the processor [4]: (Super)-pipelining, Powerful instructions, and Multiple instruction issue. Super-pipelining reduces the *CPI* and t_{cycle}. Pipelining is related to the execution of an instruction. So there are several steps are required for execution of an instruction. Those are fetching the instruction from memory, decode it, get the required operands, execute the specific operation and finally write back the result of the operation. These steps are known as the well-known *Von Neumann cycle*. For implementation of every instruction these steps are repeatedly occurred. In early of the seventies, CISC architecture took very long cycle time because of missing pipelined. If it is possible to overlap or pipeline the execution of instructions, then the throughput of instructions increases and therefore CPI decreases. However, this requires a streamlined instruction set, that means each instruction can be split into the same number of stages and each stage takes the same time and different hardware. So, this concept of execution is not possible in CISC;

therefore RISC evolved. RISCs have a reduced instruction set and support a very limited number of addressing mode like instructions fits well in a sample pipeline scheme. In principle, RISCs can issue one instruction each cycle and giving a theoretical CPI of one. There is another pipelining concept to reduce the cycle time. This is known as superpipelining. Using superpipelining *#gate_levels* can be reduced in critical path [4]. The result of RISC pipelining is interpreted as to reduce the *CPI* close to one but superpipelining decreases t_{cycle} and in fact superpipelining lead to increase of *CPI*. Besides using the pipelining and superpipelining concept, the processor configuration can reduce the number of instructions by adding more powerful instructions to the processor's instruction set. Powerful instructions are performing more work per instruction. There are two techniques for applying powerful instructions. The first one is MD-technique results in data parallel architectures and the second one is MO-technique results in operation parallel architectures. CISC architectures already applied both techniques in limited extend. The MD-technique is multiple sets of data operands per operation. That means one operation is applied to multiple set of data operands. In MD-technique, vector and SIMD (single instruction multiple data) processors both exploit the use of multiple data operands per specified operation. Both configurations implement the data parallelism differently. For example, vector processors execute a vector operation by applying this operation to a vector of data elements sequentially in time. In SIMD processors, it applies the operation concurrently to all the data elements. Figure 4 shows the execution method of vector and SIMD execution. This figure portrays both types of data parallel execution and shows how instructions are executed on a vector processor with *K* FUs and on a SIMD processor with *K* nodes. In the vector processor, each instruction uses only one FU and has a very long execution time. If the required resources are available then the next instruction can be issued even the previous instruction is still executing on different FU. Similarly, an SIMD processor executes instructions one at a time and each instruction may require all the available nodes. The

later case MO-technique is multiple operations per instruction. MO-technique is exploited by VLIW processors that have horizontally encoded instructions. Each instruction consists O fields, where O is the number of operations which can be executed concurrently. VLIWs have much in common with SIMDs. Both architectures accept a single instruction stream and each instruction specifies many operations. Although it seems more complex but it may reduce the *#Instr*. The following properties show the basic difference VLIWs and SIMDs architectures [4]:

- VLIWs can implement any mixture of FUs.
- VLIW instructions allow the different types of operation within a single instruction.
- VLIWs exploits fine grain parallelism i.e. parallelism that exists in a very small scale signal operation.
- In order to exploit a very fine grain parallelism, VLIWs requires a large communication bandwidth between FUs. In general, FUs use the register file to communicate.
- VLIW instructions are large compared to SIMD

The former three characteristics are very useful properties of VLIW and suitable for designing application specific processor.

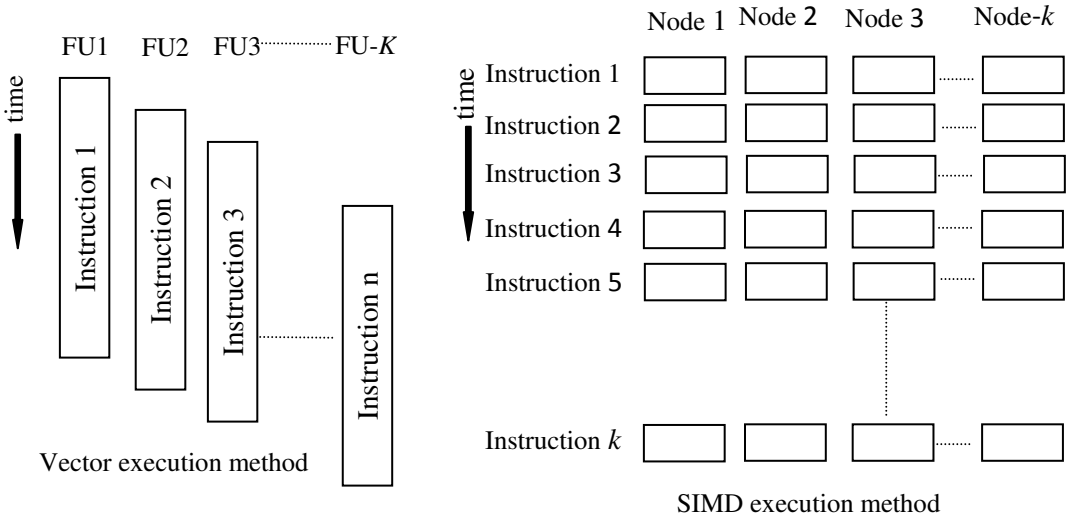


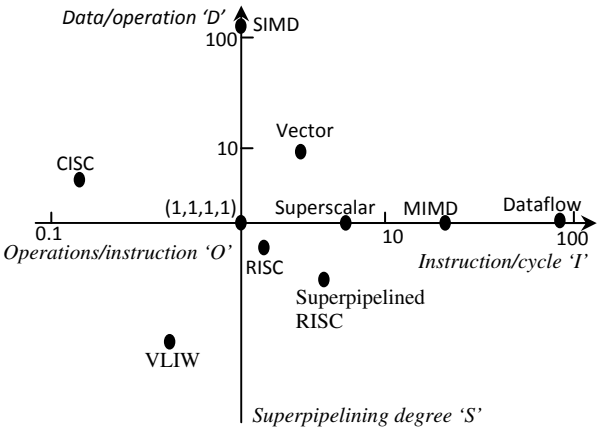
Figure 4: Data parallel operation [4].

2.3.1 Multiple Instruction Issue

In order to gear up the processor speed, multiple instruction techniques are very powerful idea, which means multiple instructions per cycle. Multiple Instruction Multiple Data (MIMD) processor has the capabilities to look ahead in the stream in order to detect multiple instructions which can be issued concurrently. Recently, multiple instruction issue architectures have attempted to improve processor performance by fetching and dispatching more than one instruction in each processor cycle. This capability is known as superscalar. In MIMD processors, communication between two instructions is extremely specified by the instruction themselves.

2.3.2 Architecture Design Space

The former explanations presented the different techniques to enhance the performance of the computer architecture. In this section, I will present the design spaces to explain the processor architecture. Each architecture of the processor can be specified as four variables like : *I* is the issue rate (instruction per cycle), *O* is the number of operations specified per instruction, *D* is the number of operand pairs to which the operation is applied and finally *S* is the superpipelining degree. Figure 5 shows the four dimensional representation of processor architecture [4].



(a)

Architecture	K	I	O	D	S	M_{par}
CISC	1	0.2	1.2	1.1	1	0.26
RISC	1	1	1	1	1.2	1.2
VLIW	10	1	10	1	1.2	12
Superscalar	3	3	1	1	1.2	3.6
Superpipelined	1	1	1	1	3	3
Vector	7	0.1	1	64	5	32
SIMD	128	1	1	128	1.2	154
MIMD	32	32	1	1	1.2	38
Dataflow	10	10	1	1	1.2	12

(b)

Figure 5: Architecture design space (a) Four dimensional representation
(b) Typical values of design spaces for different architectures[4].

From the figure 5 (a), RISC architectures are very close to the center (1,1,1,1) of the architectural design space. That means for RISC processor, the potential issues are consisted like one instruction per cycle (I = 1), where each instruction specifies one operation (O = 1), each operation applies to a single or a single pair of operands (D = 1), and the superpipelining degree equals to one ($S \approx 1$). Figure 5 (b) shows a table that represents the typical values of (I,O,D,S) for different processor configuration. Here, the amount of parallelism M_{par} and values of K, the number of FUs are also calculated for every processor architecture. This amount of parallelism M_{par} is calculated as the following equation defined by [4]:

$$M_{par} = I \times O \times D \times S \tag{3}$$

As portrayed in figure 5, to achieve a high M_{par} these four orthogonal techniques can be combined to create a hybrid architectures. One question can be raised that what should be the best combination of design space parameters (I,O,D,S)-tuple for getting maximum benefit of parallelism? The answer of this question depends on the application domain. Therefore, processors for different application domain have different architectures and amounts of parallelism.

2.4 Application Oriented Architecture

To satisfy the high requirement demands from users, it is necessary to increase the performance of hardware system. There are some issues that cause these requirements to increase the demand of application oriented architectures. Those issues are described as below[4]:

Functionality: That means, the functionalities of user applications are increasing day by day.

Larger data sets: In order to get better accuracy or more compatibility with physical reality programs are applied to larger data sets.

Merging with new domains: Sometimes, the processor configurations should be required to compatible with new eras like neural networks, expert systems, genetic algorithm based applications and so on.

Real time requirements: Several applications require making analysis with real time signals like real time image and signaling processing, control systems etc.

In order to meet the above requirements, it requires for computer architect to increase the degree of parallelism M_{par}. However, all times a parallelism M_{par} does not guarantee a speedup of architecture. So target speed is largely application dependent. The following discussion shows the difference between different application domains:

Scalar domain: This is considered as the general purpose computing domain where the compilers, text formatters and symbolic programs are mainly used. In this scalar domain, programs may use many pointers, allocate heap area, and spend a lot of time in the operating system. In these program most of the operations are 32 bit integer based operations rather than floating point operation.

Vector domain: Programs in this vector domain use many operations like scientific and highly numeric applications based functions. Typical operation in this domain is is the dot-product on double precision floating point vectors.

Application specific domain: The performance of the processor configuration greatly depends on the specification of input application. For example, signal processing applications fit into this domain. The nature of operations in this domain may be integer based and also floating point as well.

From the above discussion, it can be said that the processor configurations supporting these three domains are called *general purpose*, *super or vector* and *application specific processor* respectively. Among them, *application specific processor* domain has more exploitable parallelism as compared *to scalar or vector processor* domain.

There are two types of exploitable parallelism as below:

Operational parallelism: This kind of parallelism occurs between different operations of a single threaded program.

Data Parallelism: This parallelism exists when one or more operations can be applied to many data elements in parallel. It focuses on distributing the data across different parallel computing nodes. In a multiprocessor system, data parallelism is achieved when each processor performs the same task on different pieces of distributed data. In some situations, a single execution thread controls operations on all pieces of data. In others, different threads control the operation, but they execute the same code.

To explain the difference of operational and data parallelism, let us take a function **F** to a vector **b** and assigns the result to array vector **a** :

```
for i from lower_limit to upper_limit
do a[i]= F(b[i])
```

If the vectors **a** and **b** does not overlap then operation **F** can be applied to all set of data in parallel. That all iterations in this loop can be executed concurrently. Therefore, this is called as data parallelism. On the other hand, operation level parallelism depends on the number of operations executed during the compilation of **F**. In general, all programs contains limited amount of operation parallelism besides the data parallelism in scientific and application specific domain.

According to the characteristics of the *Single Instruction Stream Computers* (SISCs), the parallelism can be described as per the orientation of processor. So this types of parallelism can be written as below:

Instruction level parallel processors (ILPPs): The main aim of ILPPs is to support the exploitation of operational parallelism. Under this category, processors have multiple FUs those are usually used to support different types of operations. Besides this ILPPs apply the superpipeling technique that means *I, O,* or *S* are greater than one. Superscalar, VLIWs, superpipelined, dataflow processors and processors using TTA are usually belonged to ILPPs category.

Data level parallel processors (DLPPs): This kind of processors supports data parallelism. In DLPPs, the value of *D* is in the range from tens to thousands and *I* and *O* are usually one. SIMD and vector processor are the example of DLPPs.

Operational parallelism is not much easy like data parallelism. Operational parallelism is limited but it is available in every processor operation. Therefore, this technique being exploiting the parallelism will always increase performance of processor. As it is mentioned earlier that ILPPs use operational parallelism but it also gets benefit from data parallelism as well [4]. However theoretically, ILPPs are more powerful than DLPPs. Nevertheless, there are some complexities in current ILPPs like VLIW and I will discuss in next chapter regarding this complexity. For this reason, this current ILPPs do not allow a very high degree of parallelism. The solution of this problem is to bring the concept of transport triggered architecture (TTA) which will be discussed in later.

On the other hand, depending on the supported application, application specific processors (ASPs) can exploit both types of parallelism. In ASPs it is possible to eliminate unnecessary features like, virtual memory, high precision integer to floating point support cache memory etc. For this reason ASPs require less power, reduce complexities and allow to support higher M_{par} values or same M_{par} value at lower cost. In case of SISCs, it becomes also powerful by exploiting both types of parallelism. But it creates problem when the control flow of a program is strongly data dependent. To answer this problem multiple instruction stream computers (MISCs) may be the solution to the high power demand. It contains many nodes, which exploit operation or data parallelism. In this thesis, I will mainly exploit the TTA architectures and its implementation tool, so the discussion on MISC is not further explained.

2.5 Parallel Computing: Amdahl's Law

Amdahl's law is also known as Amdahl's argument and people who practice the parallelization of code all experienced Amdahl's law. This is used to find the maximum expected improvement to an overall system when only part of the system is improved. It is often used in parallel computing to predict the theoretical maximum

speed up using multiple processors. So, Amdahl's law states that the speedup achieved when parallelizing an application using N processor is limited by [4]:

$$Speedup \equiv \frac{\text{serial processing time}}{\text{parallel processing time}} \leq \frac{1}{f/N + 1 - f} \tag{4}$$

where f is the fraction of program that can be parallelized and the serial function $1 - f$ cannot be parallelized. So, from the equation 4, it will give a relationship between number of processors and overall speedup. It is not true that if we apply parallel processors or increase the value of N then no matter speedup will be increasing linearly with respect to N. There should be a certain point after that the speedup will be independent with respect to N. The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. For example, making a microprocessor twice does not mean that the computer system shows a speedup of two. It depends on the number of parallel portion of the executed program.

2.6 Complexity of Instruction Level Parallel Processors

In previous sections, I have discussed about the instruction level parallel processors but it has several limitations. Due to the complexities of design, it will take long time to market and high cost. Therefore, in this chapter, I will discuss the nature of complexities for implementing the ILPPs. The VLIW and superpipelined processors are traditional ILPPs. In these processors, they clearly illustrate what happens to the data path complexity when adding function units or increasing pipelining.

2.6.1 Data Path Complexity

In general there are several steps required to execute an instruction and as it is mentioned earlier that these steps are known as *Von Neumann Cycle*. These steps are explained as below:

Instruction fetch: Instructions are fetched from the instruction or cache memory. Fetching instructions is a main bottleneck due to the relative slow access times. This slow access time can be reduced by perfecting instructions before the processing unit requires them. The prefetched instructions are loaded into a prefetch buffer where they are retained until needed by the processor.

Instruction decode: The instruction decode unit decodes and sequences all instructions and depending upon processor, it also includes debug control coprocessor, instructions and system control coprocessor etc. For example, in ARM cortex architecture, the instruction decode unit handles the sequence of exceptions, debug events, and memory built in self test (MBIST) etc.

Issue the instruction: If the required resources are available and possible data & control hazards are resolved then an instruction can be issued.

Operand fetch: It fetches the required source operands and each operation may require zero or more source operands. It may require complex address arithmetic to fetch operands from data memory.

Execute: This stage performs the operations specified in the instruction. For example in ARM cortex, the instruction execute unit consists of two symmetric Arithmetic Logical Unit (ALU) pipelines, an address generator for load and store instructions, and the multiply pipeline. The execute pipelines also perform register write back.

Write-back: This stage writes the results of the operation to the locations specified by the destination operand.

Using two source operands, most operations deliver only one result value. But for preparing the result value, it may require multiple succeeding operations. This section I discussed the basic steps for executing one instruction. In next section, I will explain the data path flow of a non pipelined processors.

2.6.2 Non-Pipelined Processor

The above steps are executing in a sequential manner for non-pipelined processor. For example, the instruction fetch stage has to wait for the next instruction until the write-back of the current instruction has been completed. Figure 6 shows basic data path of a simple non-pipelined processor. In this data path, a general purpose register file (RF) is used for the operand and result values, and the task of specified FU is to perform the required operations fixed by the instruction set of this processor. A simple FU contains one output port and two input ports. To make this data path simple I did not include the immediate register and special purpose register like program counter in Fig. 6. So these RFs are also used as source and destination registers. Figure 6 (b) shows connectivity graph (CG) of the a simple non pipelined processor. The CG of processor is a bipartite graph required to mention the data transport in data path. The definition of CG is discussed in Appendix I. Therefore, this CG has a related architectural complexity. For the given data path showed in figure 6 (b), the architectural complexity is [4]

$$AC_{compl}(\text{non - pipelined}) = (N+5, N+5, 3N+4) \tag{5}$$

where N equals to the number of general purpose register. Although the connectivity graph shows the connection between source and destination, it does not tell how to implement this connectivity. Since, there are many options for the data path to implement this connection, so this architectural complexity fails to indicate the real measurement of complexity. It requires another quantities of complexity: the bus complexity and data path complexity those are described in Appendix I. As implementation of any connectivity graph requires at least one shared read write bus. This causes in a non-pipelined processor, because a maximum of only one data transport per cycles is supported. Figure 6 shows such an implementation of data path. Therefore, the data path complexity for this non-pipelined processor can be determined from figure 7 that shows a different view of the data path including all the

necessary read and write connections. It can be known as the connectivity model of the processor. Therefore, the data path complexity for this non pipelined processor is given by:

$$DP_{compl}\,(\text{non}-\text{pipelined}\ ) = (4, N+4, N+3, 1, N+4) \tag{6}$$

where DP_{compl} means the data path complexity. From figure 7, equation 6 can easily be derived and in which the maximum number of read connections to any bus and maximum number of write ports to any register is $N+2$ and 1 respectively. So there is no register with more than one write port.

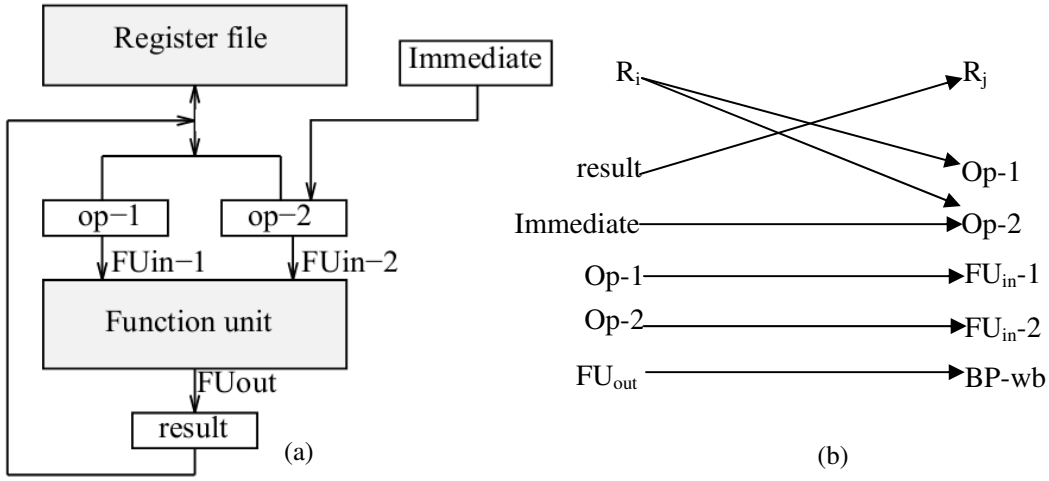
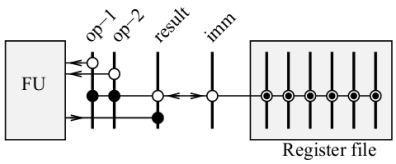
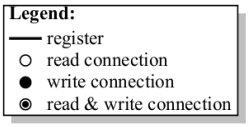


Figure 6: Data path and connectivity path of a simple non-pipelined processor (a) Data path (b) Connectivity graph [4].





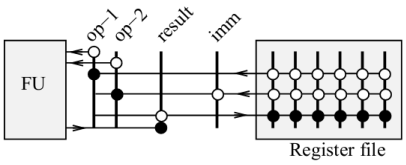


Figure 7: Connectivity model of a non-pipelined processor [4].

Figure 8: Connectivity model of a pipelined processor [4].

If I analyze figure 7, it has four buses and among them three are very simple: they only serve to connect FU and outputs to corresponding to source operand and result registers.

2.6.3 Pipelined Processor

There should be changed in connectivity model if we apply the pipelined feature in processor. Figure 8 shows the connectivity mode for pipelined processor. Figure 8 shows that it has 6 buses: the source operands and the FU result value have to be read concurrently from and written to the RF. Its architectural complexity is same as the non-pipelined architecture. Nevertheless, its data path complexity can be written as below:

$$DP_{compl}\,(\text{pipelined}) = (6, 2N+4, N+3, 3, N+4) \tag{7}$$

If I compare equation 7 with equation 8 that means compared to the non-pipelined, pipelining contributes the following $DP_{coml.}$:

$$\Delta DP_{compl}\,(\text{pipelined, non} - \text{pipelined}) = (2, N, 0, 2, 0) \tag{8}$$

From equation 8, the complexity of the pipelined processor is addition of extra buses and the corresponding to the register ports.

2.7 Implementation Details of RISC Processors

The pioneer development in designing computing system is the change of architectural design from CISC to RISC principles. This change shows that the extra functionality does not always decrease the execution time. On the other hand, CISC may increase the execution time. Sometimes, the extra functionality may add critical timing path within a processor, which increases the cycle time. It requires complex pipelining scheme for complex instructions. Similarly, this complex hardware will increase the design time. Hence, product cost & time-to-market will be increases [4].

RISC processors have only a small number of instructions compared to a CISC. The instructions are also smaller in size with a smaller number of fields and usually fixed length. Most instructions have the same format with limited number of addressing modes, which are executed by hardware. RISC processors have an instruction cache, a data cache, only load and store instructions reference memory [5]. The main bottleneck of RISC is to pipeline the execution of instructions, which reduces the *CPI*.

RISCs pipelined the *Von Neumann Cycle* and performed each step in a single cycle. As a result, the execution of each step has to be time balanced and the complexity of each step should be reduced. RISC has done pipelining through different ways: caching, uniform instruction format, large RF, one simple operation per instruction. On chip, caching for data and instruction reduces the time for instructions and data to single cycle. Due to the single instruction size, RISC reduces the instruction decoding time and complexity. RISC has large RF and most operations use operands located in registers only. Operand fetch and write back steps are performed very easily in a single cycle. For this reason in RISC the instruction set can easily be pipelined. Figure 9 shows the pipelining diagram of the simple RISC processor. It has four pipeline stages: IF, DC, EX and WB stages. During the decode stage the instruction is decoded, issued, and concurrently the source operand values are fetched from the RF. During the execution stage all operations including the memory access operations are performed. In RISC processor, the data move instructions support only one additional addressing mode; besides the register-direct addressing mode supported by all operations, data moves may address one memory operand, using the resister indirect addressing mode [4]. Using the pipeline showed in figure 9, for a RISC architecture *CPI* equals to 1. However, absence of precautions the value of CPI may increase because of hazards and cache misses. There are three types of hazards: structural, control and data hazards. Because of insufficient hardware to fulfill the requirements of all instructions in the pipeline, structural hazard may occur. For example, a separate memory access path required to avoid the structural hazard between execute and instruction fetch stage. Instructions changing in the program counter can create the control hazards. For example, the address of the next instruction is not known at the end of the current instruction fetch stage. To solve this problem, branch target buffer (*BTB*) can be well known solution. A *BTB* is not visible to the architectural level. Instructions from specified address can be fetched and executed without changing the state of the processor. If the processors are strongly depending on the data dependent

operations then data hazards may occur. For example, in figure 9, instructions *i* and *i* + 1 have data dependency condition. That means instruction *i* +1 uses the result of instruction *i*. Therefore, the decode stage of *i* +1 has to be locked until cycle 5. Here 2 cycles are lost and the effective latency of an operation is 3 cycles. The result of instruction *i* would not be available until instruction *i* + 3. Therefore, in this case the instruction *i* has two delay slots. The compiler can solve this latency problem by putting the two independent instructions between this dependent time. This is not all time very easy task for compiler and this problem is getting worse to exploit the instruction level parallelism.

This data hazard problem can be solved by implementing so-called bypass circuit in the configurations. This bypass circuit can directly forward the result value to the execution unit. Therefore, it is bypassing the RFs hence known as bypass circuit. In figure 9, this direct forwarding is shown by indicating the arrow mark. Figure 10 shows the data path and figure 11 represents the connectivity graph including bypass circuit for simple RISC processor [4]. The FU executes all the arithmetic, logic and memory operations including load and store operations by using ALU (arithmetic logic unit) and memory unit.

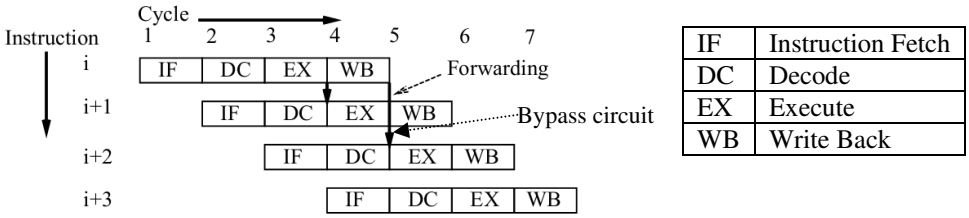


Figure 9: Four stage RISC pipelining diagram [4].

The FU takes data from two registers: op-1 and op-2, which can be fed by data from the RF or bypass circuit. The BP-wb (bypass write back) register is used to hold the result value for one cycle. After applying the bypass circuit, the architectural complexity for this RISC processor is equal to:

$$AC_{compl}(\text{simple} - \text{RISC}) = (N+5, N+5, 3N+8) \quad (9)$$

For applying, the bypass four extra connections are required without changing the number of source and destination nodes.

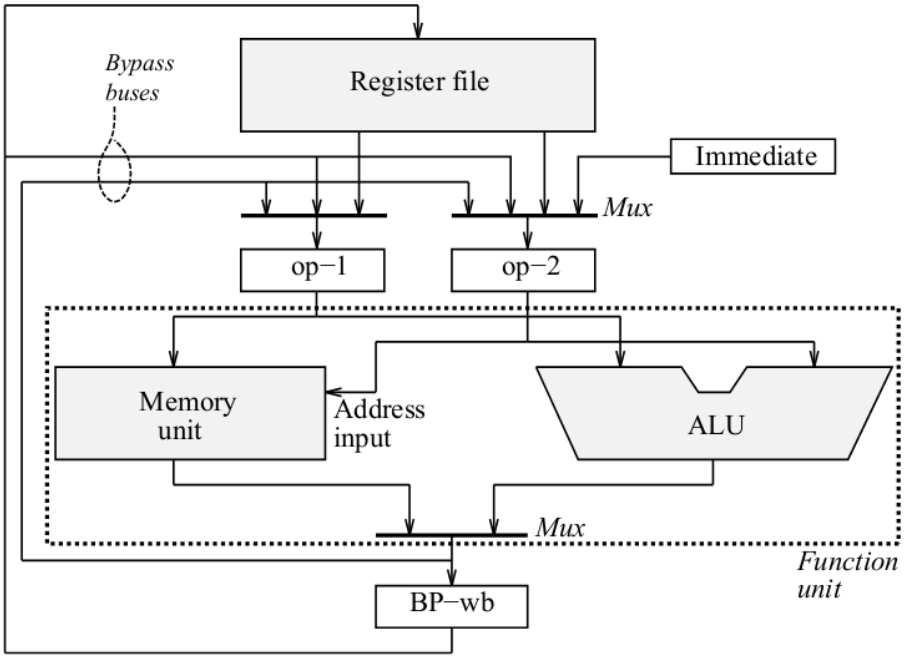


Figure 10: Data path of RISC processor [4].

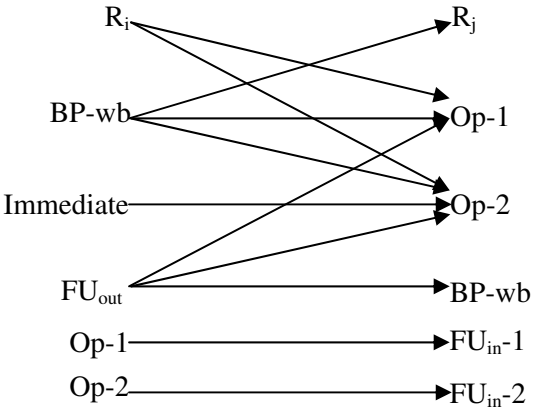


Figure 11: Connectivity graph of RISC processor.

Figure 12 shows the connectivity model of this RISC processor. It has divided into three parts: FU, Bypass and RF. Bypass circuit contains source operand registers,

bypass write back register and their connectivity. From figure 12, the data path complexity can be written as below:

$$DP_{compl}\,(\text{simple} - \text{RISC}) = (7{,}2N + 4, N + 8{,}3, N + 4) \tag{10}$$

So, the differential data path complexity between bypassing and without bypassing is given by [4]

$$\Delta DP_{compl}\,(\text{bypass, without bypass}) = (1, 0, 5, 0, 0) \tag{11}$$

where maximum read connection and write port are $RC_{max} = N$ and $WP_{max} = 4$ respectively. The real difference is restricted to four write connections.

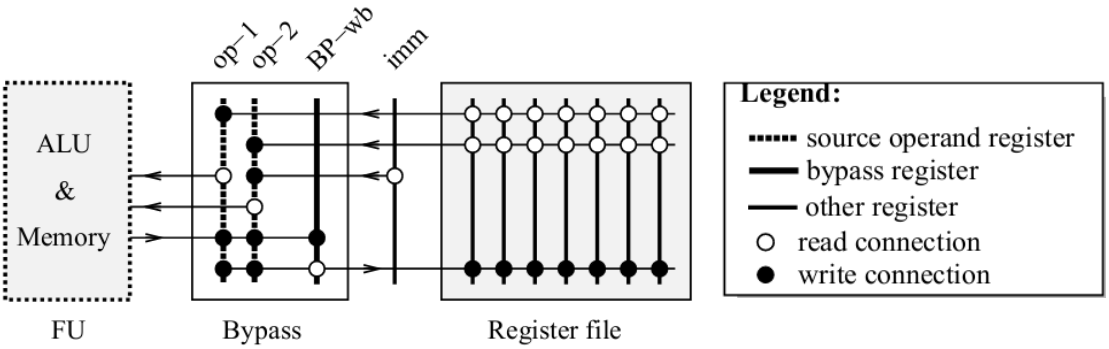


Figure 12: Connectivity model of a RISC processor [4].

2.7.1 Superpipelined Architecture

In order to reduce the execution cycle time, superpipelined architectures extend the pipeline concept like instruction fetch, execute and memory stages are pipelined in its configuration. In superpipelined architectures, the execution stage is divided into S sections and depending on S there are two types of latencies: equal latency and non-equal latency. In execution stage, for equal latency, all operations require S execution cycles. The connectivity model for superpipelined architecture is same as simple RISC processor and therefore the data complexity is also same as RISC processor.

For non-equal latency, the FU supports operations having different latencies upto S. Let us assume that FU of superpipelined processor supports operations of all possible latencies $L, L \in 1, 2, 3..., S$. The data path of superpipelined processor is shown in Fig.

13. Figure 14 showed the connectivity model for *S*-stage superpipelined processor. So the data path complexity for this processor is given by [4]:

$$DP_{compl}\text{(superpipel ined)} = (8+S, 5+2N+S, 7+N+3S, 3, N+3+S) \quad (12)$$

Therefore, by differentiating equation 12 with respect to *S* it is possible to calculate complexity added in each extra pipelining stage.

$$\partial DP_{compl}\text{(superpipel ined)} / \partial S = (1, 1, 3, 0, 1) \quad (13)$$

The bypass network complexity grows linearly with the number of superpipelined stages but it does not increase the complexity of RF unit.

2.7.2 VLIW Architecture

As it is mentioned in previous section that superpipelined processors exploit internal FU concurrency mentioned in figure 13 for reusing its hardware multiple times. Instead of internal FU concurrency, VLIWs exploit external FU concurrency where it contains multiple FUs and each FU supports RISC style operations. So, each VLIW instruction specifies multiple RISC operations. Figure 15 shows the data path of VLIW processor for two single cycle FUs. From Fig. 15, FUs share a bus for immediate values. That means only one immediate can be specified per instruction. Figure 16 shows the connectivity model for K single cycle FUs. The data path complexity for VLIW processor is given by [4]:

$$DP_{compl}(VLIW) = (1+6K, 1+3K+2NK, 4K+NK+4K^2, 3K, 1+N+3K) \quad (14)$$

Differentiating equation 14, the additional complexity for each extra FU is given by:

$$\partial DP_{compl}(VLIW) / \partial K = (6, 3+2N, 4+N+8K, 3, 3) \quad (15)$$

where $RC_{max} = N$, $WP_{max} = 2 + 2K$, and $\#BP_{cmp} = 4K^2$.

Therefore, the bypassing network complexity equals to the square of the number of FUs. The bypassing time is linearly proportional to the function of *K*. So, adding more FUs will increase the complexity of the VLIW processor.

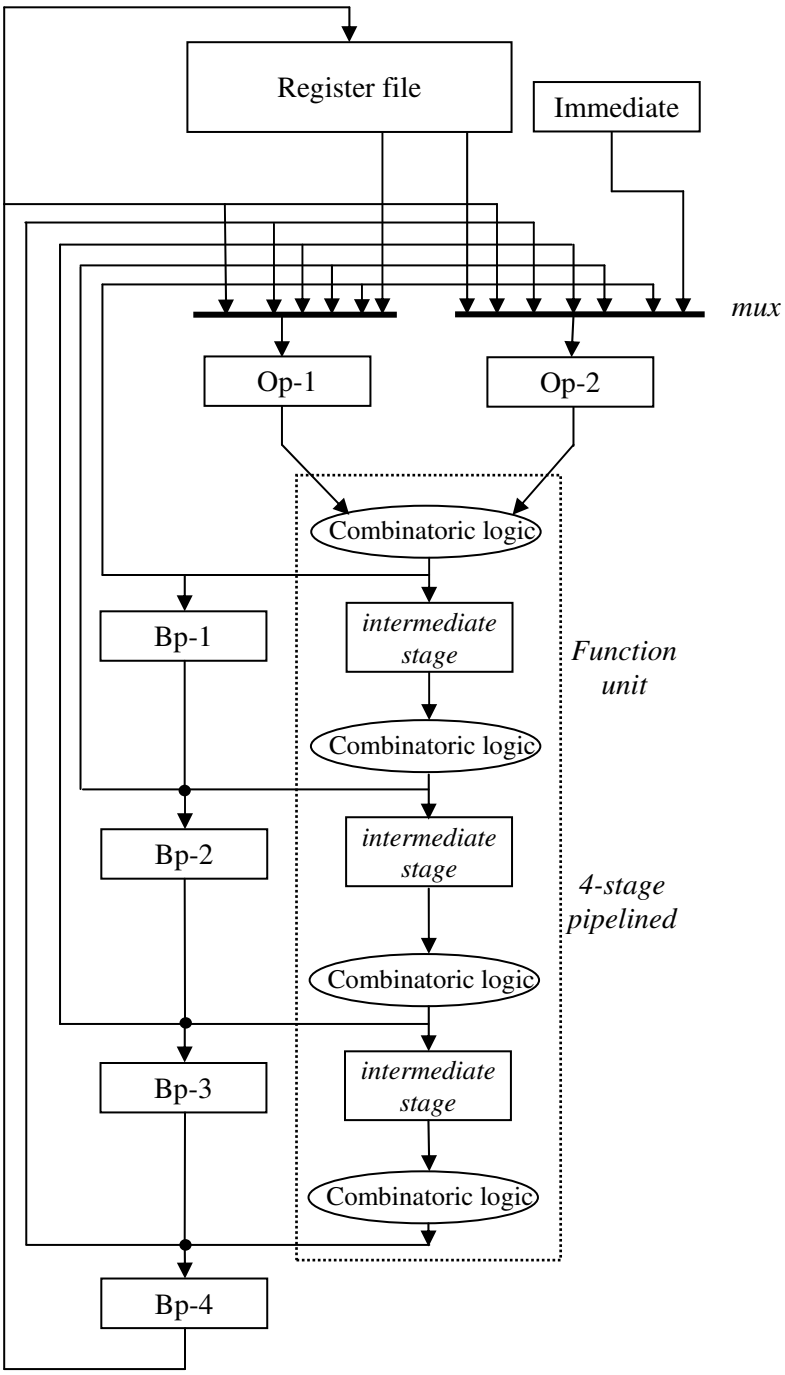


Figure 13: Data path of a four stage superpipelined processor[4].

2.7.3 Comparative Study on VLIW and Superpipelined Architectures

From the discussions of the previous sections, a VLIW and superpipelined architectures have similar nature of behavior. For example, for both architectures the compiler has to search for independent operations which can be scheduled into one VLIW instruction or pipelined fashion. The characteristics of superpipelined processor are given below:

- It uses the hardware resources efficiently.
- There is no classification of FU. So, for similar types of operational executions, there is no chance of FU conflicts.
- It has scheduling advantage [4].
- Additional latency occurs during the operations of non-numeric scalar code.
- In superpipelined architecture, its performance is limited by clock and data skew and its bypassing complexity is linearly proportional with *S* [4].

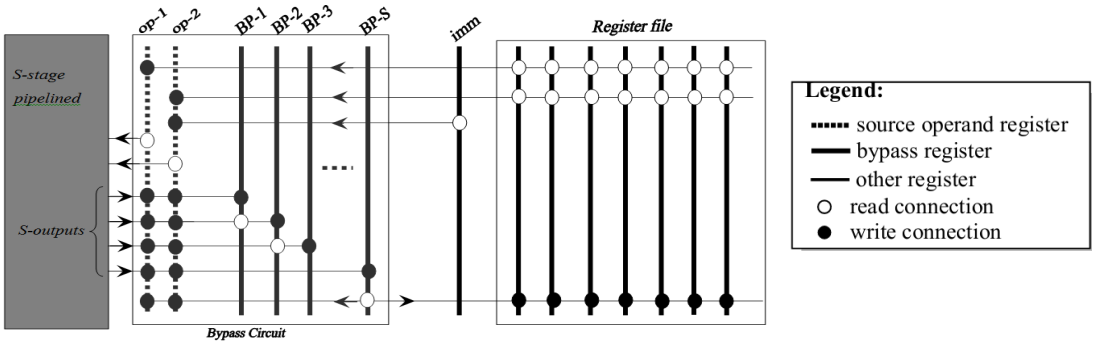


Figure 14: Connectivity model of an *S* – stage superpipelined processor [4].

VLIW architectures are characterized by instructions that each specify several independent operations. This is compared to RISC instructions that typically specify one operation and CISC instructions that typically specify several dependent operations. The characteristics of VLIW architectures are given below:

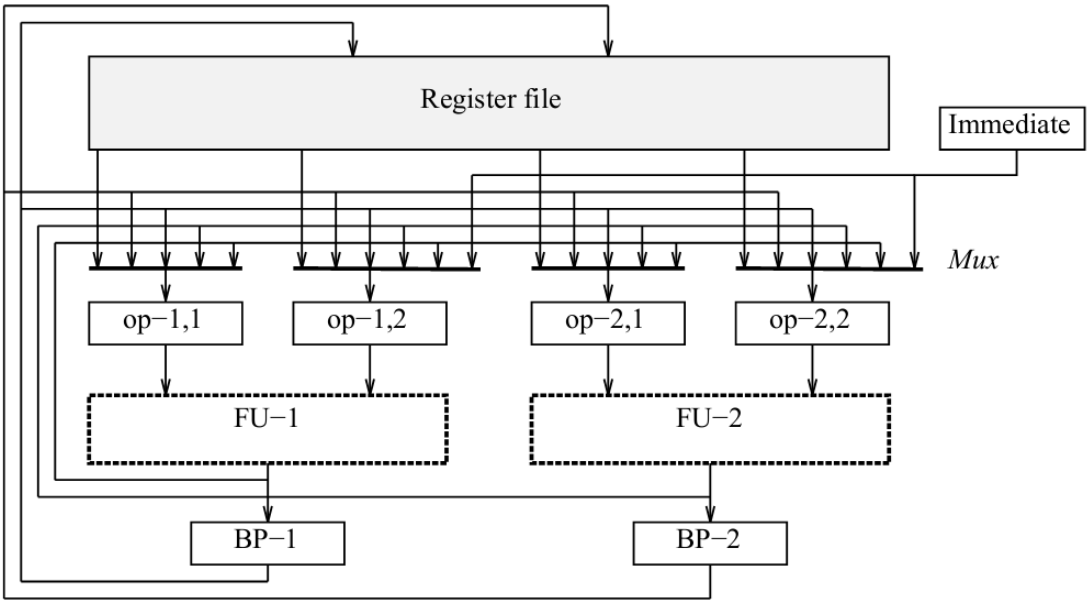


Figure 15: Data path diagram of VLIW processor with two FUs [4].

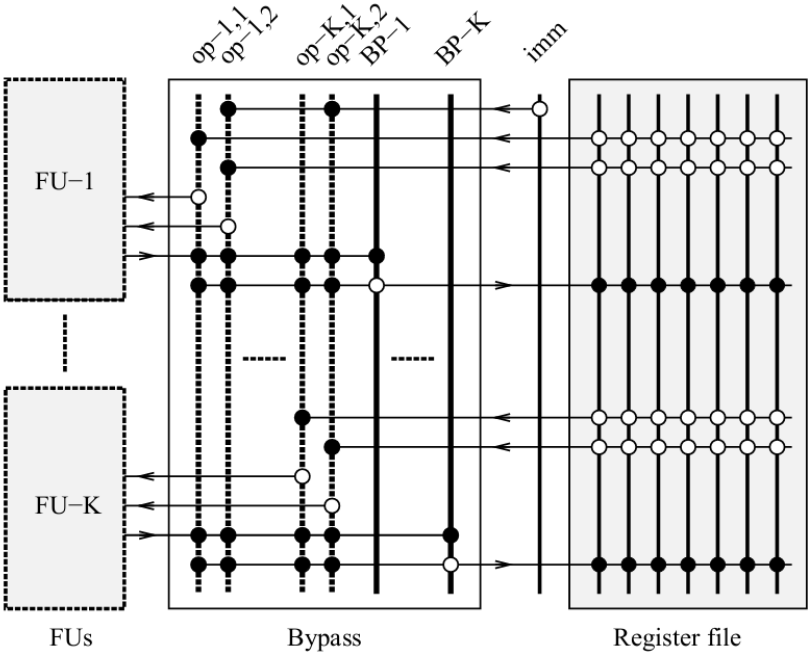


Figure 16: Connectivity graph of a VLIW processor with K FUs [4].

- For using the scalar code applications, VLIW configurations are suitable processor because it has no latching overhead.

- It uses different concurrency operation of FUs like integer adders, floating point adders, shifters (left and right) etc.
- Number of FUs K is strongly limited by hardware constraints.
- Bypass and RF complexity are defined as $O(K^2)$ and $O(K)$ respectively.

From the above characteristics, the combination of superpipelined and VLIW principles leads to a very powerful processor. It will support both vector and scalar code based on the specific applications. Figure 17 shows the connectivity model for combined processor technique of VLIW and superpipelined principles with K S cycle FUs. So the data path complexity is given by [4]:

$$DP_{compl}(\text{superpipelined} - \text{VLIW}) = \begin{pmatrix} 1+(5+S)K, 1+(2+2N+S)K, (3+N+S)K+2(S+1)K^2, 3K, \\ 1+N+(2+S)K \end{pmatrix} \tag{16}$$

Differentiating equation 16, the additional complexity for each extra FU is given by:

$$\partial DP_{compl}(\text{superpipelined} - \text{VLIW}) = (5+S, 2+2N+S, 3+N+S+4(S+1)K, 3, 2+S) \tag{17}$$

$RC_{max} = N$, $WP_{max} = 2 + (S + 1)\ K$ and the number of bus complexity $\#BP_{cmp} = 2(S+1)K^2$. Though the superpipelined VLIW has high performance but it is suffering of bypassing network complexity for larger value of K or S.

From the evolution of processor from CISC to superpipelined VLIW, for exploiting the large amount of concurrency, the complexity of bypass and RF components depends on the number of external FUs supported by the processor. For this superpipelined VLIW architecture, the area and timing parameters are a function of S and K during the fabrication process. The bypass complexity can be defined as following equation:

$$BP_{compl} \equiv (\#Bus, \#RC, \#WC, WP_{max}, \#Regs, \#BP_{cmp}) \tag{18}$$

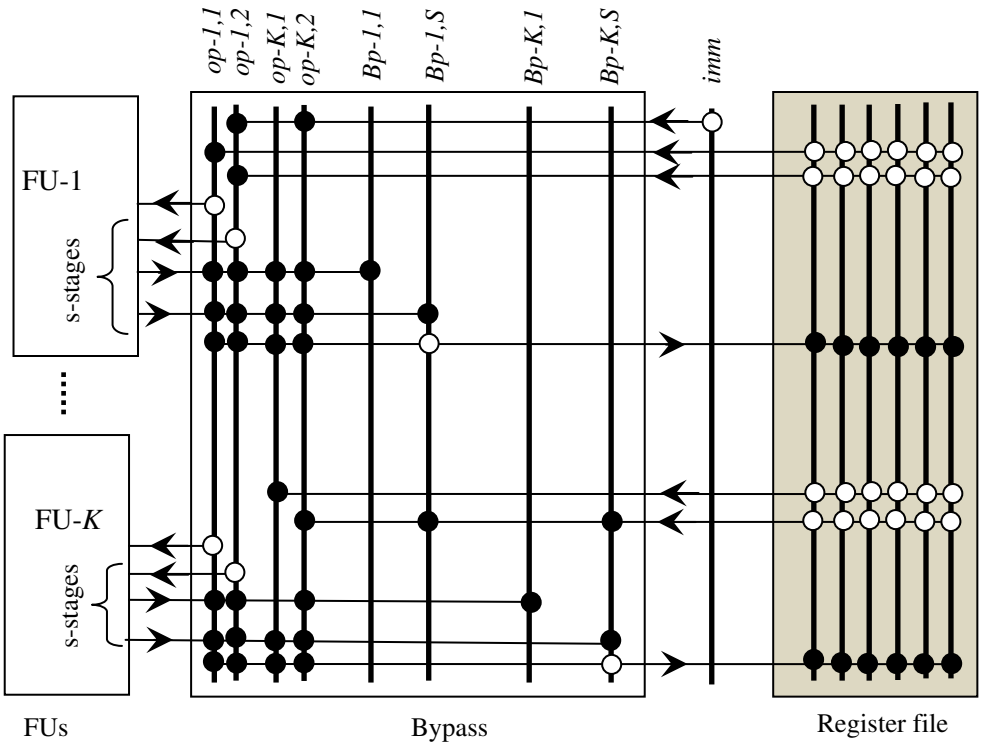


Figure 17: Connectivity graph of a superpipelined VLIW processor with *K S* cycle FUs [4].

Table I(a): Summary of Bypass and RF complexity for different architectures [4].

Architecture Name	Bypass Complexity						Register file
	#Bus	*#RC*	*#WC*	WP_{max}	*#Regs*	$\#BP_{cmp}$	*#RP*
Simple RISC	2	1	5	2	3	4	3
Advanced RISC	3	2	11	3	5	9	3
Superpipelined	$S+1$	S	$3S+2$	$S+1$	$S+2$	$2S$	3
VLIW	$2K$	K	$4K^2+K$	$2K$	$3K$	$4K^2$	$3K$
Superpipelined VLIW	$K(S+1)$	KS	$2K^2(S+1)+KS$	$K(S+1)$	$K(2+S)$	$2K^2(S+1)$	$3K$

Table I (a) shows the bypass and RF complexities for different architectures. From this table I (a) it can be said that VLIW and superpipelined processors have several good features like capability to exploit instruction level parallelism and suited for application specific operations by tailoring their functionality.

However, they are not fully scalable for large number of FUs. In next chapter, I will be discussing the different architecture to solve this problem and fully scalable to huge number of FUs.

Chapter 3

Transport Triggered Architecture (TTA)

In the previous chapters, I have discussed that how the instruction level parallelism becomes one of the major architectural methods to increase the execution speed of single processing nodes. Superpipelined VLIW and VLIW are the main processors for exploiting this type of parallelism. VLIWs are more dominating because they avoid the large run-time control overhead of superscalar and dataflow processors. The performance of VLIWs is high because it has multiple FUs for executing operation concurrently. Moreover, VLIWs exploit pipelining and their FUs can further superpipelined.

3.1 VLIW to TTA

To improve the execution speed of processor, exploiting concurrent execution of instructions which is known as Instruction Level Parallelism (ILP) is very important. This is an attractive approach to satisfy the high performance requirements. There are two main categories for exploiting the ILP. First category is like traditional CPU, such as superscalar processor and it can exploit the ILP at run time. This type of architecture is known as EIRT (exploiting ILP at run time) architecture [6]. The second category is VLIW and TTA based processors that exploit the ILP at compile time. It is known as EICT (exploiting ILP at compile time) architectures [6]. In this category the programmer or compiler finds the parallel instructions statistically before run time. Due to the flexibility and scalability behavior of VLIW architecture, it is an interesting choice for the design of ASIPs. VLIWs are constructed from multiple, concurrently operating function units (FUs) where each FU supports RISC style operations. That means, a VLIW processor does not need to include a complex instruction dependency detection hardware logic which simplifies the processor implementation. In contrast, the scalability of a traditional VLIW processor is

seriously affected by the structure of the architecture. In VLIW, the reason to limit its scalability is the complexity of the connectivity of required data path especially for register file (RF) and bypass circuit. The data bandwidth between registers and FUs depends on the number of selected FUs. Similarly, the instruction bandwidth also depends on it. However, when all FUs are utilized, the available data bandwidth is still rarely utilized. So, a new architecture is required to reduce this underutilization of RF and bypass bandwidth. The concept of this new architecture is Transport Triggered Architecture (TTA) [4]. The three step process of this transport triggering concept is 1) reducing the RF complexity, 2) reducing the bypass complexity and 3) the mirroring the programming paradigm [4].

3.1.1 Reducing the RF Complexity

Generally, in VLIWs with *K* FUs need 3*K* ports from RF. 2*K* ports required for reading and *K* ports required for writing. These 3*K* ports are utilizing in worst case situation when each FU needs to perform two reads and one write operations on the RF simultaneously. This amount of traffic can be reduced because there are some reasons that not all these 3K ports of *K* FUs are required to keep the FUs busy. The following scenarios may occur during the operations or sequence of operations [4]:

Source operands: Every operation does not require two RF source operands. For examples, register to register copies, operations with immediate operands, loads with direct, indirect or displacement (offset) addressing, jumps, and calls etc these operations only one source operand. Similarly not all operations like jumps, calls, and stores produce a result for the RF.

Bypassing: During the execution of FUs, FUs take values from RFs. But, in case of bypassing circuits are applied then bypassing values between FUs is needed when operations need operand values which are not yet available in the RF. Once the operations for which the results are not yet written back to the RFs, but some FUs are going to use of that result value then bypassing circuits bypass that value to the FUs.

For that reason when all usages of a value can be bypassed, it is not needed to write this value in the RF; in that case, the result value is said to be dead.

Operand sharing: Sometimes, an operand value may be used multiple times by the following operations. If the operand value is still in the bypass, the RF read traffic can be further reduced by operand sharing. Similarly, a RF read port is shared by multiple read operations in case of reading the same register in same cycles during the multiple operations.

Depending on the above explanation, it is possible to reduce the number of RF ports and number of RFs hence, RF complexities will be reduced. So, it is necessary to know the technique of how to control a RF with a limited number of ports. There are two control techniques for this option: 1) dynamic or run time control, and 2) static or compile time control [4].

Dynamic control: In this technique, the hardware will assign operands to port on basis of availability. In order to multiplex the available ports between the operands, hardware locks the ports for one or more cycles when there are many operands. So, the locking ports should be chosen such that locking does not contribute much to the *CPI*.

Static control: it is very difficult for hardware to determine which RF operands are to be read and written by using dynamic control. Because at compile time this information should be preciously necessary. In static control technique, a separate FU named as register unit (RU) is implemented and it has a limited number of read and write ports. Figure 18 shows the data path of a VLIW with 2 FUs and one RU. This RU has one write port and two read ports. So remarkable changes can be found between two data path of VLIW: one is using RU and another one is without RU. So in Fig. 18, BP-1 and BP-2 bypass registers and their associated bypass busses are disappeared. Because, this bypassing unit is localized within one unit.

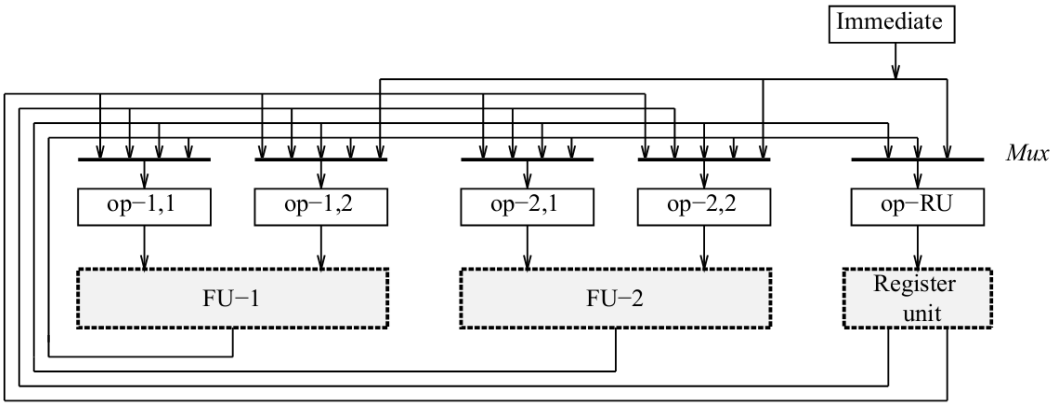


Figure 18: Data path of VLIW architecture with a separate Register Unit (RU) [4].

Here the bypassed values are saved in RU. The following example shows how such an architecture operates and programmed. Let us consider the following instructions and scheduled with the architecture explained in figure 18.

```
1. add r10,r1,#8
2. sub r3,r10,r2
3. add r6, r5, #0
4. nor r8,r3,r4
```

In third instruction, register 6 (`r6`) stores the addition result between the value of (`r5`) and 0. So it is just an activity of register copy. This kind of instruction is used for RISC processor because of its limited ALU functionality. The pipelined execution of this schedule is shown in figure 19. This schedule has three stages: reading register stage (RR), execution stage (EX), and write back stage (WB). From figure 16 it shows that the timing concept of RU is different. Register fetch, execution and write back are still pipelined.The corresponding value of `r10` is never written back or fetched from RU. That means it is used only within the bypass circuit. That's why in Fig. 19, value of `r10` is not written in the register. Similarly, the value of `r3` has to be bypassed internally within the RU. So it requires 6 cycles to execute all the four instructions. But it is possible to optimize this schedule. For example, the third instruction is

independent from other and can be scheduled in cycle two on FU-2. The fourth instruction is scheduled in cycle three such that `r3` need not to be stored in the RU. By this way, it will require less cycle to schedule these instructions. This frees a write port, which is needed in order to optimize the schedule.

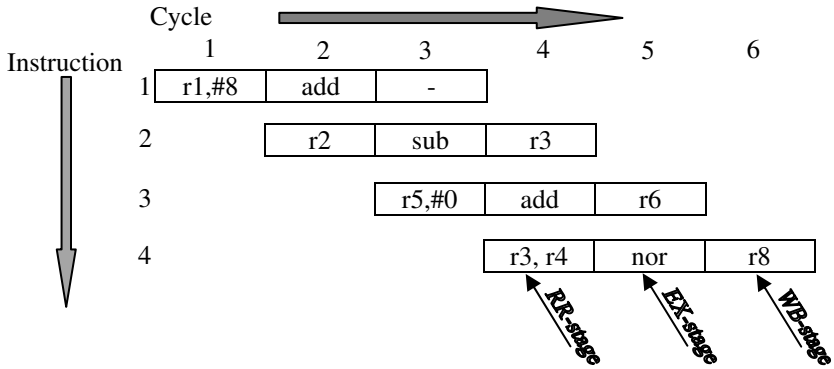


Figure 19: Pipelining diagram of four instructions [4]

Figure 20 shows the connectivity graph of VLIW architecture with separate RU. From figure 20, the data path complexity can be written as:

$$DP_{compl} = \begin{pmatrix} 3K+R+W+1, 2K+W+RN+1, 2K^2+K(2R+W+1)+WN+1, \\ N+2K+W+1, R+W \end{pmatrix} \quad (19)$$

Therefore, the extra FU adds a complexity as following equation:

$$\partial DP_{compl} / \partial K = (3, 2, 4K+2R+W+1, 2, 0) \quad (20)$$

where $RC_{max} = N$ and $WP_{max} = \max(W, K+R+1)$

If I compare this connectivity diagram with connectivity diagram diagram (without RU) then we notice a remarkable change I mean reduction in connectivity complexity. Still there is a major problem in this architecture like huge number of comparator. This is shown in following equation in terms of bypass complexity.

$$BP_{compl} = (2K+W) \times (K+R) \quad (21)$$

This can be modified by applying bypass complexity reducing technique.

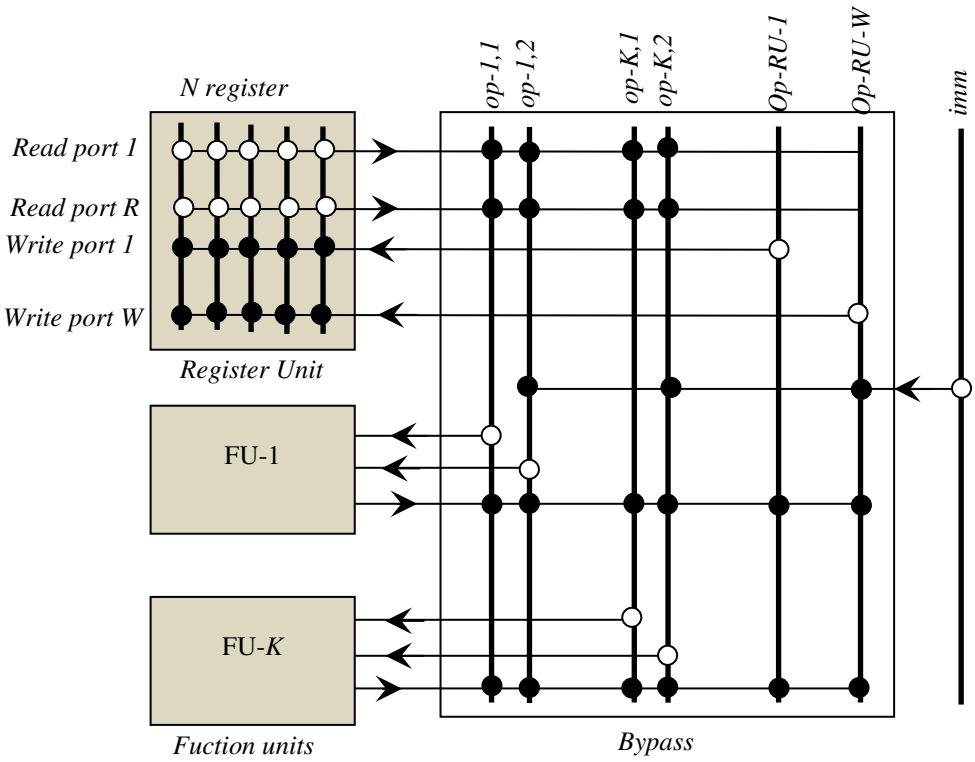


Figure 20: Connectivity diagram of VLIW processor with separate register unit (RU).

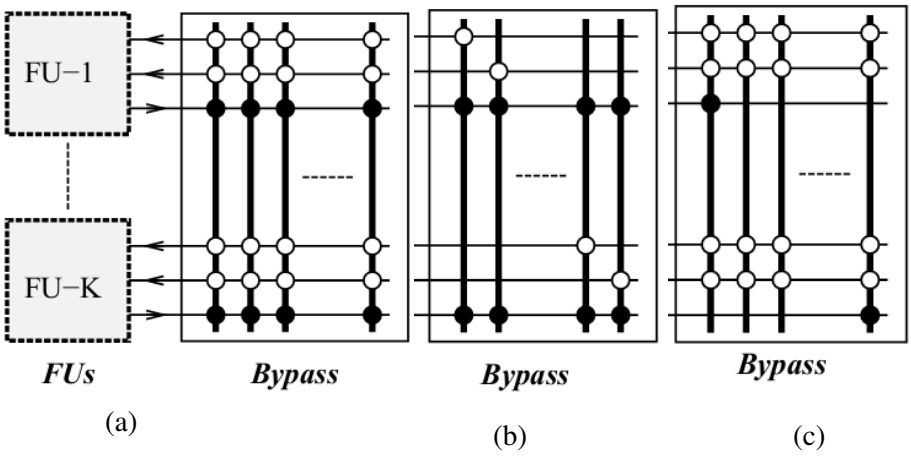


Figure 21: Connectivity status of bypass register

(a) Fully (b) Limited read and (c) Limited write connectivity [4].

3.1.2 Reducing Bypass Complexity

Bypass complexity of VLIW processor depends on the operand identifiers, number of read and write connections and number of bypass buses or global buses. In case of placing RF as a separate FU, the compiler exactly knows which input the multiplexer should be read from. Therefore, it is possible to make the bypass circuit visible at the architectural level. There are several options for this data path visibility: fully connected, limited read connectivity and limited write connectivity. Figure 21 shows these visibility connections for bypass circuit.

For fully connected network like figure 21 (a), all the read and write ports are connected with bypass network. Here, this is not an actual solution because the bypass is now considered as a shared registered file and this register file has to be bypassed as per the pipelining access by using separate stages for RR, EXE and WB. Therefore, to reduce this complexity, these connections (read and write ports of FUs in bypass network) should be limited. In order to do this, there are two possible options: one is to reduce the read connections and another one is to reduce the write connections. Figure 21 (b) shows the limited read connectivity for only one read connection per bypass register. For figure 21 (b), the bypass complexity can be written as [4]:

$$BP_{compl}(\#Bus, \#R_C, \#W_C, WP_{max}, \#Regs) = (3K, 2K, KN, 1, K, N) \quad (22)$$

where N is equivalent as the number of operand registers ($\approx 2K$). If $N = cK$, c is a constant value, then the bypass complexity per FU increment is given below:

$$\partial BP_{compl} / \partial K = (3, 2, 2cK, 0, 1, c) \quad (23)$$

From equation 23, it can be said that 3 buses are added per FU.

Limited write connectivity

Figure 21 (c) shows the limited write connectivity of the FUs for VLIW processor. Therefore, the bypass complexity can be written as [4]:

$$BP_{compl}(\#Bus, \#R_C, \#W_C, WP_{max}, \#Regs) = (3K, 2KN, N, N, 1, N) \quad (24)$$

where N is the number of bypass registers, $N \geq K$. Again $N = cK$, c is a constant value, and then the bypass complexity per FU increment is given below:

$$\partial BP_{compl} / \partial K = (3, 4cK, c, c, 0, c) \tag{25}$$

According to the equation 25, the incremental complexity is more in limited write connectivity compared to limited read connectivity. Because, this limited write connectivity leads to $2K$ non-local bypass buses for reading operands.

In VLIW architectures, still there is a problem. For example, the number of bypass buses is linearly proportional to the number of FUs. This number cannot be chosen independently. For example, suppose ALU is split into three major components: adder, shifter, and logical units. From a concurrency point of view, this splitting is very good but from a bypass complexity point of view, VLIW structure is not more attractive. It is trying to solve this problem in transport-triggered architecture (TTA).

3.2 Transport Triggered Architecture (TTA)

The connectivity diagram of bypass registers mentioned in figure 21 is not fully utilized during the execution time of FU. Because, it is necessary to design the bypass transport capacity for worst case traffic conditions. When the number of FU outputs is larger than the communication requirements, then it is required to reduce the bypass capacity. Number of FUs may increase in the following situations [4]:

FU splitting: any FU can be split upon its different functionality. For example, an ALU has different execution units: adder/subtractor, a shifter and a logical unit. These units are split to reduce the FU resource conflicts during the operation mapping. So it allow more concurrency without a large increase of hardware.

FUs with multiple outputs: sometimes FUs are generating multiple results. In this case, multiple outputs may share a single bypass bus.

Superpipelined FUs: as it is mentioned earlier, in superpipelined architecture FUs are split into different stages and it may contain multiple outputs with different latency. Like FUs with multiple outputs, it may share a single bypass bus.

As per the previous discussion, it is required to reduce the number of busses for improving bypass utilization. A FU may write single or multiple buses depending on scheduling requirements. There are two types of scheduling activities: scheduling of operation and scheduling of transport. Based upon the scheduling of register port, scheduling of transport can be done either at run time or at compile time. As discussed earlier that run time scheduling is very expensive so, it was proposed to schedule these buses at compile time [4]. Besides reducing complexity, compiler intellectually handles the transport priority in more transport than available buses situation. These transports are separated from operations. Figure 22 shows these two different views of the resulting architectures for full connectivity on bypass buses.

There are two types of views showed in figure 22. In the simple view the read and write connections are drawn as seen from the FU point of view. In this architecture, the bus connections are in cascade form that means a FU first writes its result on a local result bus the result is distributed to one of the operand register via global bypass buses. To avoid this cascade problem, the connectivity model is developed shown in figure 22 (b). For this connectivity model the the bypass complexity is given below [4]:

$$BP_{compl} = (3K + M, K(2 + M), K(1 + 2M), K, M, 2K) \tag{26}$$

M is number of bypass buses. M is a constant value, and then the bypass complexity per FU increment is given below:

$$\partial BP_{compl} / \partial K = (3, 2 + M, 1 + 2M, 1, 0, 2) \tag{27}$$

From equation 26, among the total number of buses $3K + M$ only M of them are used for global inter FU communication. For constant value of M the bypass complexity is linearly proportional to the number of FUs. In practical equation 26 may be reduced as below

$$BP_{compl} = (3K, 3K, 3K, \lceil K / M \rceil, 1, 2K) \tag{28}$$

The bypass complexity per FU increment is:

$$\partial BP_{compl} / \partial K = (3,3,3,1/M,0,2) \quad (29)$$

Therefore, this complexity is extremely low. This connectivity as well as the complexity is highly application dependent. The transport is visible at the architectural level that implies that the specification of operation can be hidden. Here the data transport can trigger the operation as a side effect of operation. So no extra instruction required for triggering.

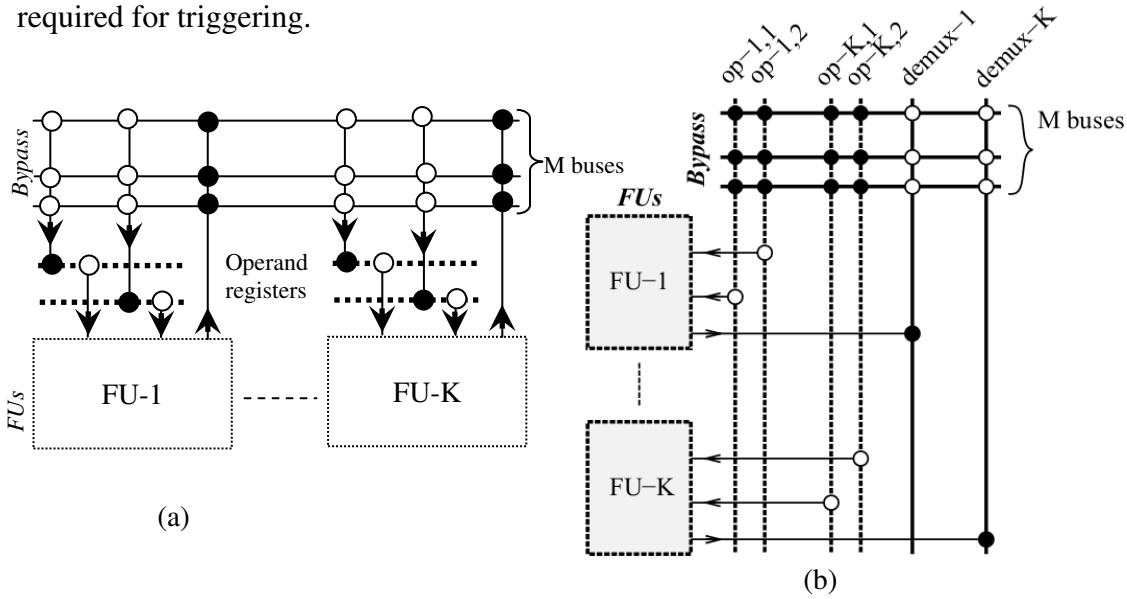


Figure 22: Architecture with visible bypass transports
(a) Simple view (b) Connectivity model [4].

According to ref [4], this newly developed architecture is known as transport triggered architecture (TTA) and the traditional architectures are known as operation triggered architectures (OTA).

TTAs are broader classification of VLIW architecture and it requires fewer constrains for scheduling data compared to VLIWs. By considering the conditions like having the same FUs, choosing the proper connectivity and selecting proper compiler schedules, TTAs become VLIW processor. Figure 23 represents the traditional VLIW architectures [4].

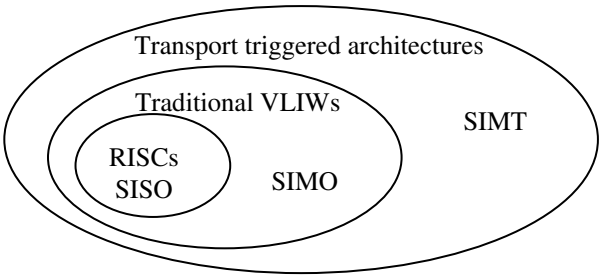


Figure 23: Architectural view for OTAs and TTAs [4].

One TTA instruction divided into several bus fields depends upon the number of buses in the architecture. Each *bus-field* specifies one *move* operation from source (*src*) to destination (*dst*). The *i*-bit indicates as source id and this source id may be interpreted as an immediate or as a register specification. The following example shows the programming in a TTA architecture. For example, TTA has three buses and this example represents the TTA scheduling of previous example.

```
   Bus-1            Bus-2            Bus-3
#8 ->O1add  ;   r1 ->O2add ;     r5 -> r6;
Radd -> O1sub;  R2 ->O2sub ;       --
Rsub -> O1nor;  R4 ->O2nor ;       --
Rnor -> r8   ;     --       ;      --
```

From this example, it requires 8 moves and four instructions to execute this operation.

3.2.1 Hardware Aspects of TTAs

The figure 24 shows an example of TTA processor. FUs, RFs, data memory, instruction memory and interconnection network are included in this architecture. An FU may contain a general purpose register file or logic units; in that case, it is named a register unit (RF) or arithmetic logic unit (ALU). Each FU is connected to the inter-connection network with one or more so-called input and output sockets. In middle, there is an interconnection network, which consists transport bus, socket and connection. Input sockets contain multiplexers which feed data from the buses into the FUs. Output sockets contain de-multiplexers; they put FU results on the buses [4]. The transport buses are used to transfer operand i.e. it executes with instructions. Here the

number of buses is customized as to reduce the cycle counts. It is the task of the compiler to optimize the required transports, given a certain connectivity, such that the cycle count (the number of executed cycles) is minimized [4]. The FUs of TTA architecture are internally pipelined and it is possible to implement one or more operations by using TTA FUs. One of the input and output ports of FU is called trigger port and as its name when an operand is transferred to this port, the operation execution is triggered. Then the result can be read from the output port after the time defined by the static latency of the operation. One of the important aspects of this TTA architecture is that FUs are may be called as register which means that the values are stored in the port until the next operation overwrites that port. Thus the traffic on the register may be reduced [7].

The register files (RFs) do not differ much from the FUs that have discussed earlier. Like FU, the RFs are connected to the IC and their connections are visible to the programmer. The TTA template also allows the customization of the register files as well as function units by the programmer and this brings a tremendous improvement of performance to the processor. The following characteristics are observed for TTA architecture, which is very interesting from the hardware design point of view:

Modularity: TTAs are constructed by using different FUs and bus connections. FUs are completely independent each other and connect with interconnection network mentioned in figure 24. Controller unit controls the FU pipeline. Under this modularity characteristic, the hardware design process is fully automated.

Flexibility and scalability: TTA architectures are very much flexible. Because the interconnection network is separated from the FUs and both can be designed independently. But for VLIW its scenario is different. If the FU changes then it is required to modify the interconnection network. The FU of TTA architecture is flexible in terms of functionality. It may contain multiple inputs and outputs including different operands.

Processor cycle time: This is very important characteristics for TTA architectures. The processor can be optimized for operation throughput instead of latency. To optimize the processor, it requires superpipelining those FUs, which constrain the achievable cycle time. Advanced bus implementation techniques are required to optimize the processor.

Hardware efficiency: Hardware efficiency of TTA processor is very high. It is very efficient to handle the hardware change aspects. TTA architecture supports one operation format and it uses reduced decoding logic among the RISC design. In TTA architecture register efficiency and transport efficiency are very high. It is not required to allocate RF stages for all the values produced during the course of a program so, in TTA it requires less number of RF. FU splitting is another aspect of TTA processor. FU logic can be split into independent parts used for different functionality. For TTA architecture, splitting FU has no impact on interconnection network and splitting FU can be used concurrently which increases the efficiency of hardware use.

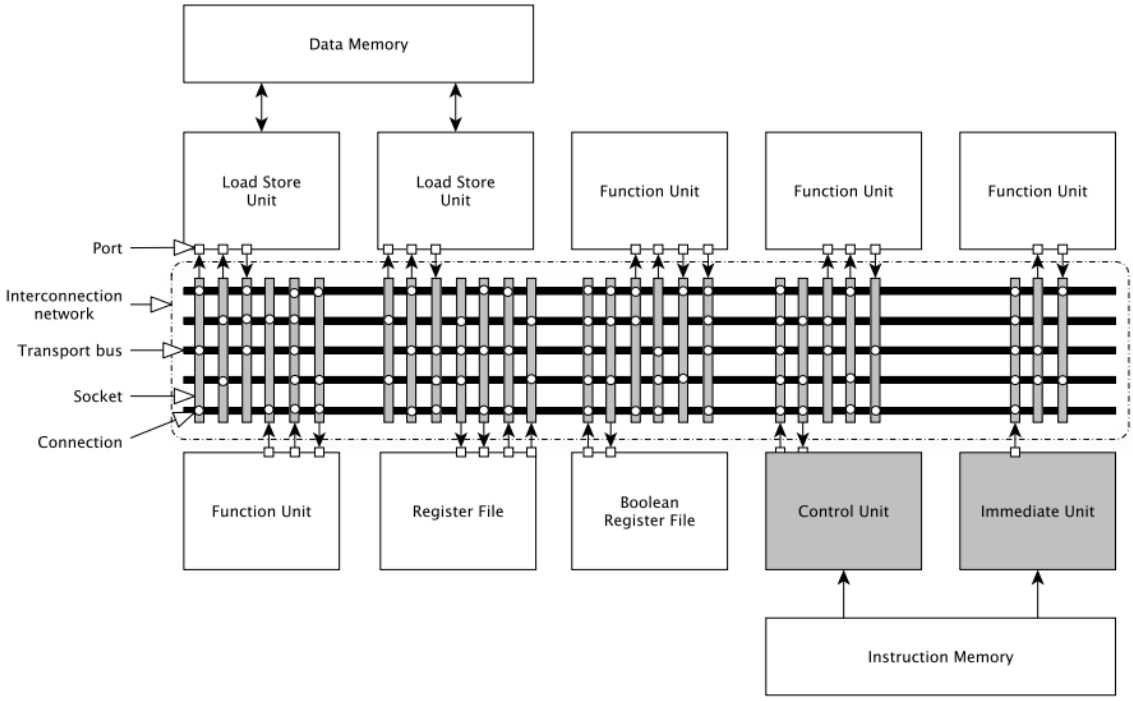


Figure 24: Example of a Transport Triggered Architecture (TTA) [3].

From the above discussion TTA architecture ensures the economy usage of hardware architecture. That means it will exchange the complexities between compiler (software) and hardware stage. Hence, this characteristics make TTAs a suitable architecture for application specific processors.

Until chapter 3, I have discussed the different processor architectures. According to the ref [4], TTA style processor is very good for implementing the application specific design. To generate the application specific processor design, I took the LTcodec system as input design. So next chapter I will discuss the basic of LTcodec theorem followed by the related works regarding the implementation of LTcodec.

Chapter 4

Luby Transform Encoder and Decoder

The binary erasure channel (BEC) is a real world channel environment which is a common communication channel model used in coding theory and information theory. Since the absence of feedback concept in forward error correction channel, advanced adaptation schemes or reliable transmission modes are infeasible in the BEC environment [8]. Therefore, research has been done to fulfill the BEC requirements. Luby *et. al*. explained a channel code with potentially limitless redundancy (rateless) and used it to solve the reliable broadcast problem in BEC [9]. This coding scheme is known as the fountain code. Luby Transform (LT) code and Raptor code are two such fountain codes based on its degree distribution function. These codes have been extensively proposed to solve the transmission problem through wired internet and the resulting behaviors are investigated on erasure channels. Like the low density parity check (LDPC), the decoding part of the LT code includes an iterative belief propagation algorithm or Log-BP algorithm. So, the decoder architecture of the LT code has followed a similar architecture to that of the LDPC decoder. In [10], the LDPC decoder was implemented by using parity check matrix directly mapped into the hardware. In [11], the VLSI architecture of LDPC was studied and authors tried to reduce the gap between decoding throughput and hardware complexity.

4.1 Coding Theory

C. E. Shannon wrote in his paper [12] that "the fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point." According to Shannon, messages are referred to or are correlated according to some system with certain physical or conceptual entities. However, the solution of this fundamental communication problem is theoretical. The

ideology of this problem is related to the transmitting or receiving the message signal. In conventional procedure, we should encode the selected message by adding some redundant information, such that even if the transmitted encoded message is corrupted by noise, there will be sufficient redundancy in it to recover the original message. Regarding this statement two individual problems should be raised: how much redundancy is required? This is related to quantitative question. Another one is what kind of redundancy is the best choice? This is related to qualitative question. These are two interesting questions. In the receiving end, the original message recovery depends on the amount of redundancy. Therefore, how many redundant bits are required for recover the transmitted messages. Alternatively, it makes sense that what is the optimum use of the communication resources at this disposal, e.g., of channel bandwidth. Each and every coding scheme assigns a value known as information rate that means what portion of that transmitted signal is useful. The qualitative solution is seeking for actual coding schemes, which should not only optimally use the communication resources, but also be equipped with the set of encoding and decoding algorithms, which can be performed practically and efficiently. For this reason, the aim of the code designer is to apply the code scheme such a way that the maximum information rate may be achieve with a vanishing probability of decoding error and efficient encoding and decoding algorithms. Shannon showed the answer regarding the quantitative question and proved that for reliable transmission, there is a certain limit to the information rate over a noisy channel. According to Shannon's theorem, for a communication channel C, the channel capacity $Cap(C) \in [0,1]$ and the information rate R are related as $R < Cap\ (C)$ for reliable communication. That means it is necessary to exist a reliable coding scheme of information rate R. Therefore, the question is still remaining which coding scheme is more reliable and close to the channel capacity. Over the last few decades, this coding theory has been developed tremendously. Researches from various fields of mathematics and engineering are doing research on it, posing and answering beautiful problems of both the theories and

the practical. Still the efficient coding scheme has been searching. Among the researchers, Wozencraft and Reiffen [13] illustrated that "*Any code of which we cannot think is good*". It was the predominant concept of early 90s. This dominant attitude should be changed after introducing the Turbo codes [13]. The IP of Turbo codes depends on using the pseudorandom interleavers in the encoding algorithm and iterative decoding algorithm. Turbo code has very structured encoding and decoding algorithm including enough randomness. After introducing the Turbo code, it was considered as the first practical codes which approached the channel capacity. Turbo codes played the vital role in the field of error correction coding. But in fact initially this code was rejected by the referees of the conference board. However today, Turbo code is an important tool of everyday technology making our lives very easier. It is employed in mobile communication, satellite communication standards, in IEEE 802.16 metropolitan wireless network standards and so on. Immediate after Turbo code, low-density parity code (LDPC) was rediscovered by many researchers independently like MacKay, Neal, Wiberg, Sipser and Spielman [13]. They showed that LDPC codes have excellent performance comparable to and often exceeding that to Turbo codes. After that, huge research efforts devoted to understand of this new new approach as an efficient error correcting code. It overcomes the problem of classical coding theory, which deals mainly with the algebraic construction of codes. As a result, nowadays practical codes and their decoding algorithms have low computational complexity and are amenable to rigorous mathematical analysis [13]. From the ref [13], the new attitude of coding theory is: "*Codes are viewed as large complex systems described by random sparse graphical models*". Therefore, decoding can be executed as the inference on the sparse graphical models. Bayesian procedure called the belief propagation algorithm is chosen as the decoding algorithm [13]. During the application of Bayesian procedure, it is realizes that Belief propagation is exceptionally efficient inference on sparse graphical models and in particular, on the sparse factor the graphical models corresponding to LDPC codes. This spare factor

graph is often called Tanner graphs [13]. Soon after the rediscovery of LDPC code, it has been realized that the iterative decoder of LDPC codes is a belief propagation decoder. Eventually, it has also been shown that decoding of Turbo codes is another representation of belief propagation algorithm [13]. Thus, belief propagation schemes changed the way of thinking of error correction coding. It seems that the best redundancy from the qualitative question of the code designer is the redundancy that can be represented by a sparse graphical model on which we can run a belief propagation algorithm.

4.2 Fundamentals of Channel Coding

Channel coding is very important for reliable data transmission and reception. When data carrying signal is propagated through channel then it is seriously affected by the response of the channel. So in the receiving end, receiver will receive this exhausted bit streams. So, successful recovery depends on the channel response. Therefore, a modeling like channel coding is mandatory for remove the effect of this unwanted noise or fading due to the channel.

4.2.1 Channel Models

To ensure the reliable transmission, channel coding is an obligatory part of communication. The main objective of channel coding is to transmit a message across a noisy channel. Here message is a sequence of k symbols $\mathbf{x} = (x_1, x_2, x_3, ..., x_k) \in X^k$, which are elements from a predetermined alphabet X. For this channel encoding purpose, the encoder maps the sequence $\mathbf{x}$ to the codeword $\mathbf{y} = (y_1, y_2, y_3, ..., y_n) \in Y^n$ and then transmitted through the channel and impaired by the channel noise.

The decoder observes a sequence of corrupted symbols, i.e. , a received word $\mathbf{z} = (z_1, z_2, z_3, ..., z_n) \in Z^n$ and estimates $\mathbf{y}$ based on $\mathbf{z}$. Vectors $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$ can realized of random variables, $\mathbf{X}$ on X^k, $\mathbf{Y}$ on Y^n, $\mathbf{Z}$ on Z^n, respectively. Similarly, each x_i, y_i, and z_i

is a realization of scalar random variables X_i, Y_i, and Z_i respectively. In addition, we assume that each X_i, Y_i, and Z_i is independent and identically distributed (*i.i.d*) according to probability density function $P_X(x)$, $P_Y(y)$, $P_Z(z)$ respectively. The relationship between **Y** and **Z** is modeled by a conditional probability density function $P_{\mathbf{Z}|\mathbf{Y}}(\mathbf{z}|\mathbf{y})$. The meaning of communication channel modeling is to specify its probability density function. Figure 25 shows three communication channels named as symmetric channel, erasure channel and Z_{cha} channel.

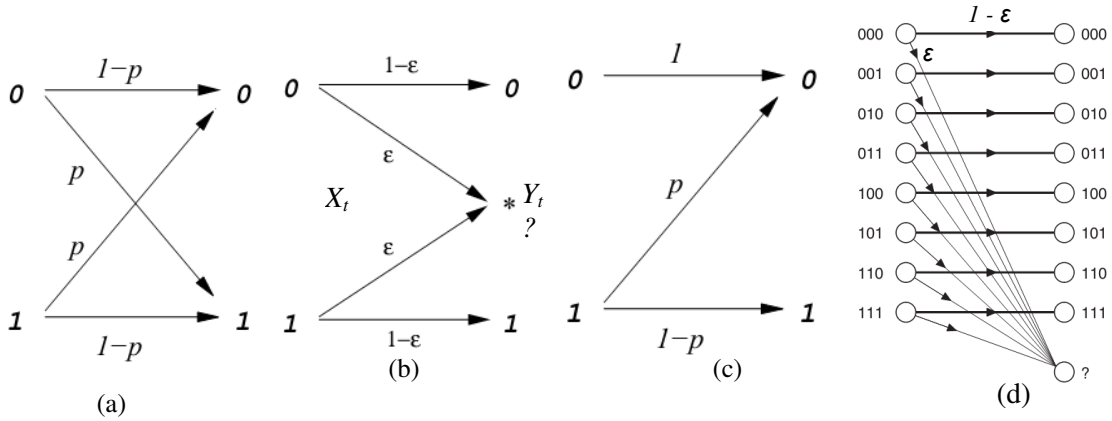


Figure 25: Three communication channels (a) memoryless symmetric (b) binary erasure (c) Z_{cha} channel (d) the 8-ary erasure channel [14].

4.2.1.1 Binary-Input, Memoryless and Symmetric (BIMS) Channels

Here we assume that the channel models are binary-input, memoryless and symmetric (BIMS channels). In memoryless case, for any input $x = (x_1, x_2, \ldots, x_N)$, the output message is a string of N letters, $y = (y_1, y_2, \ldots, y_N)$, from the alphabet $y_i \in Y$. Figure 25 (a) showed the model of BIMS channel. These channels have a binary codeword symbol alphabet Y represented either as $F_2 = \{0, 1\}$ or as set $\{-1, +1\}$. BIMS channels have no memory that means the output of such channel at any time instant depends only on its input at that time instant, i.e., $P_{\mathbf{Z}|\mathbf{Y}}(\mathbf{z} \mid \mathbf{y}) = \prod_{j=1}^{n} P_{z_j|Y_j}(z_j \mid y_j)$.

The meaning of symmetric channel is that the channel output is symmetric in its input. The maximum amount of information per symbol that can be conveyed about the codeword **Y** from the received word **Z** in the case of a memoryless channel C, is

referred to as the channel capacity [13]:

$Cap(C) = \sup_{P_Y(y)} I(Y;Z)$. Where sup is supreme function and I $(Y;Z)$ denotes denotes mutual information between the random variables Y and Z. According to Shannon theorem, for reliable transmission the value of code rate R satisfies the condition $R<Cap$ (C).

4.2.1.2 Binary Erasure Channel (BEC)

The binary erasure channel (BEC) is the simplest non-trivial channel model. It was first introduced by Elias as a toy example in 1954 [15]. Nevertheless, nowadays this is a real world problem specially in Internet promoted area. Basically, erasure channel can be used to model data networks or packet switching networks, where packets either arrive correctly or are lost due to buffer overflows or excessive delays. For example, files sent over the internet are chopped into packets, and each packet is either received without error or not received. Erasure channels model situations where information may be lost but is never corrupted. The BEC model the erasure in the simplest form like: signal bits are transmitted and either received correctly or known to be lost. It the receiving end, decoder will recovery this lost part of transmitted signal. Figure 25 (b) shows the BEC (ε). Time, indexed by t, is discrete and the transmitter and receiver are synchronized. The channel input at time t denoted by X_t, is binary $X_t \in \{0,1\}$. The corresponding output Y_t takes on values in the set {0, 1, *}, where * indicates an erasure. Each transmitted bit is either erased with probability ε, or received correctly: $Y_t \in \{X_t, *\}$ and $P\{Y_t = *\} = \varepsilon$. Each erasure is t independent because of the memoryless channel. The capacity of the BEC (ε) is C_{BEC} (ε) =1 − ε bits per channel use. Therefore, it can be shown that C_{BEC} (ε) ≤ 1 − ε. Figure 25 (d) portrays a simple channel model describing the BEC situation with q-ary erasure channel. That means, all inputs are set of input alphabet {0, 1, 2, 3,……,q-1}. The alphabet size q is 2^l, where l is the number of bits in a packet. The eight possible

inputs {0, 1,…..,7} are shown in figure 25 (d) by the binary packet 000, 001,…111. Instead of FEC technique, if the communication system is ARQ then the total number of retransmission depends on the value of ε. If the erasure probability ε is large, the number of feedback messages sent by the first protocol is very high.

4.2.1.3 Z_{cha} Channel

Z_{cha} is also known as binary asymmetric channel. This channel contains binary input and output value where the cross over 1→ 0 occurs with probability p whereas the crossover 0←1 never occurs. Figure 25 (c) represents the scenario of Z_{cha} channel. For example, X and Y are the random variables describing the probability distributions of the input and the output of the channel, respectively. So the crossovers of the channel are characterized by the conditional probabilities: $\mathrm{P}\{Y=0 \mid X=0\}=1$, $\mathrm{P}\{Y=0 \mid X=1\}=p$, $\mathrm{P}\{Y=1 \mid X=0\}=0$, and $\mathrm{P}\{Y=1 \mid X=1\}=1-p$. That means for Z_{cha} channel, a 0 is always transmitted correctly but a 1 becomes a 0 with probability p. The name of this channel comes from its graphical representation figure 25 (c).

4.3 Linear Codes

Linear codes are most common channel codes where both the message and the code word symbol alphabet restricted to F_2. A binary linear coding scheme can be viewed as a linear mapping from the set of messages F_2^k to the set of code words $C \subset \mathrm{F}_2^n$, where C forms a k dimensional vector subspace of F_2^n . Generally, this vector space C is called as code that follows particular manner of the coding scheme. It is referred as (n, k) binary linear code, where n is the length of codeword, k is the dimension of the code and R is known code rate defined as k/n.

Linear code can be fully described by its basis $\{\mathbf{g_1}, \mathbf{g_2},\ldots,\mathbf{g_k}\}$, where $\mathbf{g_i} \in \mathrm{F}_2^n$, leads to the generator matrix representation of a linier code. A $n \times k$ matrix **G** is called the generator matrix of code C if $\mathbf{c} \in C \Leftrightarrow \exists \mathrm{x} \in \mathrm{F}_2^k; \mathbf{Gx} = \mathbf{c}$

Note that any matrix with columns that form a basis of C is a generator matrix of C and that representation by generator matrix allows a simple mechanism of mapping the messages to the code words. On the other hand C can be specified as its dual (orthogonal) sub space $C^{\perp}$ within F_2^n and its basis $\{\mathbf{h}_1, \mathbf{h}_2, \ldots\ldots, \mathbf{h}_{n-k}\}$. The dual subspace of C is defined as $C^{\perp} = \{\mathbf{c}' \in \mathrm{F}_2^n : \mathbf{c}' \cdot c = 0 \forall \mathbf{c} \in C\}$.

By this way, it is possible to represent a parity check matrix of a linear code. An $(n-k)\times n$ matrix $\mathbf{H}$ is the parity check matrix of C if $\mathbf{c} \in C \Leftrightarrow \mathbf{Hc} = 0$. Therefore, it can be written that any matrix with rows that form a basis of $C^{\perp}$ is a parity check matrix of C.

In fountain codes, coding schemes have no fixed rate. Each row of the generator matrix of such coding scheme can be viewed as a random variable on F_2^k, where k is the dimension of the code. At any time instant $j \in \mathrm{N}$, the fountain encoder generates a single encoded symbol $y_j = \mathbf{v}_j \cdot \mathbf{x}$ from the message $\mathbf{x} \in \mathrm{F}_2^k$ where $\mathbf{x}$ is a randomly chosen row vector from F_2^k. In this scheme the receiver observes a number of received word symbols $z_{i_1}, z_{i_2}, \ldots\ldots, z_{i_n}$ corresponding to the transmitted symbols $y_{i_1}, \mathrm{y}_{i_2}, \ldots\ldots, y_{i_n}$. The resulting code at the receiver end is an (n, k) binary linear code described by a generator matrix with vectors $v_{i_1}, \mathrm{v}_{i_2}, \ldots\ldots, v_{i_n}$ as its rows. If the decoder fails to decode then receiver will collect additional encoded symbols which result in a code of greater length.

4.4 Belief Propagation Decoding Algorithm

Like many other algorithms, decoding of linear codes deals with the optimization of a rather complicated global function of a large number of variables. For this reason, decoding is difficult compared to encoding procedure for this linear code. One important aspect of this decoding is the factorization of global product into a local functions i.e., functions defined on small subsets of the set of all variables. Then it is

possible to get a starting point in the construction of the efficient algorithm. This factorization is usually visualized with a bipartite graph, called factor graph. The factor graph is used to represent relations between local functions and variables. It describes which variables are arguments of which local functions. It can be said that a factor graph is a graphical model on which Bayesian inference can be performed and in particular the Belief Propagation (BP) algorithm [13]. In order to optimize Belief propagation algorithm simply exploits the factorization of the global function to efficiently compute the global function many times. This is on the same conceptual level as the distributive law computations. For example, a function of three variables can be formed as two ways: $f(a,b,c) = ab + ac$ and $f(a,b,c) = a(b+c)$ .Therefore it is clearly more efficient to compute the factorized version of the function (second function) compared to the first one. In first function, it requires two multiplications and one addition whereas in second form of function it requires one addition and one multiplication.

The complete explanation of BP algorithm will be discussed in Appendix II. It is not only used in iterative decoding procedures for sparse matrix codes but also used in BCJR, Viterbi, Kalman filtering and certain instances of the fast Fourier transformation. Now I will discuss how BP algorithm relates to the decoding problem of binary linear codes.

4.4.1 Binary-input MAP Decoding via Belief Propagation

Let us assume that binary codewords of length n are transmitted through a binary input memoryless symmetric channel. Consider that the codeword $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathrm{F}_2^n$, is generated by an (n, k) linear code C described by its parity check matrix $\mathbf{H} = (h_i^j) \in \mathrm{F}_2^{(n-k)\times n}$. Note that the received word is $\mathbf{y} = (y_1, y_2, \ldots, y_n)$. Assume that the channel is described by its transition probability $\mathrm{P}_{\mathbf{Y}|\mathbf{X}}(\mathbf{y} \mid \mathbf{x}) = \prod_{j=1}^{n} P_{Y_j|X_j}(y_j \mid x_j)$. *Maximum a posteriori* (MAP) decoding problem can be described as the optimization

problem:

$$\hat{x}_i^{MAP} = \arg\max_{x_i \in \{0,1\}} \mathrm{P}_{X_i|\mathbf{Y}}(x_i \mid \mathbf{y}), i \in N_n . \tag{30}$$

The previous can be transformed as follows

$$\hat{x}_i^{MAP} = \arg\max_{x_i \in \{0,1\}} \sum_{\sim x_i} \mathrm{P}_{\mathbf{X}|\mathbf{Y}}(\mathbf{x} \mid \mathbf{y}) \tag{31}$$

Equation 31 is written from law of total probability and by applying the Bayes's law

$$\hat{x}_i^{MAP} = \arg\max_{x_i \in \{0,1\}} \sum_{\sim x_i} \mathrm{P}_{\mathbf{Y}|\mathbf{X}}(\mathbf{y} \mid \mathbf{x}) \mathrm{P}_{\mathbf{X}}(\mathbf{x}) = \arg\max_{x_i \in \{0,1\}} \sum_{\sim x_i} \left(\prod_{j=1}^{n} \mathrm{P}_{Y_j|X_j}(y_j \mid x_j) \right) \mathrm{X}_{\{\mathrm{x} \in C\}} \tag{32}$$

where $\mathrm{X}_{\{\cdot\}}$ is the indicator function. In the last step, we have used the fact that the channel is memoryless and that codewords have uniform prior. We write $\sum_{\sim x_i}$ to indicate a summation over all components of x (except x_i) and not the components of y. Assume that the code indicator function $\mathrm{X}_{\{\cdot\}}$ has a factorized form. From equation 32 it is then clear that the bit-wise decoding problem is equivalent to calculating the marginal of a factorized function and choosing the value that maximizes this marginal.

Example: Consider the binary linear code $C(H)$ defined by the parity check matrix

$$H = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

In this case $\arg\max_{x_i \in \{0,1\}} \mathrm{P}_{X_i|\mathbf{Y}}(x_i \mid \mathbf{y})$ can be factorized as

$$\arg\max_{x_i \in \{0,1\}} \sum_{\sim x_i} \left(\prod_{j=1}^{7} \mathrm{P}_{Y_j|X_j}(y_j \mid x_j) \right) \mathrm{X}_{\{x_1+x_2+x_4=0\}} \mathrm{X}_{\{x_3+x_4+x_6=0\}} \mathrm{X}_{\{x_4+x_5+x_7=0\}} .$$

The corresponding factor graph is shown in the figure 26. This graph includes the Tanner graph of H but additionally contains the factor nodes, which represent the effect of the channel.

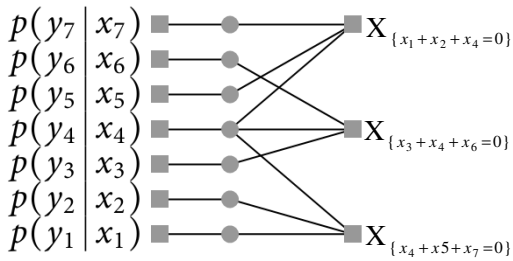


Figure 26: Factor graph for the MAP decoding [15]

For this particular case, the resulting graph is a tree. We can therefore apply the message-passing algorithm to this example to perform bitwise MAP decoding.

Therefore, MAP decoding consists of the marginalization of the function

$$f(x_1,....., x_n; y_1,....., y_n) = \left(\prod_{j=1}^{n} \mathrm{P}_{Y_j|X_j}(y_j \mid x_j)\right)\left(\prod_{j=1}^{n-k} X\{\mathrm{h}_j \cdot \mathrm{x} = 0\}\right)$$

over each variable $x_i, i \in N_n$, where $\mathbf{h}_j, j \in N_{n-k}$, denotes the *j*-th row of the parity check matrix **H**. This marginalization can be performed by a belief propagation algorithm on a factor graph corresponding to the parity check matrix **H**. This is shown in the previous example.

4.4.2 Message-Passing Rules for Bit-wise MAP Decoding

In binary message domain *u(x)* is denoted as message signal and can be thought of as a real valued vector of length 2, ($u(1)$, $u(0)$) (here we think of the bit values as {0,1}). The initial such message sent from the factor leaf node representing the *i*-th channel realization to the variable node *i* is ($p_{Y_i|X_i}(y_i|1)$, $p_{Y_i|X_i}(y_i|0)$) as mentioned in figure 26. A variable node of degree $K + 1$ showed in figure 27 the message passing rule calls for a pointwise multiplication [15]:

$$\mu(1) = \prod_{k=1}^{K} \mu_k(1) \quad , \quad \mu(0) = \prod_{k=1}^{K} \mu_k(0) . \tag{33}$$

Now take the ratio $r_k = \mu_k(1) / \mu_k(0)$. Now putting the relationship from equation 33,

we have

$$r = \frac{\mu(1)}{\mu(0)} = \frac{\prod_{k=1}^{K} \mu_k(1)}{\prod_{k=1}^{K} \mu_k(0)} = \prod_{k=1}^{K} r_k \tag{32}$$

That means that the ratio of the outgoing message at a variable node is the product of the incoming ratios. Again if take the log-likelihood ratios $l_k = \ln(r_k)$, then processing rule is $l = \sum_{k=1}^{K} l_k$. Therefore, 'r' and 'l' can be denoted as likelihood and log-likelihood ratios.

Consider now the ratio of an outgoing message at a check node, which has degree J + 1 showed in figure 27.

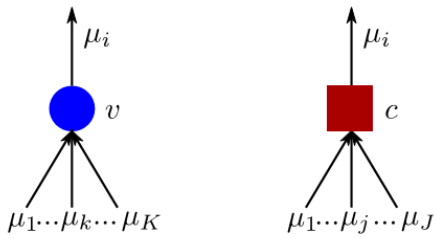


Figure 27 : A variable node (v) with K + 1 neighbors and a check node (c) with J + 1 neighbors.

For a check node it can be written that $f(x_1, \ldots\ldots, x_n) = \mathrm{X}_{\{\prod_{j=1}^{J} x_j = \mathrm{x}\}}$. We assume that the x_i takes values in {0, 1} and instead of $\sum_{j=1}^{J} x_j = x$ it can be written as $\prod_{j=1}^{J} x_j = x$.

Therefore,

$$r = \frac{\mu(1)}{\mu(0)} = \frac{\sum_{\sim x} f(1, x_1, \ldots, x_J) \prod_{j=1}^{J} \mu_j(x_j)}{\sum_{\sim x} f(0, x_1, \ldots, x_J) \prod_{j=1}^{J} \mu_j(x_j)} = \frac{\sum_{x_1, \ldots x_J : \prod_{j=1}^{J} x_j = 1} \prod_{j=1}^{J} \mu_j(x_j)}{\sum_{x_1, \ldots x_J : \prod_{j=1}^{J} x_j = 0} \prod_{j=1}^{J} \mu_j(x_j)} = \frac{\sum_{x_1, \ldots x_J : \prod_{j=1}^{J} x_j = 1} \prod_{j=1}^{J} \frac{\mu_j(x_j)}{\mu_j(0)}}{\sum_{x_1, \ldots x_J : \prod_{j=1}^{J} x_j = 0} \prod_{j=1}^{J} \frac{\mu_j(x_j)}{\mu_j(0)}}$$

$$= \frac{\sum_{x_1,\ldots x_J:\prod_{j=1}^{J} x_j=1} \prod_{j=1}^{J} r_j^{(1+x_j)/2}}{\sum_{x_1,\ldots x_J:\prod_{j=1}^{J} x_j=0} \prod_{j=1}^{J} r_j^{(1+x_j)/2}} = \frac{\prod_{j=1}^{J}(r_j+1)+\prod_{j=1}^{J}(r_j-1)}{\prod_{j=1}^{J}(r_j+1)-\prod_{j=1}^{J}(r_j-1)} \tag{33}$$

In the equation 33, it has a term like $\prod_{j=1}^{J}(r_j+1)$, we get the sum of all products of the individual terms r_j, j= 1,….,J.

For example $\prod_{j=1}^{3}(r_j+1)=1+r_1+r_2+r_3+r_1r_2+r_2r_3+r_3r_1+r_1r_2r_3$. Similar fashion can also be applied for $\prod_{j=1}^{J}(r_j-1)$. For this reason equation 33 is developed by using the following relationship

$$\prod_{j=1}^{J}(r_j+1)+\prod_{j=1}^{J}(r_j-1)=2\sum_{x_1,\ldots x_J:\prod_{j=1}^{J} x_j=1} \prod_{j=1}^{J} r_j^{(1+x_j)/2}.$$

Now, in equation 33, divide the numerator and denominator by $\prod_{j=1}^{J}(r_j+1)$, it can be written as

$$r = \frac{1+\prod_j \frac{r_j-1}{r_j+1}}{1-\prod_j \frac{r_j-1}{r_j+1}}.$$

So, $\frac{r-1}{r+1}=\prod_j \frac{r_j-1}{r_j+1}$. Now if $r=e^l$ then we see that $\frac{r-1}{r+1}=\tanh(l/2)$. Combining these two statements, it can be written as

$$\tanh(l/2) = \frac{r-1}{r+1} = \prod_{j=1}^{J} \frac{r_j - 1}{r_j + 1} = \prod_{j=1}^{J} \tanh(l_j / 2)$$
$$\therefore l = 2\tanh^{-1}\left(\prod_{j=1}^{J} \tanh(l_j / 2)\right) \tag{34}$$

For the case of binary input memoryless channels, we discussed the message passing rules for bit-wise MAP decoding of a parity check code and saw that if the factor graph of a code is a tree, the sum-product solution is equal to the MAP decoding solution. The message passing algorithm can efficiently perform MAP decoding for the codes whose corresponding factor graph is a tree. But the class of code that has a tree like factor graph is not powerful enough to perform well using this message passing algorithm. Because it may contain low weight codewords and has a large probability of error.

Two fundamentals rules are derived by these equations. Equation 33 represented as 'sum' rule and equation 34 portrays 'tanh' rule. These rules are important for the belief propagation algorithm in decoding of binary linear codes over BIMS channels. In another sense, these rules are the base for decoding of LDPC, LDGM, LT and any fountain codes.

4.5 Fountain Codes

It is very easy to imagine that users are receiving data from satellite during the car driving or the vehicles are receiving navigation updates data from the satellite. In this case, the packet may be lost for many reasons like car is in deep signal fading tunnel, or the channel erasure property is too high for signal degradation. Therefore, whatever the reasons, there are packets lost in this communication. If the communication system is ARQ then the system throughput degenerates as the number of receivers become large. Indeed, if each of the hundreds of thousands of receivers drops only a small fraction of packets and requests their retransmission, chances are that every packet must be retransmitted, and that the broadcaster will need to repeat the entire

transmission several times. As it is mentioned earlier that the above channel is known as binary erasure channel, let us assume that the transmitter needs to communicate a certain message of k packets to a large number of receivers. Each receiver $j \in N_r$, where r is the number of receivers correctly receives a certain fraction $\left(1 - p_e^{(j)}\right)$ of all transmitted signal. Therefore, $p_e^{(j)}$ is the instantaneous packet loss rate observed by the jth receiver. In order to avoid feedback request, it requires some form of channel coding mechanism applicable for erasure channels. The classic block codes for erasure correction are called Reed–Solomon codes [16]. An (N, K) Reed–Solomon code (over an alphabet of size $q = 2^l$) has the ideal property that if any K of the N transmitted symbols are received then the original K source symbols can be recovered (Reed–Solomon codes exist for $N < q$). Practically RS code is applicable for small value of K, N and q. In RS coding, standard implementations of encoding and decoding consume the cost order of $K(N - K)\log_2 N$ packet operations. Moreover, like other block code, in RS code it is required to know the value of code rate R and erasure probability p before transmission. If p is larger than the expected value then the receiver will receive fewer than K. Therefore, another encoding technique pioneered by Michael Luby [17] is required to overcome this problem. Soon, fountain codes [13] would be born.

4.5.1 Properties of Fountain Codes

In order to avoid the necessity to modify the encoding scheme whenever conditions in a loss prone network change, the idea of a digital fountain arose rather naturally. The digital fountain encoder should be able to produce an endless supply of encoded packets per message and these packets are then just sprayed across the network, finally each receiver simply keeps on collecting them until their number reaches some threshold larger than message length. They can then attempt the reconstruction of the original message, and a judicious choice of encoding scheme should be the one that provides high probability of successful reconstruction when received bit are only

marginally larger than message bit. In such schemes, no feedback is ever required. The encoder of a fountain code is a metaphorical fountainthat produces an endless supply of water drops that means encoded packets. Suppose the original file has a size of Kl bits where K is the number of packets and each drop contains l encoded bits. Now any receiver wishes to receive the encoded file then it will hold the bucket under the fountain and collects the the number of drops the bucket is a little larger than K so that it can recover the original message. In fountain code, the number of encoded bit generated from the source message is potentially limitless. For this reason, it is known as rateless code. In fact, it simultaneously supports both extremes of packet loss rates, since the users with low packet loss can collect their packets very quickly and tune out of the broadcast. Furthermore, it assumes that each produced encoded packet is equally useful to the receiver. The size of the encoded packet is determined on the fly that means depending on its erasure characteristics, every receiver will receive different size of packets. Fountain code is near optimal for every erasure channel. Regardless of the statistics of the erasure scenario of the channel, encoder will generate packets as are needed to recover the source data. The source data should be recovered from K' encoded data where K' is slightly larger than K. Moreover, the fountain code has very less encoding and decoding complexities. A digital fountain that transmits the encoded packet should have the following properties:

- It can generate an endless supply of encoding packets with constant encoding cost per packet in terms of time or arithmetic operations.
- A user can reconstruct the message using any K packets with constant decoding cost per packet, meaning the decoding is linear in K.
- The space needed to store any data during encoding and decoding is linear in K.

These properties show digital fountains are as reliable and efficient as TCP systems, but also universal and tolerant, properties desired in networks.

4.5.2 The Random Linear Fountain

Based on the above properties, we can identify the fountain-coding scheme for an arbitrary channel model with a probabilistic process that assigns to the message an infinite sequence of encoded symbols, all of which are the evaluations of an independently selected function of the message. Assume that an encoder has a file of size K packets $s_1, s_2, s_3, \ldots, s_K$. Here, the concept of packet is an elementary unit that is either transmitted intact or erased by the erasure channel. Let us assume that in n clock cycle the encoder generates K random bits $\{G_{kn}\}$ and the transmitted packet t_n is set to the bitwise sum, modulo 2 of the source packets for which G_{kn} is 1 [18].

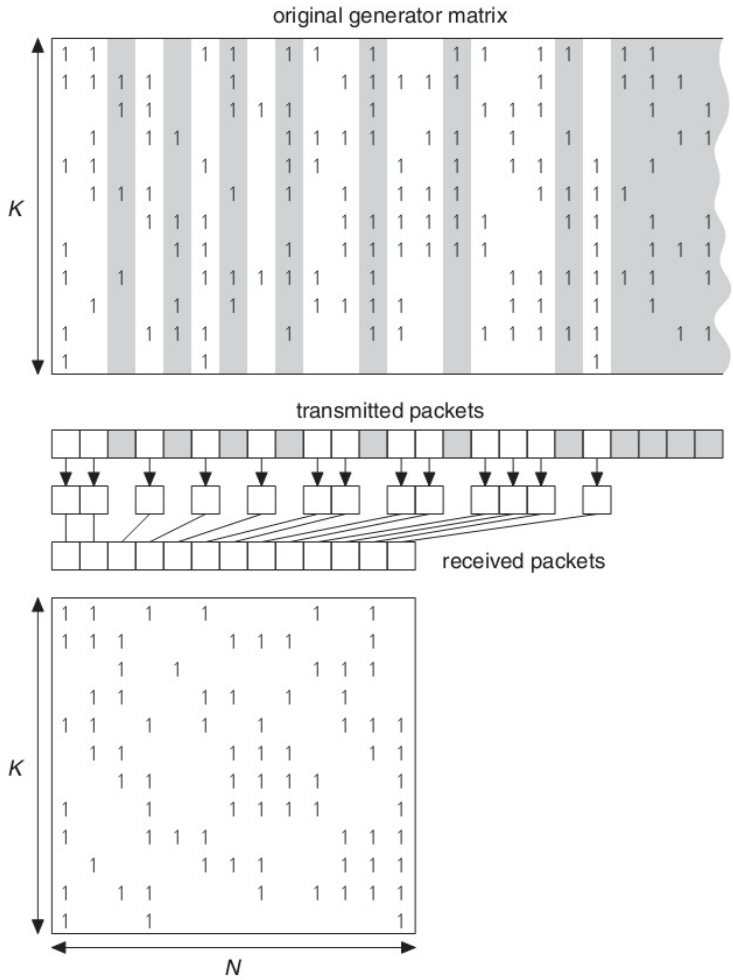


Figure 28: Transmission scenario of binary fountain code over BEC [18].

$$t_n = \sum_{k=1}^{K} s_k G_{kn} \tag{34}$$

Figure 28 shows the transmission scenario through BEC using a generator matrix of a random linear code.

The gray lines show the lost packet sprayed from the transmitter and the bottom part of figure 28 shows the received packet be the receiver. Therefore, in the receiving matrix the lost packet columns are missing. Point should be noted that the top part of the figure 28 represents the original generator matrix but due to the erasure property of channel in receiving end receiver knows the fragment of the generator matrix ***G*** associated with its packets. Let we assume that after erasure, receiver collects *N* packets from the transmitted signal. Therefore, the dimension of ***G*** is *K* – by–*N* matrix. Now the question should be posted that, what is the chance that the receiver will recover the entire source file without error?

For example if *N*<*K*, the receiver has not enough information to decode the transmitted file. If *N* = *K* the receiver will receive *K* – by–*K* matrix ***G*** and can be able to decode by using the following rule:

$$s_k = \sum_{n=1}^{N} t_n G_{nk}^{-1} \tag{35}$$

where $\boldsymbol{G}^{-1}$ is the inverse of matrix ***G*** and is computed by Gaussian elimination. Let we calculate the probability of a random *K*-by-*K* binary matrix is invertible. It is the product of *K* probabilities, each of them the probability that a new column of G is linearly independent of the preceding columns. We need to ensure that ***G*** has *K* non-zero columns. The probability that the first column of ***G*** has non-zero value is $(1\text{-}2^{-K})$. Similarly the probability is $(1\text{-}2^{-(K-1)})$ that the second column is equal neither to the all-zero column nor to the first column of ***G***. So the probability of invertibility is $\left(1-\frac{1}{2^K}\right)\left(1-\frac{1}{2^{(K-1)}}\right)\times\cdots\times\left(1-\frac{1}{8}\right)\left(1-\frac{1}{4}\right)\left(1-\frac{1}{2}\right)=0.289$ for *K* larger than 10. So, this invertible probability is very less and this expected value is close to one.

Now let E a small number means excess of packets and at receiving end receiver will receive this excess packets in addition with K then $N = K + E$. Therefore at receiving end the dimension of $\boldsymbol{G}$ is K – by – N. So, what is the probability that the $\boldsymbol{G}$ matrix contains an invertible K – by – K matrix? Let we assume the probability δ that means the receiver will not be able to decode the file when E excess packets have been received. Hence, δ is failure probability and $1 - \delta$ is the probability that $\boldsymbol{G}$ matrix contains an invertible K – by – K matrix. Figure 29 shows the plotting of failure probability δ with respect to E for K = 100 [18]. δ is bounded by $\delta(E) \leq 2^{-E}$ for any value of K.

In nutshell, for reliable communication, receiver has to receiver $K + \log_2 1/\delta$ encoded bit at $1 - \delta$ probability condition. As excess packets E increase then the probability of success also increases to $(1- \delta)$, where $\delta = 2^{-E}$.

The above scenario can be portrays by the following example. We hypothetically think that we throw N balls independently at random into K bins, where K is very large like 1000 or 10,000. There are several questions: if $N = K$ then what fraction of bins is empty? If $N > 3K$, is there any empty bin? Or minimum how many balls are required to ensure all bins will get at least one ball?

After throwing N balls, then the probability that one particular bin is empty is $\left(1-\frac{1}{K}\right)^N \approx e^{-N/K}$. Now if $N = K$ and $N = 3K$ then the probability of one particular bin is empty is approximately $1/e$ and $1/e^3$ respectively. To make sure that all the bins have a ball, we need to throw many balls. For general value of N the expected number of empty bins is $Ke^{-N/K}$. So this expected number is almost equal to δ. Therefore $\delta = Ke^{-N/K}$, $N = K \log_e \frac{K}{\delta}$ and $N > K \log_e \frac{K}{\delta}$. This condition represents that if N satisfies this relation then each bin will get at least one ball after throwing.

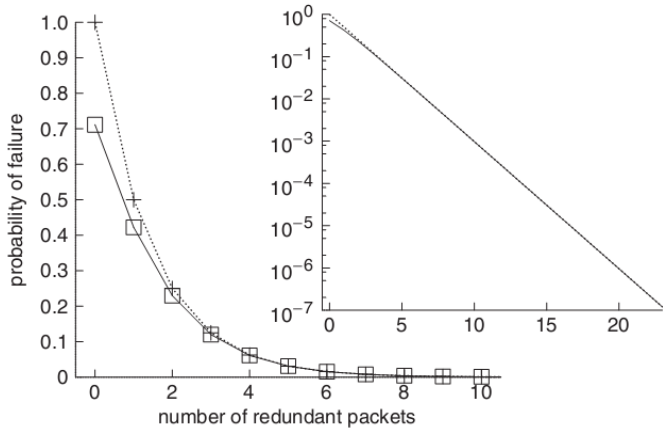


Figure 29: Properties of failure probability δ against E the number of redundant packets [18].

4.6 Luby Transform Codes

LT (Luby Transform) codes [17] are the first class of fountain codes fully realizing the digital fountain paradigm. LT codes are binary linear fountain rateless codes. The encoder can generate as many encoding symbols as required to decode k information symbols. The encoding and decoding algorithms of LT codes are simple; they are similar to parity-check processes. LT codes are efficient in the sense that the transmitter does not require an acknowledgement (ACK) from the receiver. This property is especially desired in multicast channels because it will significantly decrease the overhead incurred by processing the ACKs from multiple receivers [14]. It has two parameters: the length of the message and degree distribution on the set of message alphabet. The output degree distribution of an LT code will be identified with its generating polynomial. The analysis of LT codes is based on the decoding algorithm and degree distribution properties. For this reason, Ideal Soliton distribution and Robust Soliton distribution are introduced as the degree distribution. The importance of having a good degree distribution of encoding symbols is also investigated under this analysis. LT codes are considered as very efficient if K information symbols can be recovered from any $K + O(\sqrt{K}\ln^2(K/\delta))$ encoding symbols with probability 1 - δ using $O(K \cdot \ln(K/\delta))$ operations [14].

4.6.1 Encoding Process

Any number of encoding symbols t_n can be independently generated from source file $\{s_1,s_2,s_3,\ldots,s_K\}$ information symbols by the following encoding process:

- Determine the degree d_n of the packet from a degree distribution. This degree is chosen at random from a given node degree distribution $\rho(d)$. The appropriate choice of ρ depends on the source file size K.

Choose d_n for distinct input packets and set t_n equal to the bitwise sum, modulo 2, of those d_n packets.

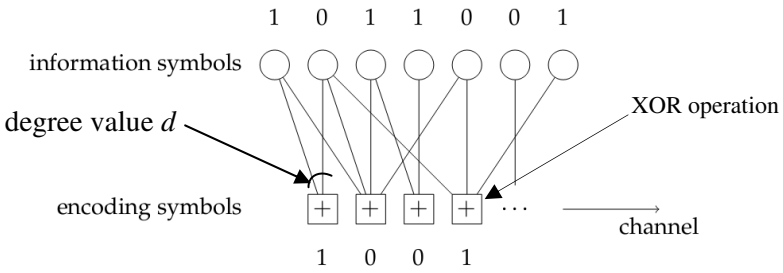


Figure 30: Encoding process of LT codes.

This process is similar to the generating parity bits except that only the parity bits are transmitted. As shown in figure 30, the degree distribution $\rho(d)$ comes from the sense that the bipartite graph (shown in figure 30) consists of information symbols as variable nodes and encoding node as factor node. The degree value d determines the performance of the LT coding so that it will successfully decode the encoded signal with lower complexity. The algorithm of LT encoder can be described as the following way:

LT encoding algorithm

Input: message $\mathbf{x} = \{s_1,s_2,s_3,\ldots,s_K\}$, probability distribution $\rho(d)$ on N_K

Output: an encoded symbol t_n

1. Sample an output degree d with probability $\rho(d)$
2. Sample d distinct message symbols $\{s_{i_1}, s_{i_2},\ldots, s_{i_d}\}$ uniformly at random

from the message $\{s_1, s_2, s_3, \ldots, s_K\}$ and XOR them, $t_n = \oplus_{j=1}^{d} s_{i_j}$.

LT codes hold two major benefits compared to the general binary linear fountain codes. Firstly, the code design is greatly simplified and the code designer needs only to specify the set of d numbers describing the degree distribution $\rho(d)$. Secondly, it is possible to select the output degree distribution in such a way that the decoding of an LT code is possible with a version of a computationally efficient belief propagation algorithm.

4.6.2 Decoding Process

The decoding of an LT code utilizes a belief propagation (BP) algorithm on the factor graph of the linear encoder $\mathrm{F}_2^K \rightarrow \mathrm{F}_2^N$ obtained by the fountain encoder map. This factor graph has the incidence matrix formed by N active rows of the LT generator matrix, which correspond to N observed encoded symbols. Decoding of LT code is easy in the case of an erasure channel. Therefore the decoder's task is to recover **s** from **t** = **sG**, where **G** is the generator matrix associated with the graph. The decoding is done by using the message passing algorithm like sum-product algorithm. In receiving end, all messages are either completely uncertain (message packet s_k could have any value with equal probability) or completely certain (s_k has a particular value with probability one). We assume that in the check node position encoder generates t_n encoded signal. The simple decoding process is illustrated by the following way:

1. Find a check node t_n that is connected to only one source packet s_k (if there is no such condition decoding halts).

   (a) Set $s_k = t_n$.

   (b) Do XOR s_k to all checks $t_{n'}$ that are connected to s_k:

   $$t_{n'} = t_{n'} \oplus s_k \text{ for all } n' \text{ such that } G_{n'k} = 1 .$$

   Remove all the edges connected to the source packet s_k.

2. Repeat (1) until all s_k are determined.

The above process is illustrated in figure 31 for a case where each packet is just one bit. There are three source packets (shown by the upper circles) and four received packets (shown by the lower check symbols), which have the values t_1, t_2, t_3, t_4 = 1011 at the start of the application.

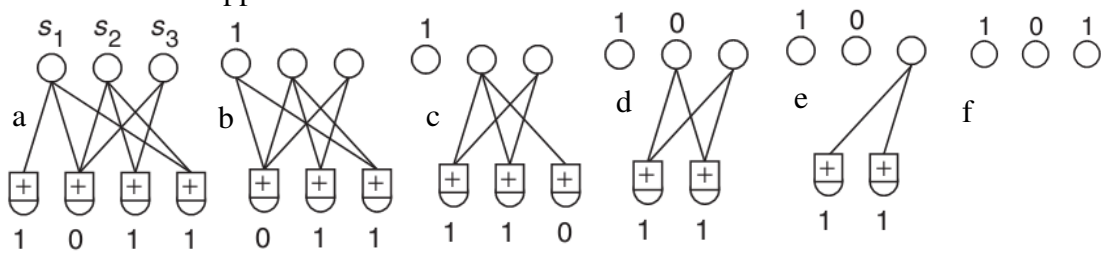


Figure 31: Example of decoding LT code for K = 3 and N = 4 [18].

In figure 31, panel 'a' shows the first iteration where the only check node is connected to a sole source bit (variable node). Then in panel 'b' we set source bit s_1 accordingly to check node bit (here s_1 = 1) then XOR the value of s_1 (1) to the check nodes to which it is connected to s_1 (panel 'c') and finally disconnecting s_1 with its edges from the graph. Thus, first iteration is completed. Similarly, at the starting of the second iteration shown in panel 'c', the fourth check node is connected to a sole source bit, s_2. Then we set s_2 to t_4 as shown in panel'd'. Finally, in third iteration, two check nodes are both connected to s_3 and they agree about the value of s_3, which is restored in panel 'f'.

From the above explanation, the decoding process is bounded into three steps: release, cover and process. In release step all encoding symbols of degree one (those which are connected to one information symbol t_1 in panel 'a' of figure 31) are released to cover their unique neighbor.

In cover step, the released encoding symbols cover their unique neighbor information symbols. In this step, the covered but not processed input symbols are sent to *ripple*, which is a set of covered unprocessed information symbols gathered through the previous iterations. That is shown in figure 31 panel 'b'.

In process step, one information symbol in the ripple is chosen to be processed. In this

step, the edges connecting the information symbol to its neighbor encoding symbols are removed and the value of each encoding symbol changes according to the information symbol. The processed information symbol is removed from the ripple. This procedure is shown in figure 31 panel 'c'. So, these working procedures explained in figure 31 had been translated into HLL for example C in this thesis work. Therefore, for implementation point of view, I make a structure of this algorithm efficiently. Otherwise, it will take more cycle for simulation using ASIP design tools. These HLL codes are transformed into the assembly instructions by the compiler of specific tool. For example in TTA, TCE compiler translates the input design into the TTA assembly code and this will be discussed in chapter 5. The following algorithm represents the LT decoding algorithm for BEC.

LT decoding algorithm

Input: channel output $t_n \in Z^N$, factor graph G_{LT} representing the active N rows in the LT generator matrix.

Output: message **x**{ $s_1, s_2, s_3, \ldots, s_K$ } $\in X^K$ (or an indicator 0 that the decoding has failed)

1. Assign an all-erasure vector **x** to variable nodes, $s_i = *, i \in N_K$
2. **while x** at least one erased sample s_j = * **do**

   find an unerased output node a, $t_a \neq *$, connected to exactly one erased variable node i, s_i = *.

   ifthere is no such output node return 0 (*decoding fails*)

   else

   sets$s_i = t_a$, t_a = *;

   set $t_b = s_i \oplus t_b, \forall b \in N(i)$;

   end if
3. **end while**
4. **return x**

The decoding process continues by the iterating the above steps. From the above algorithm, to continue the decoding process each iteration can be triggered by the encoding symbol of degree one. It is important to guarantee that there always exist encoding symbols of degree one to release during the process for successful recovery. Note that information symbols in the ripple can reduce the degrees of decoding symbols. Information symbols in the ripple keep providing the encoding symbols of degree one after each iteration and, consequently, the decoding process ends when the ripple is empty. The decoding process succeeds if all information symbols are covered by the end. Therefore, generating ripple plays a vital role in decoding process of LT code. I will discuss the generating process of ripple in terms of degree distribution.

4.6.3 Degree Distribution Design

The degree distribution $\rho(d)$ is the critical part of LT codes design. Sometimes the encoded packets must have high degree like K in order to ensure that there are not some packets connected to single node. On the other hand, many packets must have low degree, so that the decoding process can get started, and keep going, and so that the total number of addition operations involved in the encoding and decoding is kept small. The guidelines of the distribution design are following [19]:

- The sum of all degrees should be as small as possible since it corresponds to the necessary operations of decoding process.
- As few as possible codewords are required to recover the message symbols. That means the release rate of encoding symbols is low in order to keep the size of the ripple small and prevent waste of encoding symbols. Similarly, the release rate of encoding symbols is high enough to keep the ripple from dying out.

Therefore, it is required to design degree distribution of encoded signal carefully so that release rate will be balanced. This is the reason that the degree distribution plays an important role in LT codes. Moreover, the encoding and decoding complexity are

going to scale linearly with the number of edges in the graph. Now what should be the average degree of packets? As I took ball-bin example and let we think that each ball and bin are connected through edges. In order to complete successful decoding, every source packet must have at least one edge in it. The encoder throws the edges into source packets at random manner, so the number of edges must be at the order of $K\log_e K$. So the average degree of each packet must be at least $\log_e K$. The encoding and decoding complexity of LT code will definitely be at least of $K\log_e K$. Luby [17] shows that this bound of complexity can be achieved by carefully choosing the degree distribution.

4.6.3.1 Ideal Soliton Distribution

The Ideal Soliton distribution displays ideal behavior in terms of the expected number of encoding symbols needed to recovery the data. Ideally this distribution ensures that one check node has degree one at each iteration. At each iteration, when this check node is processed, the degrees in the graph are reduced in such a way that one new degree-one check node appears. In expectation, this ideal behavior is achieved by this ideal soliton distribution. In this distribution, the degree distribution follows the following criteria:

$$\begin{aligned} &\rho(d) = 1/K \quad \text{for } d = 1; \\ &\rho(d) = \frac{1}{d(d-1)} \quad \text{for } d = 2,3,\ldots\ldots, K \end{aligned} \tag{36}$$

The expected degree under this distribution is roughly $\log_e K$. According to equation 36, $\rho(1) = 1/K$ represents the initial ripple size is 1. Now to ensure the ripple size increase 1 in each iteration, all the rest $\rho(d)$ should satisfy $\frac{1}{\rho(d)\cdot d\cdot K} = \frac{d-1}{K}$ and hence equation 36 is derived.

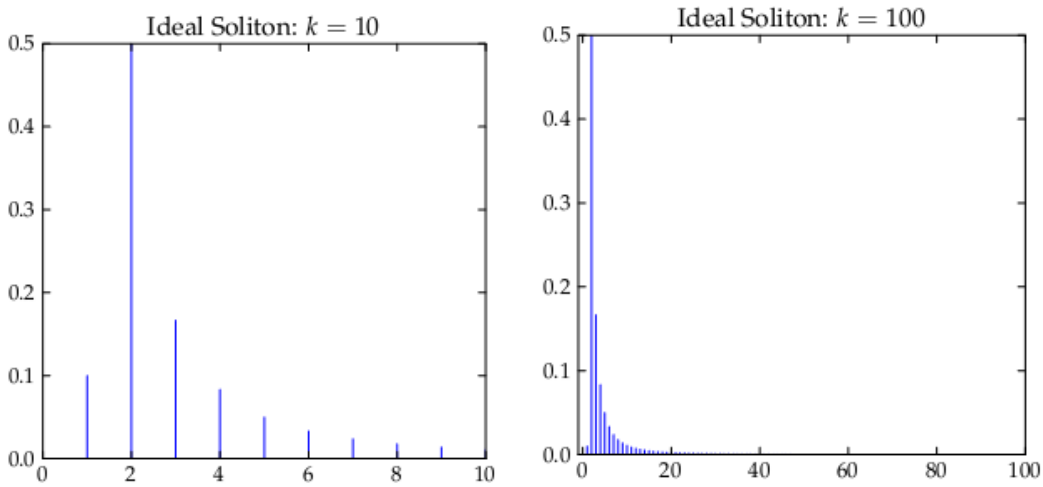


Figure 32: Ideal Soliton Distribution for $K = 10$ and 100 [14].

Figure 32 shows the performance of ideal soliton distribution for different message size. The Ideal Soliton distribution works perfectly in the sense that only K encoding symbols are sufficient to cover the K information symbols and exactly one encoding symbol is expected to be realized each time an information symbol is processes. Also in this distribution, the ripple is expected manner and there is neither the waste of encoding symbols nor the exhaustion of the ripple.

However, the practical scenario is different. In practice, the ideal soliton distribution shows very poor performance because fluctuations around the expected behavior make it very likely that at some point in the decoding process there will be no degree-one check nodes and, moreover, a few source nodes will receive no connections at all. Since the ripple size is one, it will disappear very easily during the decoding process, therefore the decoding will be failed under this distribution. Therefore, we need a distribution that ensures the ripple of large expected size enough to enable stable decoding as well as has the nice property of the Ideal Soliton distribution that maintains the expected ripple size constant in order not to waste encoding symbols. A small modification requires fixing these problems.

4.6.3.2 Robust Soliton Distribution

The problem of Ideal Soliton distribution is that the ripple size is too small so it may disappears easily. The intuition of the Robust Soliton distribution is try to solve this problem by increasing the ripple size to prevent the ripple from disappearing during the decoding process. Note that the required number of codeword symbols will increase with the ripple size, so it is also crucial to keep the ripple size small enough. The robust soliton distribution makes the ripple size $\log_e(K/\delta)\sqrt{K}$ through the whole decoding process. Therefore, the robust soliton distribution has two extra parameters c and δ ; it is designed to ensure that the expected number of degree-one checks is about $S \equiv c\log_e(K/\delta)\sqrt{K}$ rather than 1, throughout the decoding process. The parameter δ is a bound on the probability that the decoding fails to run to completion after a certain number K' of packets have been received. The parameter c is a constant of order 1. However, in practice, c can be a free parameter. Therefore, the robust soliton distribution is defined as below:

$$\tau(d) = \begin{cases} \frac{s}{K}\frac{1}{d} & \text{for } d = 1,2,...(K/S)-1 \\ \frac{s}{K}\log(S/\delta) & \text{for } d = K/S \\ 0 & \text{for } d > K/S \end{cases} \tag{37}$$

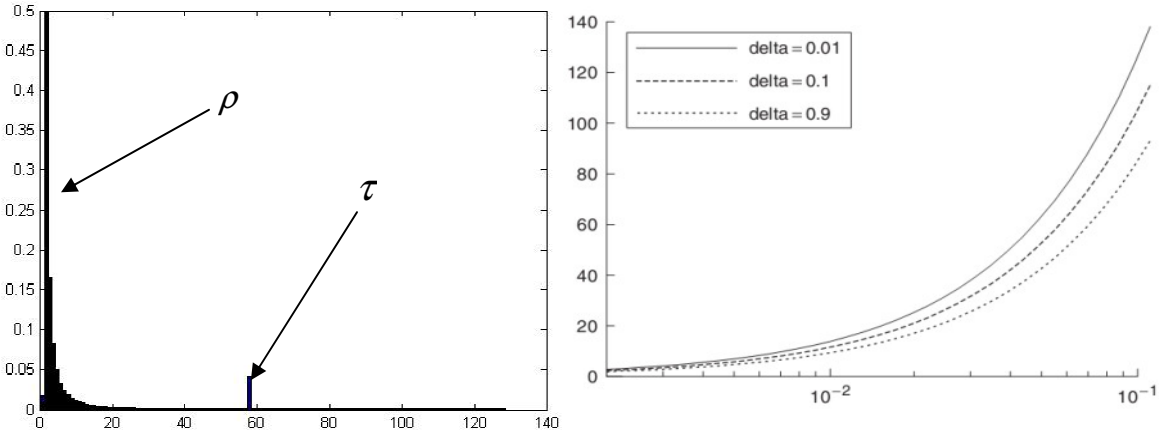
Then add the ideal soliton distribution ρ to τ and normalize to obtain the robust soliton distribution, μ

$$\mu(d) = \frac{\rho(d)+\tau(d)}{Z} \tag{38}$$

where $Z = \sum_d (\rho(d)+\tau(d))$.

In order to complete the whole decoding process, the number of encoded packets required at the receiving end with probability at least $1-\delta$ is $K' = KZ$. The detailed

analysis and comparison of Ideal and Robust Soliton distribution can be found in [18]. From the above explanation of Luby's analysis (specially equation 38), the small value of d at the end of τplays a vital role to start the decoding process and the spike in τ at $d = K/S$ ensures that every source packet is likely to be continued to check at least once. For constant value of c Luby's result shows that at the receiving end $K' = K + 2\log_e(S/\delta)S$ check nodes are necessary to finish whole decoding procedure with probability at least $1-\delta$. Figure 33 represents the comparative scenario of Ideal and Robust soliton distribution. Figure 33 (a) shows the distribution of $\rho(d)$ and $\tau(d)$ for K = 10,000, c = 0.2, δ = 0.05. The distribution ρ and τ are larger at d = 2 and d = K/S respectively. Figures 33 (a) and (b) are plotted against the two parameters c and del for K = 10,000. These figures prove that there exists a value of c such that given K' receives packets, the decoding algorithm will recover the K source packets with probability $1-\delta$.



(a) (b)

Figure 33 Comparative scenario of degree distribution (a) the distribution of $\rho(d)$ and $\tau(d)$ (b) number of degree-one checks S [18].

4.7 Hardware Implementation of LT Codec

It is mentioned earlier that the performance of LT codec is very high in digital fountain code paradigms. In previous sections, I have discussed the related theories

and mathematics of this codec. In this section, I will discuss some research articles those are related to the hardware implementation of LT codec system. I will discuss the proposed LT codec architecture in the next chapter.

Hardware Designs for LT Coding by Han Wang, Delft University of Technology [19]

In this research articles, two BEC models are proposed on different OSI layers and these channel models are used to analyze the performance of LT codec. Here H. Wang described an efficient architecture of LT codec that has a linear time complexity and the results of this architecture were measured in terms of time, area and coding performance. Now I will discuss the little bit more regarding this LT codec architecture. In broader aspect, total architecture divided into two parts: encoding architecture and decoding architecture. In encoding structure, the encoding steps are done by using $c^T = Hs^T$ equation, where c and s are vectors and H is the generator matrix. Here special memory architecture is required for this matrix multiplication. In the encoder block architecture, index counter, degree counter and global counter are used for indexing the degree value and neighbor nodes information. In this encoding architecture, a bit selector selects the neighbors of the encoding the codeword symbol from vector s following the neighbor position saved in H matrix. The modulo 2 operation is performed for generating the final value of codeword. The neighbor counting is indexed by the counter and sends to the index calculator. Finally the codeword is formed by applying the modulo 2 operation on the information symbols equal to the codeword symbol's degree value. Figure 34 shows the encoder architecture of LT codec.

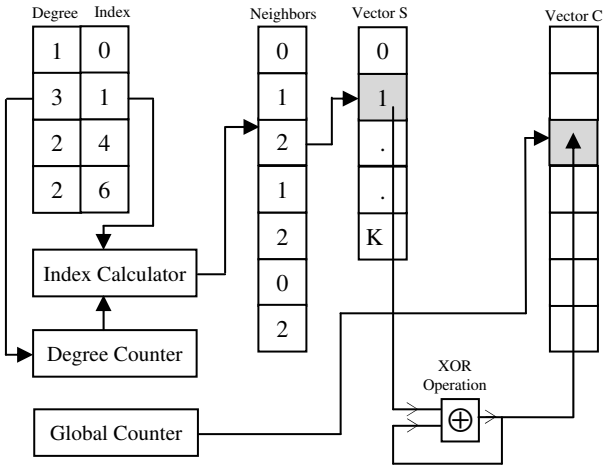


Figure 34: Hardware architecture of LT encoder [19].

Similarly, in the decoding process executes a set of operations on the receives generator matrix H to get the decoded symbols from the encoded value. In this decoding architecture, it has three stages. That means three search operations so it implies the time complexity is $O(n^3)$. In the first stage it will search for a row i in H such that degree $d(i) = 1$. In the second stage, it will search for the column j in the row i where $H(i)(j) = 1$. Then it will set $s'(j) = c(i)$, $c(k) = c(k) \oplus s'(j)$ and $H(k)(j) = 0$. Finally it will search for all $H(k)(j) = 1$ until there is no degree $d(r) = 1$. Figure 35 shows the decoder architecture of LT codec. Two simulation environments are used for implementing these architectures. ANSI C was used for implementing the software part of LT codec. For hardware implementation, this architecture was translated into HDL form and Xilinx Modelsim was used to compile this HDL program. Speed and area are reported from this tools by varying different message block length and code block length. During the simulation different code rate should be taken. The Spartan3E FPGA prototyping board had been used to measure the functionality of the encoder and decoder of LT codec. Theirs results shows that the hardware implementation was 500 times faster that the software implementation. However, there were some limitations of this research article. For example, there are no

comparisons with other encoding or decoding algorithms. Since it was implemented on the prototyping board so the performance scenario is no real like the original implementation using the standard cell during the chip design process.

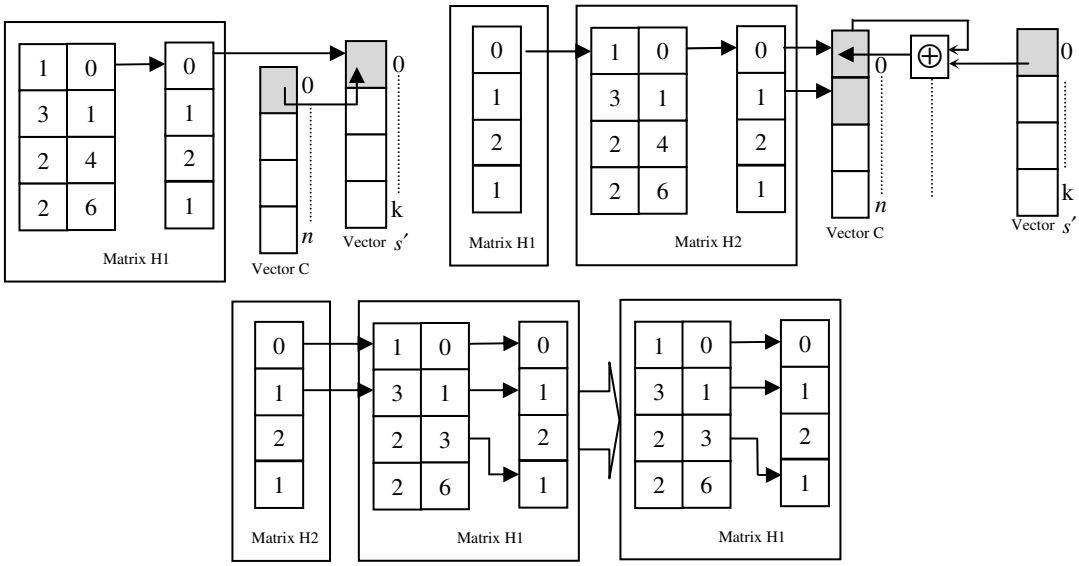
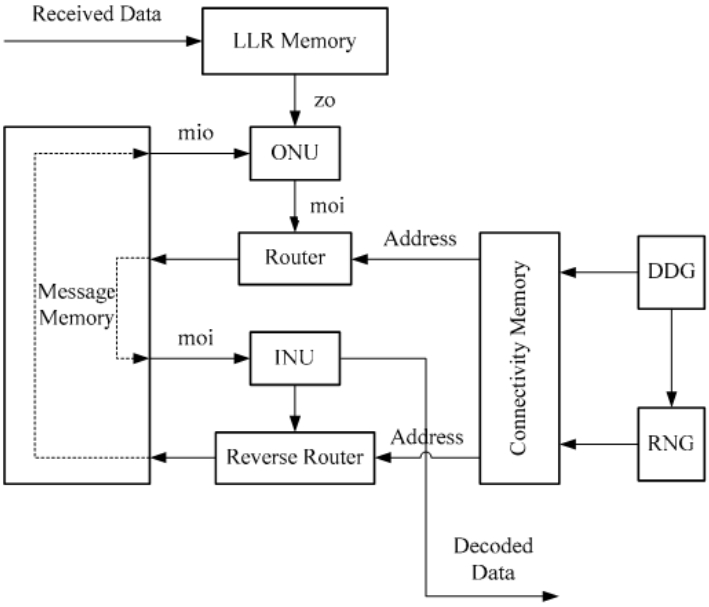


Figure 35: Hardware architecture of LT decoder [19].

Soft Decoder Architecture of LT codes by K. Zhang et.al. [20]

In this paper, K. Zhang *et. al.* presented an architecture of a soft decision LT decoder with a block length of 1024 bits and 100 iterations. Here, input node and output node processing techniques are described to accelerate the decoding speed. To apply these node-processing units, an efficient router and reverse router are designed to indicate the graphic connectivity between input and output nodes. The soft decoding procedure explained in this paper [20] is based on the sum product algorithm. In sum product algorithm LLR, message passing from check node to variable node or variable node check node operations are used which are elaborately explained in this paper. For implementation point of view, K. Zhang *et. al.* proposed an architecture for LT decoder which includes degree distribution generator (DDG), random number generator (RNG), message memory, connectivity memory, router and reverse router, output node processing unit (ONU) and input node processing unit (INU). Figure 36

(a) shows the LT decoder architecture proposed by K. Zhang. In this architecture, message memory is used to store the massage from check node node and variable node processing. ONUs are used for computing the message of check node using variable node message and the output of LLR memory. So the message memory stores the message from check node, variable node and LLR memory and fetched by ONU and INU during the time of iteration. In order to reduce the decoding latency partly parallel architecture is used in this architecture.

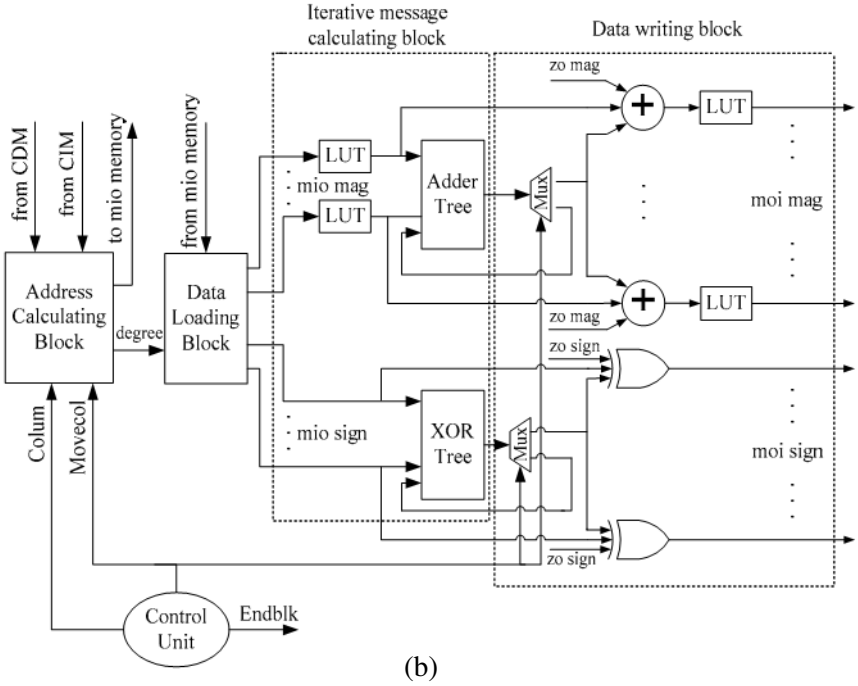


(a)

Figure 36: Architecture of LT decoder (a) complete decoder unit.

This architecture is responsible for concurrent use of input and output node processing. RNG is used for generating the degree distribution. According to this paper, degree distribution should be unchanged during the LT encoding and decoding procedure. A simple method is used for generating degree distribution using RNG and ROM. The connectivity memory stores the connection information between input and output node. That means, this memory stores the non-zero location of the generator matrix. In this paper for LT decoding process, row processing and column processing

are corresponding to the variable node processing (input node processing) and check node processing (output node processing). For this reason, a router block requires to control the proper memory location to store message from check node unit and variable node unit. Similarly, in this paper, architecture of output node processing unit and input node processing node unit are explained elaborately. In ONU look up tables are used for getting the 'tanh' result of message. Figure 36 (b) shows the ONU architecture for LT decoder. This architecture was synthesized and prototyped on Xilinx-V XC5V1x330 board. It shows that ONU consumes maximum registers as well as LUTs on the FPGA prototyping board.



(b)

Figure 36 : Architecture of LT decoder (b) output node processing unit [20].

A scalable LDPC decoder ASIC architecture with bit-serial message exchange by T. Brandon, et. al.[21]

In this paper, T. Brandon *et. al.* presented a scalable bit serial architecture of LDPC decoder. Here the decoder was implemented for a (256,128) regular (3,6) LDPC code

using TSMC 180-nm 6 metal CMOS technology. It has a decoded information throughput of 350 Mbps, core area is 6.96 mm^2 and energy efficiency is 7.56 nJ per uncoded bit at low SNR. In this architecture the decoder is fully block parallel. All bits of 256 codeword are processed by 256 variable nodes and 128 parity check nodes that together form an 8-stage iteration pipeline. For decoding the LDPC code, sum product algorithm was used which is also known as min-sum algorithm. As it is mentioned earlier that it has 128 check nodes and 256 variable nodes, so in it's decoder architecture 128 CNUs and 256 VNUs are interconnected by using interleaver network. In its VNU architecture, the variable nodes are connected into two 128-node arrays. Each array is linked by two 4-bit wide LLR buses. Similarly, each variable node contains two 4-bit registers for holding the LLR channel measurements for the two codewords being decoded. In addition, there are 4-bit shift registers for receiving the message from parity check node via interleaver network. These register holds the binary values that pass through the combinational logic that converts the values from sign-plus-magnitude format to two's complement format, forms three 6-bit sums for the outgoing messages, and converts the 6-bit sums via a saturation operation to three 4-bit sign-plus-magnitude output message values [21]. Three 4-bit shift registers are used to capture the new output messages. The pipelined interleaver contains two 4-stage shift registers in the variable node, one flip-flop in each interleaver direction and one register in the parity check nodes, for a total eight pipeline stages. The parity check node receives six bit-serial input belief message from the interleaver and computes then corresponding six bit-serial output belief messages using the standard min–sum algorithm. The details algorithm of this check node unit was described in the ref [21] including sum product algorithm. This decoder architecture was fabricated in TSMC's 180-nm 6-metal CMOS process using the SAGE-X standard cell library. Figure 37 shows the architecture of LDPC decoder proposed by T. Brandon *et. al*. To speed up the simulation run time and reduce the memory allocation, the variable nodes were grouped into pairs together with a small controller circuit and then these sub-

blocks were synthesized. The dimensions of the resulting IC core for the (256,128) code are 2639.4 µm x 2639.4 µm = 6.96 mm^2 and the total chip area is 10.82mm^2. The logic utilization area in the core is 86%. There are 259 logic gates per check-node pair, 1183 gates per variable node pair, and a total of 188,84 8 gates before clock- tree generation and buffer insertion. During chip-level final synthesis, an addition 3557 gates were added bringing the total gate count to 192,405. The operating voltage of this chip is 1.62 V and 4 ns clock period. The variable node unit consumes the maximum power which is almost the 75% of total power. This paper shows the ASIC implementation of LDPC decoder which is used the sum product algorithm as a part of decoding process. Since LT decoder also used the sum product algorithm, for this reason I have included this hardware architecture of this paper.

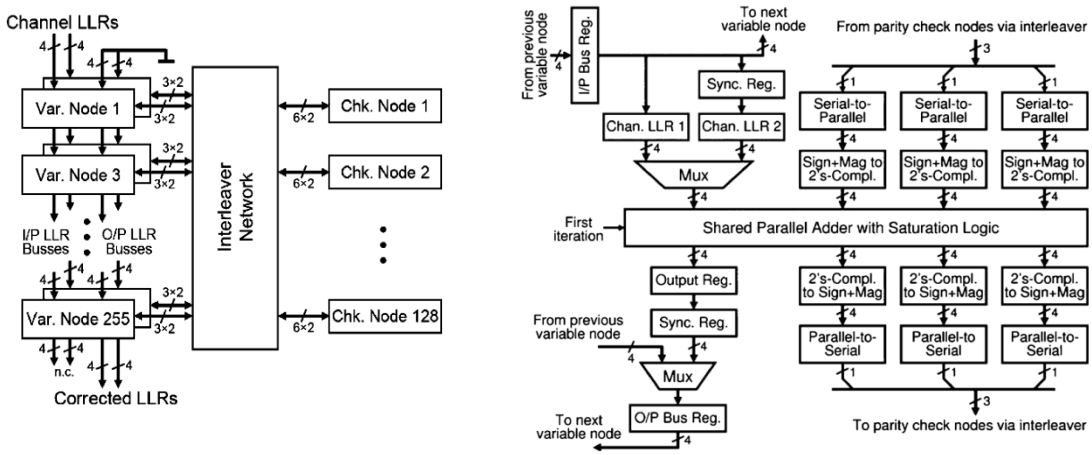


Figure 37: LDPC decoder architecture (left) and variable node unit block diagram (right) [21].

These hardware architectures are designed for ASIC implementation of LT codec. At first, these structutres are translated into HDL which is known as RTL design and then this RTL design is ready for further processing of chip design procedure. In this thesis, we are interested about the application specific processor design of LT codec application. Moreover, at the end of this processor design, RTL design will be generated by the ASIP design tools. Next chapter, we will discuss the LT codec processor design techniques.

Chapter 5

LT Codec Processor Design Using ASIP Tools

In this chapter, we will show the processor design techniques using three tools: TCE, Tensilica and OpenRISC. For application specific processor design, at first it requires two design files: one is input application written in HLL (for example in this work ltcodec.c file) and second one is processor architecture file (for example architecture definition file *.adf*, configuration file *.cfg* etc). These two design files are key structutres for processor design in application specific domain. The response of the processor depends upon these input design files. For this reason, it is very important for designers to make efficient architecture of input application and configuration files. At first, we will discuss the proposed architecture of LT codec. Then processor design parts will be discussed.

5.1 Proposed Architecture of LT Encoder and Decoder

In order to understand the LT codec architectures, at first we present an architecture for ASIC realizations of the Luby Transform (LT) encoder and decoder. However, for processor design we required HLL translation of LT codec. After discussing RTL architecture, we will discuss HLL architecture in corresponding sections.

To determine the efficiency of the LT codec architecture, the encoder and decoder are implemented with a core area of 9 mm^2 in TSMC 180-nm 1-poly 6-metal and Samsung 130-nm complementary metal–oxide–semiconductor (CMOS) technology. An empirically modified Robust Soliton degree distribution technique is applied for LT codec implementation and its performance is analyzed in terms of chip area and cycle count. Instead of including a random generator in the register transfer level (RTL) design, we use different look-up tables (LUTs) for degree distribution, edge routing, addressing and inverse edge routing. Therefore, this architecture is efficient

for hardware implementation and occupies less area inside the chip. The result shows that an area of 2.3 mm^2 is required for whole encoder and decoder implementation using TSMC library, of which 0.08 mm^2 is used for encoder implementation.

5.1.1 HW Architecture of Encoder

In an encoder, a long output encoded sequence can be produced from k input symbols $\{S_1, S_2, S_3, S_4 ..... S_k\}$ as $c_i = S_{i,1} \oplus S_{i,2} \oplus S_{i,3} \oplus ..... \oplus S_{i,d}$.

Output degree d is taken randomly from a degree distribution function explained in section 4.6.3. Figure 37.1 shows the hardware architecture of the LT encoder for 128 input bits and 256 output bits. This HW architecture is compatible for implementing ASIC implementation of LT encoder. Therefore, in this paper, for ASIP implementation, we have written this architecture in C language. For example in figure 37.1, two look-up tables (LUTs) are used to satisfy the degree distribution. If the degree distribution is 4, then pick 4 consecutive rows of address message column (4, 6, 3, 2) that point out the message value of the corresponding address of the message signal. But in our ASIP architecture instead of LUTs we have used mathematical expressions: Robust Soliton Distribution (RSD) and Ideal Soliton Distribution (ISD) for calculating degree distribution. Moreover, a uniform Random Number Generator (RNG) is applied to get the degree value from this degree distribution. In contrast, for ASIC design, the address of the message signal is randomly distributed and the combined operations of the column for degree distribution and the address of the message satisfy the distribution mentioned in equation 37. These same LUTs are also used for decoding of the encoded signal. For this reason, in ASIP design, we have translated the encoding process of LT codec in HLL by satisfying the minimum execution of operation which is very simple compared to the use of LUTs. In figure 37.1, the message signals identified by one row of the degree distribution column are added and the result is stored in a temporary register. For example, in the degree distribution column, the degree value is 4 then

message signals of address (4, 6, 3, 2) positions are identified as (1, 0, 1, 0), respectively, and the result of this addition is stored in a temporary register as 2 and after applying the modulo 2 operation, the encoded signal for degree distribution 4 is 0. The 256 bit encoded signal is generated according to the same procedure as used for the 128 rows of degree distribution column and 128 bit message signal.

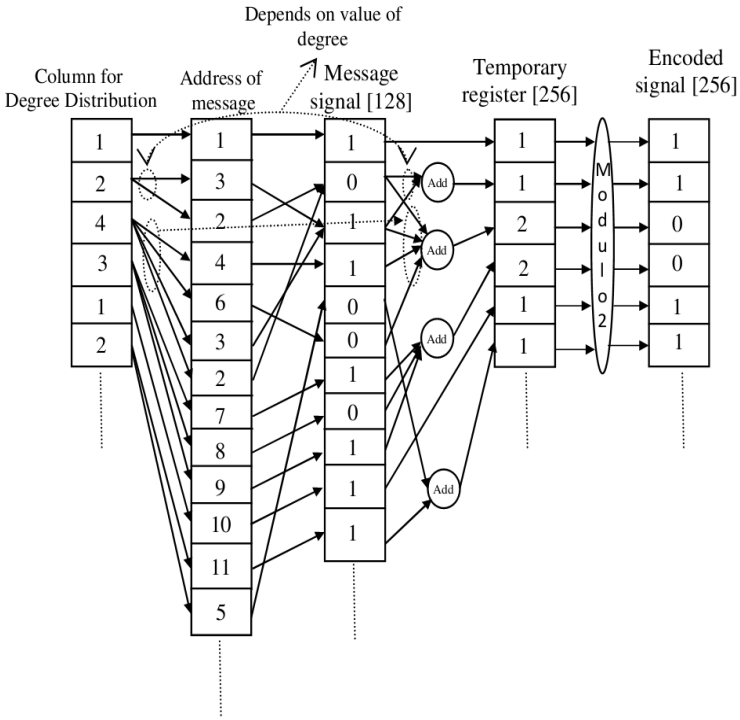


Figure 37.1: Architecture of LT Encoder.

The following process is compatible for encoder architecture mentioned in figure 37.1.

1. Create the two lists *D* and *A* that represent the degree and address of the message table, respectively.

2. Take a variable *x* that indicates the first element of table *D*.

3. Find the value of degree number *d* and let $D(x) = d$ and let $A(x) = A(x\text{-}1) + d$.

4. Take *d* numbers from 0 to *k*-1 message column, where *k* is the length of the original message by using the address $A(x)$ and save the result of their addition into the

temporary register.

5. Apply modulo 2 operation on the temporary register column.

6. Repeat steps 1 to 5 until the codeword is formed.

Here the LUTs *D* and *A* are generated from equation 1 using a computer program. The above-mentioned activities can be done very easily in HLL by obeying the following algorithm:

Input: message **x** = { $s_1, s_2, s_3, \ldots, s_K$ }, probability distribution $\rho(d)$ on N_K

Output: an encoded symbol t_n

1. Sample an output degree *d* with probability $\rho(d)$
2. Sample *d* distinct message symbols $\{s_{i_1}, s_{i_2}, \ldots, s_{i_d}\}$ uniformly at random from the message

   $\{s_1, s_2, s_3, \ldots, s_K\}$ and XOR them, $t_n = \oplus_{j=1}^{d} s_{i_j}$.

In this thesis, we have executing this algorithm as an encoder technique of LT code and designed encoder processor using ASIP tools. Now I will explain the decoding process of LT code.

5.1.2 HW Architecture of Decoder

In LT codec, decoder is more complex than encoding part. From this encoding explanation, it can be found that direct RTL mapping is quite difficult than HLL mapping. However, at first we will discuss the RTL design process of LT codec. Then we explain the decoding procedure using HLL mapping. In this LT codec implementation, we have taken 128 bits for information signal and 256 bits for encoded signal. In order to get decoded signal from encoded bit stream, soft decoding procedure is applied by using sum-product algorithm.

Channel decoding in an LT decoder is based on the log likelihood ratio (LLR) of a binary random variable $X \in \{\pm 1\}$ or $X \in \{0,1\}$ defined by the following equation,

$$LLR(X) \overset{\Delta}{=} \log\left(\frac{\Pr\{X=1\}}{\Pr\{X=0\}}\right) \tag{39}$$

where LLR(X) represents the LLR corresponding to bit X, and P(X = 0, 1) represents the probability that bit X is equal to 0 or 1. The LT decoder operates based on the sum product algorithm by passing the message (LLR values) on tanner graph. Let $L(t_{i,j})$ denote an L value message passed from check node i to variable node j and $L(h_{i,j})$ denote an L value message passed from variable node i to check node j. Then from [8], $L(t_{i,n})$ can be written as:

$$L(t_{i,j}) = 2\tanh^{-1}\left(\tanh\frac{L(\hat{c}_i)}{2}\cdot\prod_{n\in \mathrm{N}_i, n\neq j}\tanh\frac{L(h_{n,i})}{2}\right) \tag{40}$$

where $L(\hat{c}_i)$ denotes the received L value of the codeword from the channel. Similarly, the L value $L(h_{i,j})$ depends on the messages passed to variable node i. So $L(h_{i,j})$ can be obtained by [8]

$$L(h_{i,j}) = \sum_{e\in\varepsilon_i, e\neq j} L(t_{e,i}) \tag{41}$$

Similarly the L value about the decoding decision [8]

$$L(\hat{u}_i) = \sum_{e\in\varepsilon_i} L(t_{e,i}) \tag{42}$$

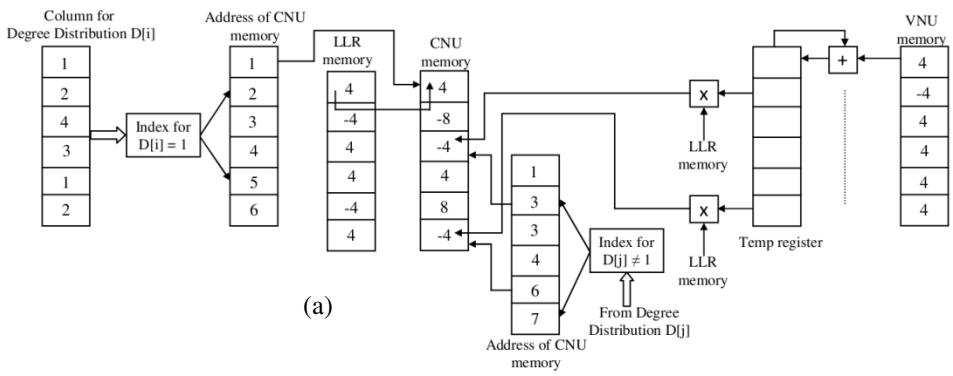


Figure 37.2: Hardware architecture of the LT Decoder: (a) CNU architecture.

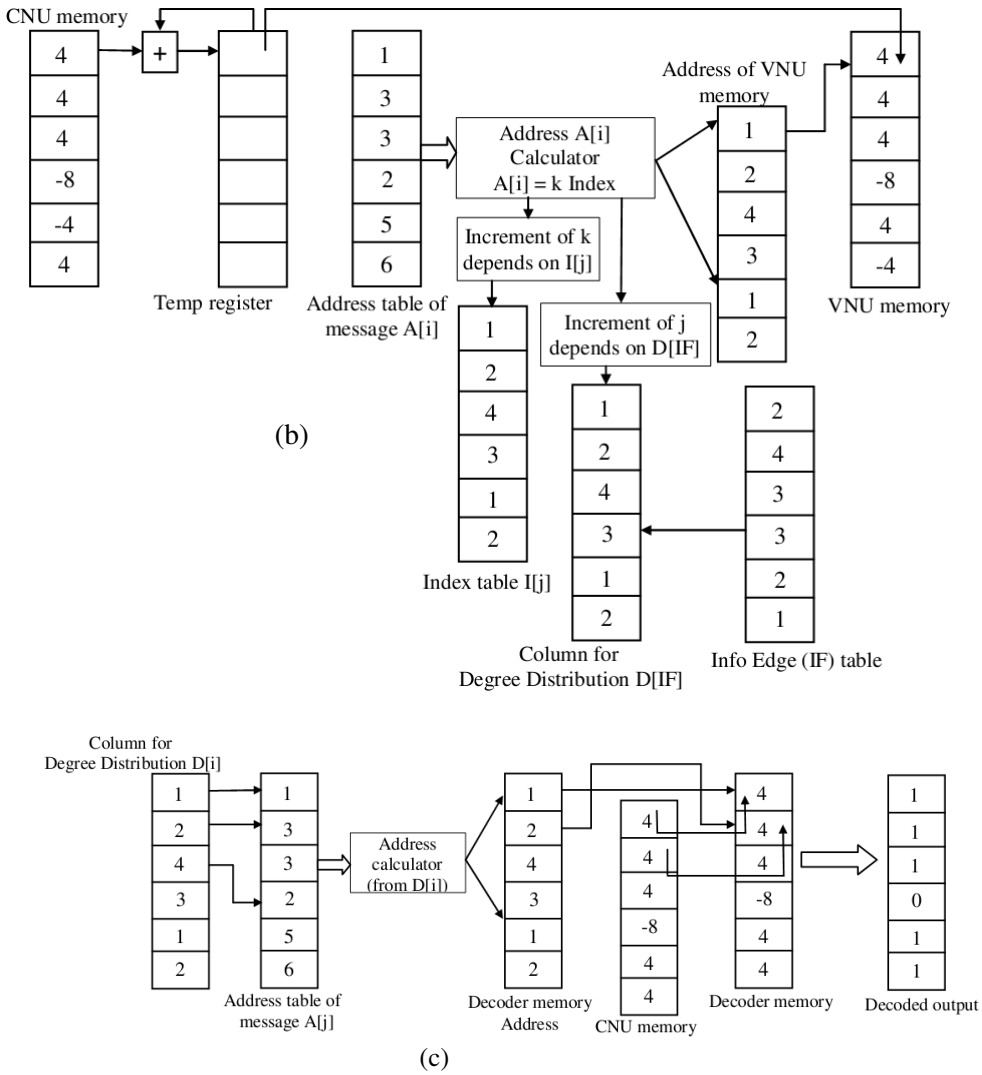


Figure 37.2: Hardware architecture of the LT Decoder: (b) VNU architecture, and (c) Final decoding stage

During this decoding process, the messages are exchanged back and forth in a number of decoding iterations between the variable nodes and check nodes. The LT decoder operates based on the sum product algorithm by passing the message (LLR values) on tanner graph. For example, equations 40 and 41 are responsible for implementing the check node unit (CNU) and the variable node unit (VNU) and equation 42 is used as

the decoding final stage. In decoding architecture, these equations are implemented in different stages and the working principle of this architecture is discussed in the next section.

CNU Operation:

In the CNU module, LLR memory is used for check node operation while the message is passing through the check node. Like encoder, the same degree distribution table is used so that when the degree is one, the counter counts the position of unity degree and CNU memory stores the message of the count address value from LLR memory. Then, the counter counts further and when the degree is not equal to one, the message from LLR of that count address is multiplied with the message from VNU memory through the operations presented in figure 37.2 (a). The CNU memory therefore has messages for degree one and updated messages for a degree greater than one. Messages pass through these CNU nodes and updated messages are stored in the CNU memory. The operations of CNU are executed as below:

1. Search for a row in degree table where $d(i) = 1$.
2. Take the message from LLR memory and store it in CNU memory, $L(i) = C(i)$.
3. Search for a row in degree table where $d(i) = x$, $x \neq 1$.
4. For each x, temporary register $T(j) = T(j) * V(j)$ and $C(j) = T(j)*L(j)$ where $j = 0,1\ldots x-1$.

VNU Operation:

As shown in figure 37.2(b), each variable node contains 4 LUTs. Two new LUTs termed as edge information and index tables are included in VNU operation. These additional tables consist of nodes and edge information provided by the degree distribution function. The VNU function unit takes data from CNU memory and stores it in VNU memory after following the operation of node routing and inverse node routing explained in figure 37.2(b). In VNU, the processing unit accumulates messages serially from the check node and stores them in the variable node memory.

The operation of VNU can be written as below:

Search for a row in address table $A(i)$ such that $A(i) = K$ and the increment of K depends on index table $I(j)$. Here, j is the variable of the Index table and its increment depends on Degree table $D(l)$. Edge information table controls the value of l in a prescribed manner. So, for $A(i) = K$, VNU processing unit accumulates LLR message format from CNU memory and stores it in the VNU memory unit.

Final Decoding Stage:

After finishing the CNU and VNU operations, CNU memory contains all the nodes and edges of the processing information. Degree and address LUTs are used for generating addresses for the decoder memory. Then, data read from the decoder memory are taken as the decoded output. Figure 37.2 (c) shows the final decoding stage architecture. Its algorithm is given below:

1. For each element of degree $D(i)$, increase the index variable k until $k = D(i)$.
2. When $k > D(i)$, then $k = 0$.
3. For every value of i and k, take the value from the address table A.
4. Using this address value, store the information from CNU memory in the decoder memory.
5. Finally, the decoded output is generated from the decoder memory.

From the above discussion, the whole decoding process is explained through the LLR operation, CNU and VNU processing unit and final decoding stage. It is performed by passing messages from check nodes to variable nodes and vice versa. Therefore, this decoding is an iterative process and messages are decoded from the code value after certain iterations. This 144 quad flat package pin LT Codec chip is fabricated by applying TSMC 180nm technology.

5.1.3 Decoding Procedure Using HLL

Decoding algorithm has been developed by using these three equations from eq. 40 to eq. 42. Through these equations, the message information is passing from check

node to variable node and variable node to check node of tanner graph. Figure 37.3 shows the typical tanner graph of LT codec. Figure 37.4 shows the HLL mapping of LT decoder. In this case, we have followed the algorithm explained by equation 40 to 42. According to this figure, first we have taken one 2D array ($L(t_{i,j})$) size of encoded signal length by maximum degree value. In encoding end, we have already generated the edge, index of those edges for variable node and degree value of check node that are explained in figure 37.3. At first we need to search which check node has single degree that means if degree is one then store the LLR values of that check node to $L(t_{i,j})$ memory. Otherwise store the message passing value calculated by using eq. 40, 41, edge and degree information in $L(t_{i,j})$ memory. Then we have taken another 1D array ($L(u_i)$) size of information signal length by one. According to eq. 42, the message value of each variable nodes should be stored in $L(u_i)$ memory. After that decoded signal is found by applying the hard decision according to figure 37.4.

In this section, we have discussed the LT encoder and decoder architecture in terms of HLL format. After that, we need to explain LT codec processor generation techniques by using ASIP design tools.

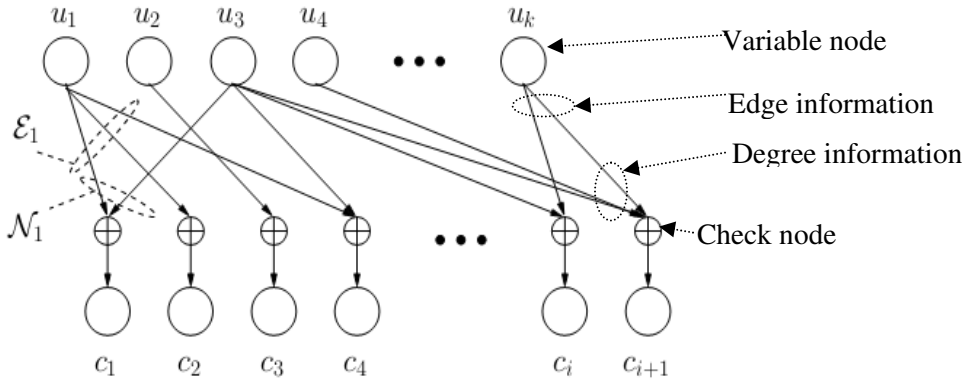


Figure 37.3: LT Codec tanner graph.

$L(t_{i,j})$ memory
(size: encoded signal length x maximum degree value)

- 4	0	0	0
4	0	0	0
-4	0	0	0
-4	0	0	0
4	0	0	0
-4	0	0	0
3.9	-3.9126	0	0
3.9	0	0	0
3.82	-3.91	0	0
3.03	-3.99	0	0

(if degree = 1) $L(t_{i,j})$ = LLR of encoded signal value

(if degree ≠ 1)

$$L(t_{i,j}) = 2\tanh^{-1}\left(\tanh\frac{L(\hat{c}_i)}{2}\cdot\prod_{n\in \mathrm{N}_i, n\neq j}\tanh\frac{L(h_{n,i})}{2}\right)$$

$$L(h_{i,j}) = \sum_{e\in\varepsilon_i, e\neq j} L(t_{e,i})$$

$$L(\hat{u}_i) = \sum_{e\in\varepsilon_i} L(t_{e,i})$$

7.3358
14.3050
-10.0862
-13.4066
9.3384
5.2213
14.2883
19.0292
7.2319
6.6509
-8.7674
-13.4615
-22.4657
21.0193
-26.1414
8.4995

$L(u_i)$ value decoding memory
(size: original signal length x 1)

$L(u_i)$ >= 0 then
Decoded signal = 0
else 1

0
0
1
1
0
0
0
0
0
0
1
1
1
0
1
0

Decoded Signal
(size: original signal length x 1)

Figure 37.4: Decoder structure using HLL.

5.2 Processor Design Using ASIP Tools

It is necessary one tool set for implementing application specific processors based on the TTA processor template. As mentioned earlier that there are different tools to satisfy this requirement. TTA based Co-Design environment (TCE) is one such tool set and its main goal is to provide a reliable and effective toolset for designing programmable application specific processors and generates machine code for them from applications written in high level language (HLL). This toolset is developed by Tampere University of Technology [22]. Processor Design (ProDe), a retargetable high level language compiler tcecc, the retargetable Instruction Set Simulator (ISS) ttasim (command line version) and proxim (graphical user interface version) and the processor generator ProGe are the most essential properties used in TCE. Using this tool, application written in high level language can be implemented in FPGA evaluation board through RTL design flow. The concept of retargetability of tools means that it can be automatically adapted to the processor architecture during run time. In TCE the designer can customize the TTA processor that means the architecture file can be modified by adding or removing FUs, RFs, data buses and even by using user defined FUs. The designer can also change the width and number of GPRs. So this tool is very flexible and customizable to improve the processor performance in terms of cycle count or other overheads.

5.2.1 ASIP Design with TCE [23]

The main goal of TCE ASIP design flow is to produce a processor in HDL language and implement this generated processor to chip design process or an FPGA evaluation board for checking functionality. Figure 38 (a) shows the complete design structure of TCE ASIP design flow. From this figure, it can be shown that the desired application in HLL and the design requirements are applied as inputs of the design flow. The design requirements may include the amount of FPGA resources, the target execution

time, the minimum clock frequency, as well as energy, area etc. So, at the beginning of the design flow it is required a starting point architecture which is known as Architecture Definition File (ADF). The structure of architecture is very important to meet the desired requirements and there are flexible activities to modify this architecture to meet the requirements. Therefore, the aim of this thesis is to depict the response of different ADFs to reduce the cycle counts to implement the input application. However, next this source code with starting point architecture (ADF) is compiled by the tcecc compiler and generates TTA Program Exchange Format (TPEF) binary file. Then the retargetable instruction set simulator ttasim receives these two files (TPEF and ADF) as input and produce the simulation results. Execution cycle count, processor resource utilization and optimally execution trace are included in this simulation result [3]. These simulation results are then feed backed to the starting point architecture (ADF) to adjust the parameters. If the minimal structure of ADF fails to meet the requirements then custom architecture is applied for simulation. However, this iteration process is known as manual processor Design Space Exploration (DSE) [3]. TCE also includes explorer tool to automate this DSE operation. On the other hand, TCE allows the designer to customize TTA processor that is FUs and transport buses etc are modified according the designer requirements. This custom operation is allowed to accelerate the application. The custom operation design flow is shown in the following figure 38 (b). From this figure, first, it is required to find a custom operation then the designers create a custom operation compiler definition by using Operation Set Editor tool (OSEd). In order to simulate the custom operation FUs, it is required simulation models written in C/C++. After this, the processor architecture and HLL source code are modified according the custom operation. In HLL source code this is done by calling the operation via TCE-specific operation macros or intrinsic. Then the feedback is taken to get the response of the new custom design and if the result is not satisfying then it is modified or another custom operation can be tested [3]. In this thesis, I showed the performance of

this custom operation in terms of cycle count, resource utilization for LT encoder and decoder as an input application file. Figure 39 (a) shows the simulation behavior of the typical custom function unit. This figure describes the architectural simulation behavior of the ADD operation. The first and the second operand (id 1 and id 2) are added up and the result is written to the output with id 3. OSAL architecture does not include the operation latencies of the custom FU.

Figure 39(b) shows an example of a TTA processor datapath using TCE tool that consists of FUs, RFs, a Boolean RF, and a custom interconnected network [24]. These data transports are clearly programmed and written to a trigger port of functional units. Figure 39 also represents instructions, defined as moves, for three buses [24]. An explanation of these instructions is given in the next section. In this figure, moves are defined for three buses performing an integer summation loaded from memory and a constant.

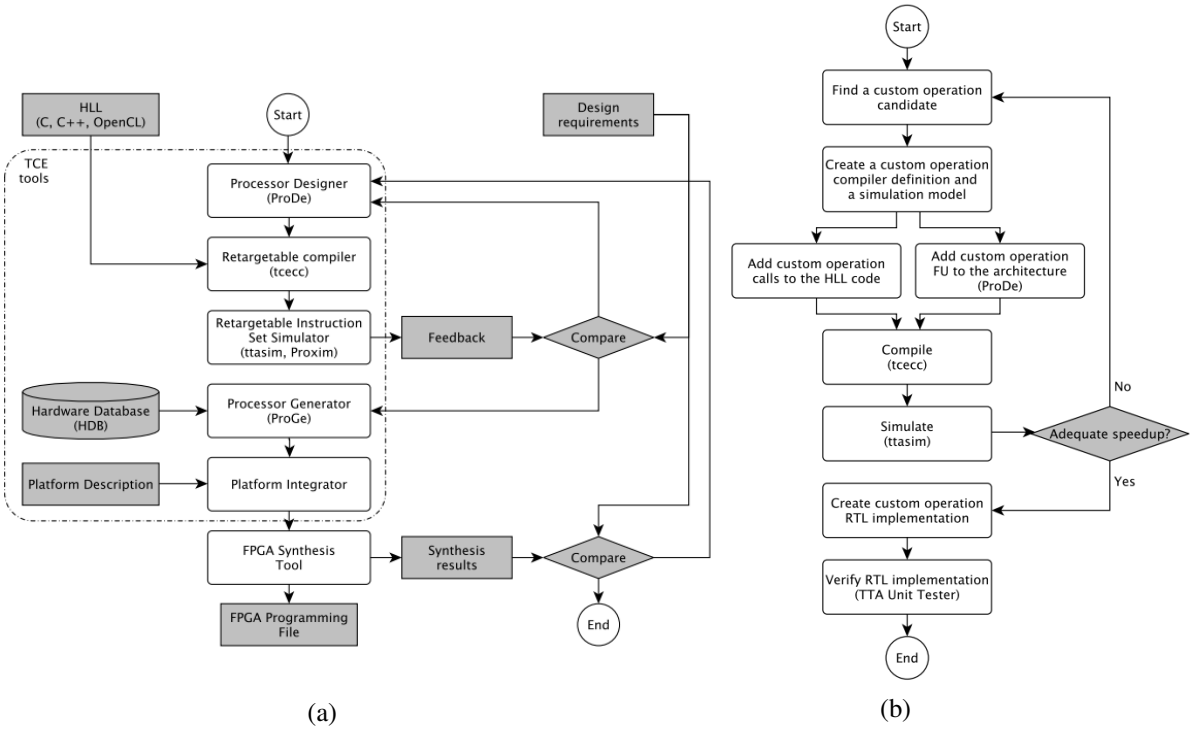
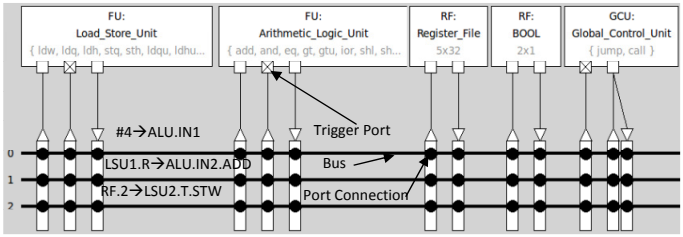


Figure 38: TCE design flow: (a) from HLL to FPGA [3] (b) TCE custom operation design flow [3].

```
#include "OSAL.hh"

OPERATION(ADD)
TRIGGER
    IO(3) = INT(1) + INT(2);
    RETURN_READY;
END_TRIGGER;
END_OPERATION(ADD);
```

(a)



(b)

Figure 39: TCE operation (a) simulation behavior of custom FU (b) Example of TTA processor data path with 3 instructions for three buses [2].

5.2.2 Processor Design Space Exploration

Design space exploration is defined as the process of finding target processor architectures with desired performance for a given applications. In TCE this process can be largely automated but trial-and-error process should be followed to get more efficient target processor. Figure 40 represents the automatic design space exploration process of TCE. TPF is a suffix of the files used to store TTA programs stored in our TTA Program Exchange Format files. ADF and IDF are the two file formats for describing the architecture and implementation data of the processor, respectively, as presented in the previous section. A processor configuration consists of an ADF/IDF pair. The design space explorer modifies resources of a given architecture and passes the modified architecture to the code generation and analysis phase for evaluation. As a result, it will produce the estimate of consumed energy, number of cycles and cycle time. This process is repeated for each modified architecture until satisfy the target architecture goal.

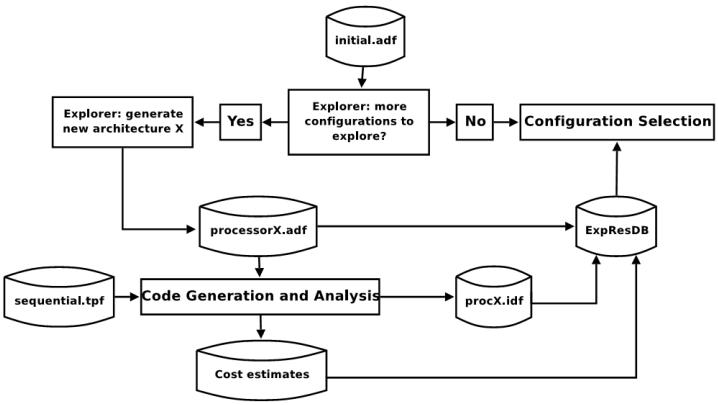


Figure 40: Automated Design Space Exploration [23].

5.2.3 TTA Programming

In TTA programming, data transports are required to read and write the operand values, and the operation is triggered when data is written to a trigger port. Sequential and parallel TTA programs represent the sequence of instructions depending on a number of buses. In sequential TTA programming, the moves are sequentially executed because of single bus architecture. Therefore, its code is not scheduled to be executed in a target structure. In a parallel TTA program, a set of moves is executed using a multiple bus structure. Therefore, each bus will be utilized in parallel in the same clock cycle. Thus, instruction level parallelism (ILP) is exploited in a parallel TTA architecture. An example of a simple TTA program is given below [25]:

```
1: 100 ->RF.1 ; 500 -> RF.2
2: RF.1 -> ALU.add.1; RF.2 -> ALU.sub.1
3: 50 ->ALU.add.2 ; 100-> ALU.sub.2
4: ALU.add.3 ->RF.1 ; ALU.sub.3 ->RF.2
5: RF.1 ->ALU.EQ.1 ; RF.2 -> ALU. EQ.2
6: !ALU. EQ.3->bool;  ……
7: !bool 2-> GCU.jamp.1
```

Here two buses are used in TTA architecture so that a couple of instructions are executed in one clock cycle. In Line 1, two general-purpose registers (RF) take

constant values from the immediate unit and store those values in the ADD and SUB modules of ALU through a load store unit (LSU). This is explained in Line 2. After finishing the similar operations in Lines 3 and 4, RF1 and RF2 hold the output values of ADD and the SUB module of ALU. Line 5 shows that these two values from GPRs are applied to two inputs of the equator (EQ) module of ALU. In Line 6, the result of the comparison is transferred to a Boolean register, which is used in conditional execution. In the last line, the value of the Boolean register is evaluated and the jump operation of the global control unit (GCU) is triggered in case a Boolean register value is false. That means the program execution is transferred back to Line 2 when the values of RF1 and RF2 are not equal. For this example, the second operand of the ADD, SUB, and EQ operations, and the first operation of the JUMP operation, are triggering ports. Therefore, this whole comparison operation is done in 7 cycles, and each cycle executes two operations for two bus architectures. That means, depending on this ILP, the speed of the processor is identified. Single bus architecture would require almost 12 cycles to execute this operation. The assembly notations of this example are taken from the TTA Based Co-design Environment (TCE) tool [25].

In TTA architecture, it is possible to add a new instruction to the target processor which implements arbitrary functionality. This custom instruction reduces longer chain operations to a single custom operation. To add this custom instruction, the ADF files of the TTA processor should be modified by introducing a new FU. In this thesis paper, we showed the ways in which the instructions set are generated from each custom function unit. The generating procedure of each efficient custom function unit, modification of ADFs, and reference design are discussed in the author's other paper [4]. The TTA code generation techniques for this FU, named CRCFAST, are discussed in detail. Moreover, this new custom architecture for implementing CRC is very efficient in terms of cycle count which is also discussed in-depth in [26].

5.2.4 Code Generation Method Using TCE Tool

In the previous section, we discussed the assembly instruction of the TTA processor, which was applied to Architecture Definition Files (ADFs) in the TCE tool [6]. In this section, we will discuss the code generation technique which is the main part of whole design flow in the TCE structure. Before going to discuss the code generation technique using TCE tool, we will show the advantage of customized code generation for TTAs. It is well-known that VLIW and TTA based processors exploit the ILP at compile time. Here, compiler finds the parallel instructions before run time. VLIWs are constructed from multiple, concurrently operating FUs where each FU supports RISC style operation. But the traditional VLIW processor architecture is not suitable for scalable operation because of its complex connectivity of required datapath especially for register file (RF) and bypass circuit. The data bandwidth and instruction bandwidth depend on the number of selected FUs. However, when all FUs are utilized, the available data bandwidth is still rarely utilized. For that reason, the concept of TTA and its code generation techniques are required. The complete design flow is divided into four phases: Initialization, Design Space Exploration, Code Generation, and Processor & Program Image Generation [25]. In initialization phase, the sequential code form of the TPEF file format is generated by compiler like TCECC (TCE C Compiler) including the architecture definition file (ADF). If this compiler is provided with multiple compilation units, the TPEF linker links them to a single TTA Program Exchange Format (TPEF) binary file. This TPEF file format is used for storing unscheduled, partially scheduled, and scheduled TTA programs to apply input to TCE. The compiler used here is known as a frontend compiler because it has no more use in the rest of the TCE toolset. Now, for TCE version 1.5, this compiler can compile only in the high level C language. Design space exploration is used to estimate the cost for different starting point architectures (ADFs). The goal of this phase is to find an optimal architecture for input design. Here the explorer removes the unused connections and resources from the starting point architecture, which is

more beneficial in terms of area, power, and time. It should be noted that if a program is simulated using various types of efficient target architecture modified either automatically or manually, parallel simulation is invoked to increase processor speed. Therefore, the Explorer creates a database named the Exploration Result Database (ExpResDB), which contains the configuration of evaluations during exploration. It also creates an Implementation Definition File (IDF) for estimating the cost of each explored target architecture.

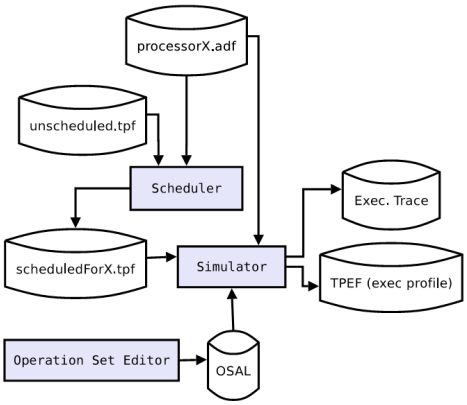
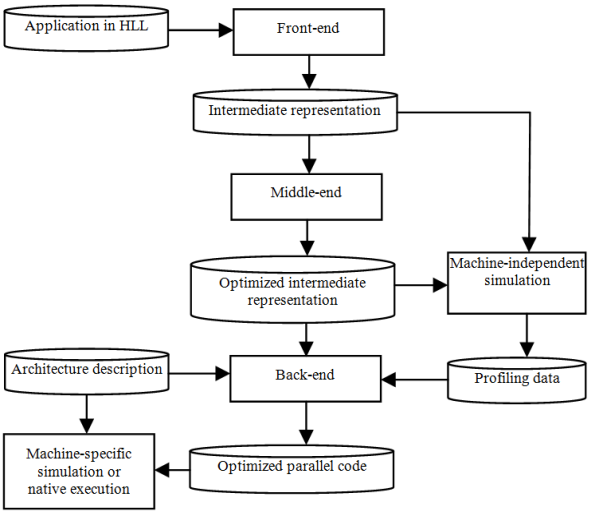


Figure 41: Code generation and analysis [25].

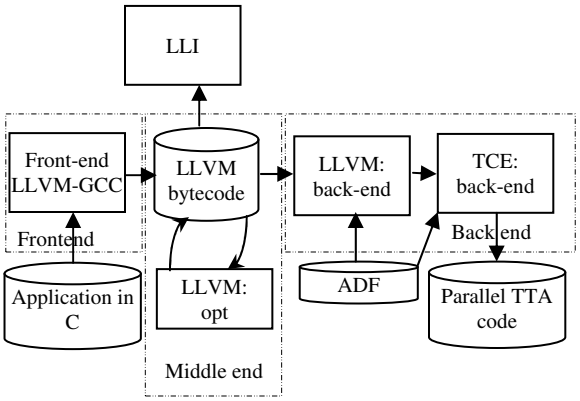
The most influential and demanding part of TCE design flow is code generation and analysis. Figure 41 shows the code generation procedure of the TCE tool. In this stage, the sequential program is converted to parallel instructions by efficiently utilizing the given target architecture. It is very difficult for a programmer to write a thousand lines of a TTA program manually, even if there is a use of semi-automatic design space exploration. Moreover, hand written code is not always efficient. Therefore, in this stage, the scheduler takes all responsibility for the performance of the entire toolset [3]. Figure 42 (a) shows the important concepts regarding an instruction scheduling compiler for the TTA architecture. Generally, the main working principle of a compiler is to translate a program written in a source language to another target language.

In TCE, the compiler is used to translate HLL like C into executable code for TTA. It

should be noted that, during this compilation, it assigns processor resources to every data transport, while avoiding any conflicts in resource usage [5]. Moreover, at the same time, all possible ILP should be exploited to facilitate efficient code execution [5]. Figure 42(a) shows that an ILP compiler has three parts: a front-end, a middle-end, and a back-end [5].



(a)



(b)

Figure 42 : Compiler structure of TCE tool (a) data flow in the ILP compiler [27] (b) structure and data flow in a TCE compiler [27].

The front-end translates the source application code written in HLL into intermediate program representation (IR), and this IR is not compiled for any particular target architecture. All possible auxiliary data, including IR, is the input to the middle-end of compiler (or back-end if there is no optimization performed on IR). The middle-end executes high-level language and architecture-independent optimization on IR produced by the front-end. To increase efficient ILP, this optimization includes dead-code elimination, function inlining, and loop unrolling. In the back-end, the compiler reads machine-independent IR, the architecture description file (ADF), and profiling information. Then it translates the code into parallel code for the target architecture. The back-end performs several optimizations using control analysis, data flow analysis, and memory reference disambiguation analysis. These optimizations comprise register allocation and instruction scheduling, which are important parts of generating efficient code executables for the target processor [27].

Figure 42 (b) shows the basic structure of the TCE compiler, which follows the same configuration of the re-targetable ILP compiler explained in Figure 42(a). The front-end of the TCE compiler is the Low Level Virtual Machine (LLVM) C front-end, which transforms an application written in C to LLVM byte-code. This LLVM byte-code, known as IR, is an architecture-independent intermediate program representation used in the LLVM framework [27]. Then this IR is optimized in the middle-end and simulated with the LLI for verification. The back-end of the TCE compiler requires the architecture definition file of the target processor. In this stage, the LLVM back-end performs machine-dependent code transformations like instruction selection and register selection. After passing this stage, the optimized code contains both machine independent and dependent information. Then this optimized code is applied to the input of the TCE back-end. The back-end performs instruction scheduling, applies TTA specific optimizations, and executes the code generation process. The optimized codes shown in table I(b), for a custom CRC architecture are generated by TCE tool [28].

Table I(b): TCE assembly instructions for CRC implementation with crcfast.adf.

Cycle	Bus 1	Bus 2
1	4 -> ALU.in2,	16777208 -> ALU.in1t.sub ;
2	0 -> CRCFAST.trigger.crcfast,	ALU.out1 -> RF.0 ;
3	gcu.ra -> LSU.in2,	_exit -> gcu.pc.call ;
4	ALU.out1 -> LSU.in1t.stw,	... ;
5	8 -> LSU.in1t.stw,	CRCFAST.output1 ->LSU.in2;
6	...,	... ;
7	0 -> LSU.in2,	4 -> LSU.in1t.stq ;

5.2.5 Program image and Processor Generation

This is the final stage of TCE design flow. This includes generation of HDL files of the selected TTA designs and bit images of the program. Program Image Generator (PIG) processes a scheduled program stored in a TPEF file and generates bit images of the programs that can be uploaded into the instruction memory of the target processor. Figure 43 shows the processor generation technique using TCE tool.

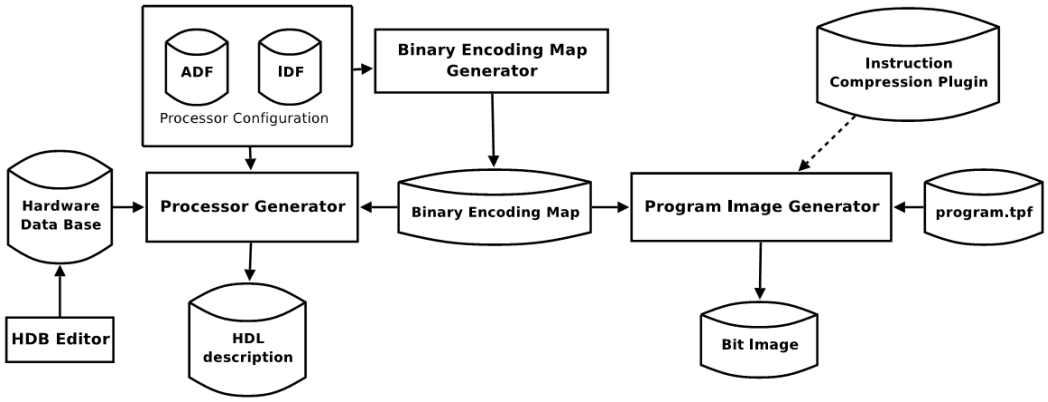


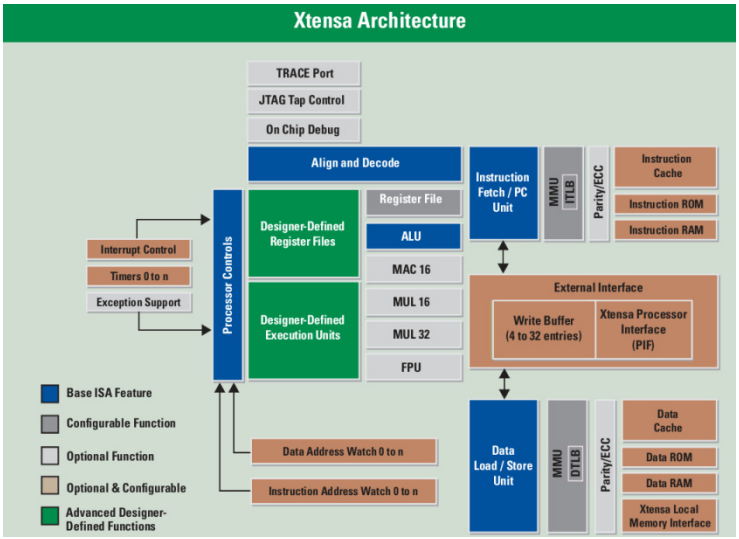
Figure 43 Block diagram of processor generation technique using TCE tool [23].

Program Image Generator (PIG) processes a scheduled program stored in a TPEF file and generates bit images of the programs that can be uploaded into the instruction memory of the target processor. Binary Encoding Map (BEM) can be generated manually or can be obtained by BEM generator. In figure 43, instruction compressed plugins are used to compress the program images and generate a corresponding

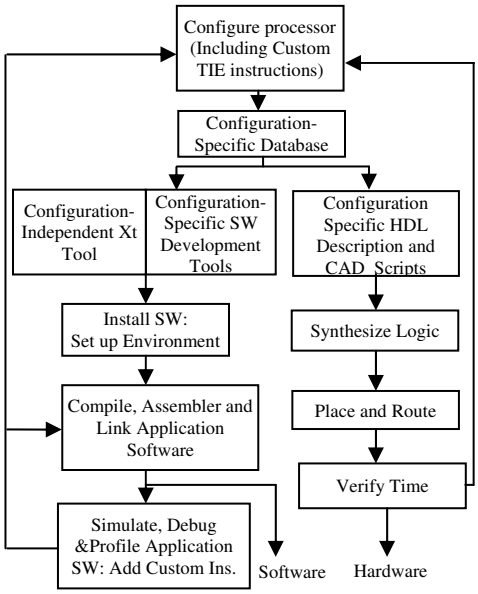
decompression block to the control unit of the target processor. Program Generator reads the ADF and IDF files of the target processor and finally produces the HDL files of the implementations and generates the interconnection network and the control logic by using Hardware Database (HDB) files.

5.3 ASIP Design Flow Using Xtensa Xplorer (XX): Tensilica Tools

Tensilica is very popular in the area of customizable processor design. It was founded by former employees of Silicon valley and EDA companies like MIPS in 1997. Like TCE tool, Tensilica also develops application specific processor for use in synthesized chip design for embedded system. Under Tensilica Xtensa Xplorer is processor IP architecture used to generate processor for input application. Besides the application of TTA-based Co-design Environment (TCE), a comparison between TCE and Tensilica tools is displayed in terms of cycle count. At first, I will discuss an ASIP oriented design flow using Xtensa Xplorer (XX) integrated development environment (IDE) as the design framework under Tensilica tool. Using the XX, it is possible to integrate software development, processor optimization and multiple-processor system-on chip (SoC) architecture into one common platform. From it, we can profile our input application code to identify the cycle consumed by the function used in input design. Then we can make necessary change to speed up that code. There are various building blocks in the Xtensa architecture. Figure 44 (a) shows the structure of Xtensa architecture. This figure shows the range of configurability, extensibility with Xtensa processor. In this architecture, system designer should specify the different blocks of configuration function units. Advanced designer-defined functions are one kind of hardware execution units and registers.



(a)



(b)

Figure 44: Configuration of Xtensa Xplorer
(a) Xtensa architecture [29] (b) Xtensa design Flow.

Figure 44 (b) represents overall design flow of XX. In this figure, the fist block contains different configurations selected upon the nature of input application. Based

on these properties of this architecture, I have taken different configurations of architectures to simulate our input application. For this reason, I have taken 16 preconfigured cores and the result is tabulated after simulating the input application using those cores. Then we apply some custom logic levels to processor for accelerating the processor performance. These preconfigured cores are divided into four broad categories; Communication, HiFi/Audio, Video/Imaging and Diamond or General Purpose Controller. The Communication configuration core is known as ConnX D2 DSP engine. In this thesis, two ConnX configurations known as XRC_D2MR and XRC_D2SA are used for simulation and show very good performance between all other configurations. The XRC_D2xx configuration includes dual 16-bit multiply-accumulate (MAC) units and 40-bit register file to the base RISC architecture of the Xtensa LX processor. This engine uses two-way SIMD (single instruction, multiple data) instructions to provide high performance on vectorizable C code. It implements an improved form of VLIW instructions and five-stage pipeline. Figure 45 shows the basic architecture of the ConnX D2 engine with two MAC units with register banks [30]. The ConnX D2 instruction set is designed for numeric computations like add-subtract, add-compare or add-modulo etc required for digital signal processing. This ConnX D2 core exploits seven DSP-centric addressing scheme mentioned in figure 45. In order to provide excellent performance, it includes data manipulation instructions like shifting, swapping, and logical operations. Our input design is LT codec and it has huge number of shifting, swapping and logical operations. So, this processor architecture is suitable for our input design. Besides this, I have simulated our LT codec design using other configurations. So, I have briefly explained these architecture. For more interest, it is recommended to study the reference manual of Tensilica tool. The HiFi/Audio engine (330HiFi) is optimized for audio processor, voice codecs and pre- and post-processing modules. This configuration includes the Xtensa LX processor that is the basis of the 330HiFi processor. It extends the HiFi 2 Audio Engine ISA for hardware perfecting, 32 x 24

bit multiply/accumulate operations, circular buffer loads and stores and bidirectional shift. There are two main components in this engine: a DSP subsystem that operates primarily on 24-bit data items and other one is a subsystem to assist with bit stream access and variable length encoding and decoding [8].

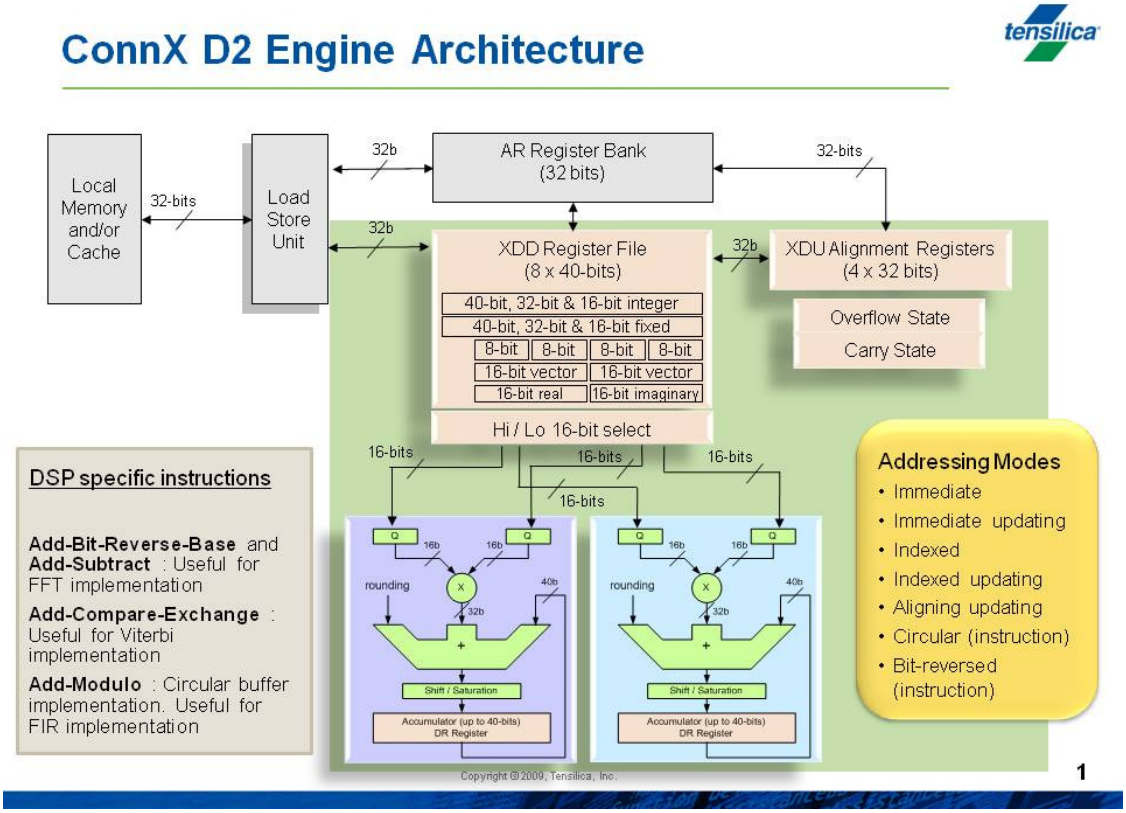


Figure 45: A simplified architecture of ConnXD2 DSP engine [30].

So this architecture is fully compatible for audio/video compression or processing operation. Another category of processor known as Diamond or General Purpose Controllers are optimized for SoC design and it can be used in any application where a controller is required. Diamond controllers are based on a modern RISC architecture.Among these controllers Diamond 106Micro and 108Mini are cache-less controllers and designed for lowest area and power. The Diamond 106Micro has an iterative, multicycle multiplier and uses a non-windowed 16-entry AR register file. So

it is ideal for fast context switching and does better performance for nested function calls. The diamond 108Mini has full 32x32 multiplier and divider and 32-bit input and output general-purpose I/O (GPIO) ports. The Diamond 212GP and 233L are applicable for medium level performance and they have caches, local memories, divider, 32-bit input/output GPIO ports and other DSP instructions. Therefore, Diamond 212GP and 33L are ideal for hard drive controller, imaging, printing, networking etc. The Diamond 570T can generate up to 64-bit Very Long Instruction Word (VLIW) instruction bundles as per the requirement of input design. This VLIW instruction contains two or three operations or instructions. The 570T processor also includes 32-bit input and output GPIO ports with 32-bit input and output FIFO interface. Therefore, this FIFO interface provides a very useful mechanism for the processor to communicate with other RTL blocks, devices and processors [31]. Next, we will show the comparative performance of all these processor architecture.

5.3.1 Extension via TIE

Tensilica Instruction Extension (TIE) is a language that lets designers incorporate application-specific functionality in the processor by adding new instructions. To accelerate the speed of the processor, in Tensilica, it is possible to apply the custom operation in input design. Tensilica Instruction Extension (TIE) language is a powerful way to optimize the processor and is used to describe new instructions, new registers and execution units that are automatically added to the Xtensa processor. Xtensa cores take TIE files as input and create a version of Xtensa processor to complete the tool chain incorporate with new TIE instruction. The processor architect's job is to decide which applications are common enough to warrant some level of support through dedicated instructions.

Figure 46 shows the TIE generation technique using Xtensa processor. This TIE can be generated automatically or manually, depends on the performance of TIE instructions. In this work, we have used TIE instructions generated automatically to

profile our input design and it shows good performance. So using TIE instruction, processor creates single instructions that perform the multiple general purpose instruction

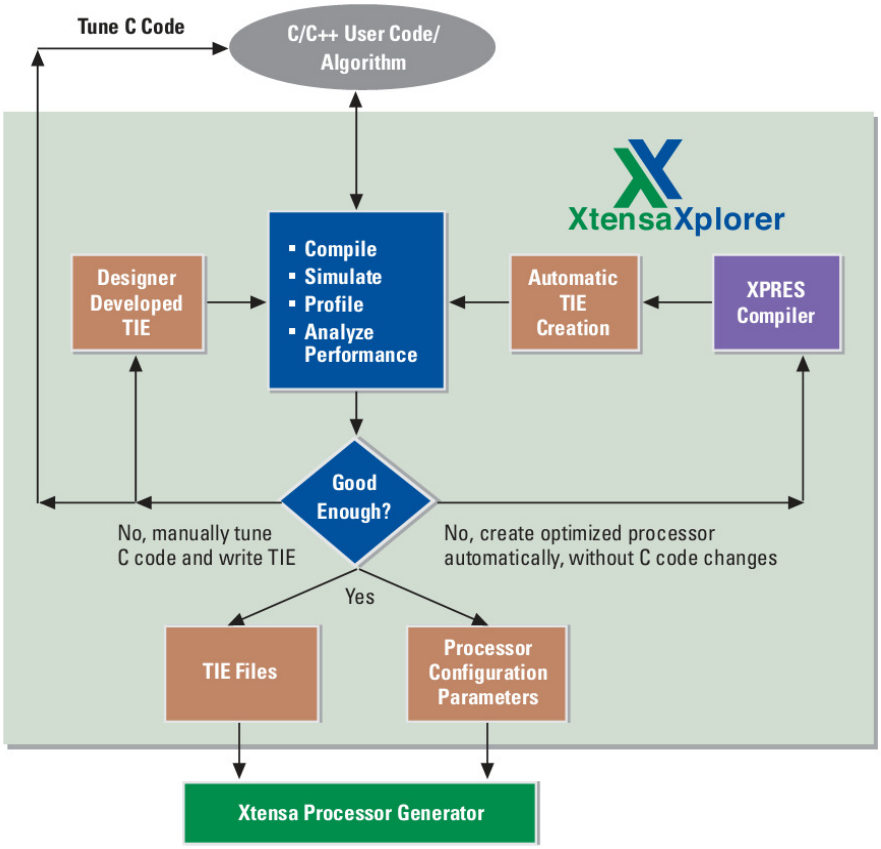


Figure 46: Generation of custom TIE instructions [29].

As mentioned above, TIE instructions improve the execution speed of the input application running on Xtensa processor. Some other techniques like Flexible Instruction Extensions (FLIX), Single Instruction Multiple Data (SIMD) and Fusion can be executable through TIE operation. In this paper, we applied only FLIX instruction to the input application. In Xtensa, FLIX instructions are multi-operation instructions (32-bit or 64-bit long) that allow a processor to perform multiple, simultaneous, independent operations. In FLIX, processors are encoding the multiple operations into a wide instruction word. The XCC compiler takes the FLIX operation

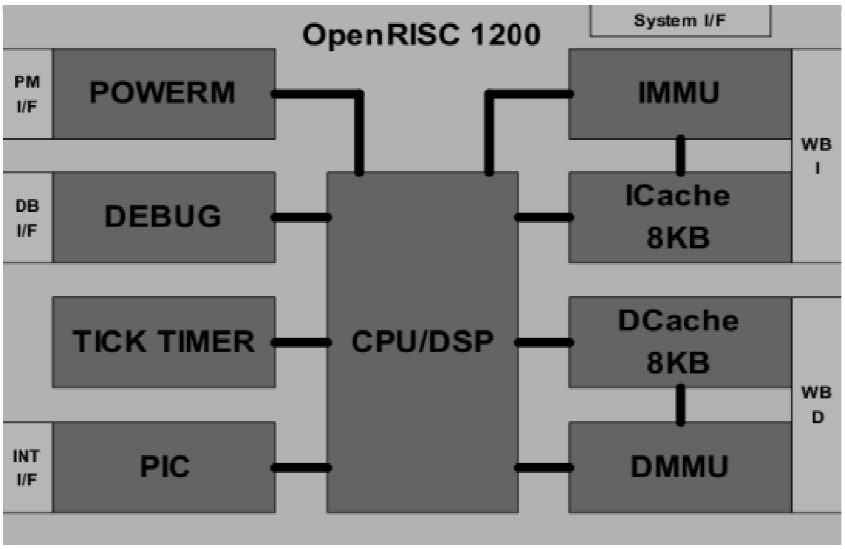
and converts it into FLIX format instruction as per the requirements to accelerate the input code [32]. The performance of FLIX instruction is discussed in simulation result chapter.

5.4 OpenRISC Tool

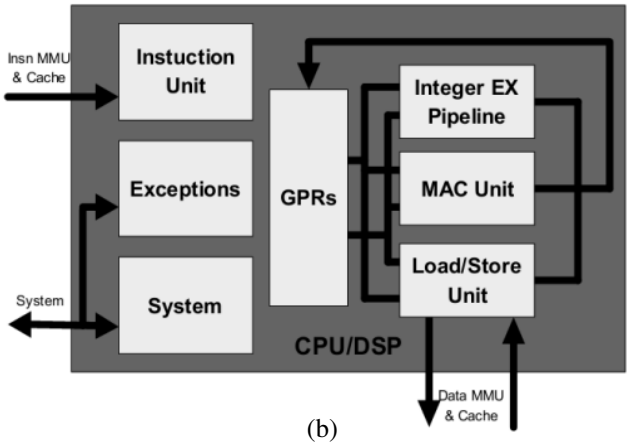
The OpenRISC architecture is one of the latest in the development of modern open architectures. It consists a family of 32- and 64-bit RISC/DSP processors. This kind of architecture allows a spectrum of chip and system implementations at a variety of price/performance points for a range of applications. OpenRISC 1200 is a synthesizable processor developed and managed by OpenCores and using this OR 1200 processor, systems are designed with emphasis on performance, simplicity, low power consumption, scalability, and versatility. It targets medium and high performance networking, portable, embedded, and automotive applications. Therefore, OR 1200 is an open source IP-core available from the OpenCores website as a Verilog HDL model.

By using this tool, the design can be simulated by two ways. The first uses the RTL simulation of primary design by using Icarus Verilog or Mentor Graphic's Modelsim and the second method involves creating a cycle accurate from hardware description language using verilator tool.

In this thesis, RTL simulation (Icarus Verilog Simulator) is donefor reference designs by using OpenRisc architecture, which consists 5-stage single-issue integer pipeline, virtual memory support and basic DSP capabilities [33]. Figure 47(a) shows an overview of OpenRisc 1200 core architecture. For RTL implementation, all the blocks of OpenRisc 1200 IP core are written in Verilog HDL and are published under the GNU License. Here the test programs are compiled to Executable and Linkable Format (ELF) file format, which can be executed both in ISS and RTL simulator.



(a)



(b)

Figure 47: Architecture overview:
(a) OpenRisc core's architecture (b) CPU/DSP block diagram of OpenRisc

This implementation also includes a register set, cache operation, power management unit, programmable interrupt controller (PIC), debug unit and tick timer. Moreover, other peripheral devices can be used by 32-bit Wishbone bus interface. But in this work, we didn't use any peripheral interface. As I mention earlier that the test design by using OpenRisc processor is simulated by RTL simulator (Icarus Verilog simulator) and observed the log files generated as output of that simulator, so this design is not intended for implementation on FPGA prototyping board, rather purely

for making comparison between TTA and OpenRisc processor.

Memory Addressing is one of the important operations of OpenRisc architecture. The processor computes an effective address when memory access instruction is executed. This addressing is also applicable for fetching the next sequential instruction. Fetching instructions from main memory is the main bottleneck of RISC processor. The access time depends on the fetching instructions and this can be alleviated by perfecting instructions before they are required by the processing unit [5]. The memory operand warps around from the maximum effective address and Load/Store instructions using these address mode contain a signed 16-bit immediate value and add to contents of a general purpose register specified in the instruction [34]. OpenRisc 1200 implements 32-bit 32 general-purpose registers (GPRs). The Load/Store Unit (LSU) transfers all data between the GPRs and CPU's internal bus. In figure 47(b), the instruction unit implements the basic instruction pipeline, fetching instructions from memory subsystem, disfetches them to available execution units and maintains a state history to ensure a precise execution model. It implements the 32 bit part of the OpenRISC 1000 architecture. Figure 47 (b) shows the different units of CPU architecture in OpenRISC processor.

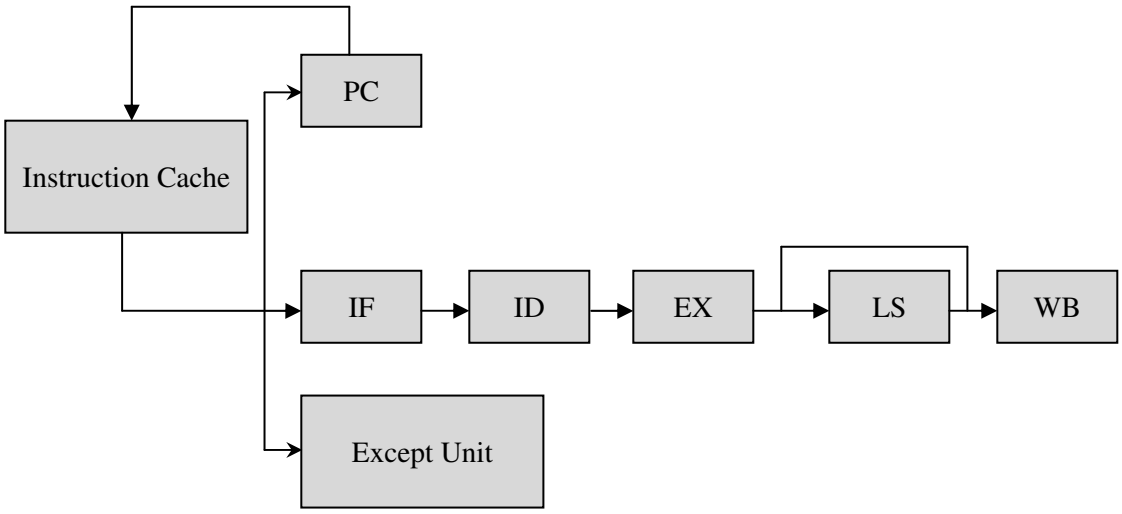


Figure 47 (c): Architecture overview: OpenRISC 1200 5 stages pipeline.

The instruction unit implements the basic instruction sets of the OR1200 core. This instruction unit fetches instruction from the memory system and dispatches them to the available execution units like LSU, ALU, MAC units. The basic operation of instruction unit is similar to that of the RISC processor which is already discussed in the previous chapter. But The OpenRISC1000 architecture defines five instruction's formats and two addressing modes those are explained elaborately in its product ref manual [34]. Besides the GPRs and SPRs, OR 1200 has some important registers like Supervision register, Exception supervision register, Program counter register, exception program counter register and exception effective address registers.

OR 1200 has LSU which is responsible for transferring data between GPRs and the internal data bus of CPU. The LSU has been implemented as as independent unit OR 1200 architecture so that if there is a data dependency then memory system only be affected. The LSU can execute one load instruction every two clock cycles. It has ALU like RISC processor architecture.

MAC unit executes the basic DSP operations and MAC instructions. In OR 1200 MAC unit is fully pipelined. In every clock cycle, it has ability to accept new MAC operation. The MAC instruction has 32-bit operands and a 48-bit accumulator.

System unit connects all the CPU signals to the system signals except those which are connected through the Wishbone interfaces.

The exception unit oversees the exceptions generated by the OR1200 processor core. For example the system calls, memory access conditions, interrupt request etc are handles by the exception units.

For this OpenRisc processor, there are five-stage pipeline named as fetch, decode, execution, memory and write-back [34]. These five instructions are in progress at any given clock cycle and each stage of the pipeline performs its task in parallel with all other stages. So in this thesis, the execution clock cycles are counted for OpenRisc processor by applying two reference designs named as LT encoder and LT decoder architectures. The result will be discussed elaborately in experimental result section.

Figure 47 (c) shows the five stages pipeline architecture of OpenRISC processor. As it is mentioned earlier that pipelining is one of the most important phenomenon to verify the processor. It has strong effort to speed up the processor. Using this Pipelining technique an instruction' execution is divide into a number of independent steps to improve the throughput of a processor. These independent steps are called pipeline stages. Each pipeline stage ends up in a storage (pipeline registers) of its execution so that the subsequent stages can use the result. Therefore the pipelining architecture of OR 1200 processor is similar to the pipelining that I have discussed elaborately in previous chapters.

In this chapter, I have discussed proposed architecture of LT codec, processor design three different ASIP design tools and their architectures. Nevertheless, this discussion is not sufficient for understanding the complete tools. To get adequate information reference manual and user guides of corresponding are recommended. However, there are many other tools, mentioned in earlier chapter for designing and simulating ASIP work. In this thesis, I took only three tools for comparing their results.

Chapter 6

Simulation Result

I have implemented and generated application specific processor for LT codec using TCE, Xtensa and OpenRISC processor design tools. I took TCE as a main designing tool and other two tools have been taken for comparing the results using TCE.

I have translated the complete encoding and decoding algorithm using C program. Before feeding in the decoding module, I apply noise to corrupt the transmitted signal through the channel. Therefore, the overall communication can be modeled by the figure 48.

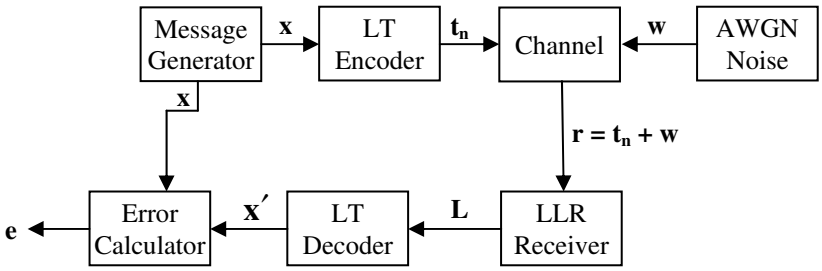


Figure 48: Simulation model of LT codec communication.

The main aim of this thesis is to implement figure 48 using ASIP design tools. The results of this implementation based how efficiently I will produce LT codec processor and its efficiency should be calculated in terms of cycle count and time required for simulation. Area, number of gates and cells required to implement this architecture have been discussed in reference [35].

6.1 LT Codec Simulation Using TCE Tool

Simulation procedures using TCE tool have been discussed elaborately for CRC application in ref [26]. First, we need to compile the input design by TCE C compiler (TCECC). Then, the starting point architecture is required as input for retargetable compiler TCECC. The structure of this ADF depends on the input application system

written in HLL. As it is known that this starting point architecture contains collection of FUs, RFs, Immediate Units (IUs), and transport buses. FUs perform operations, RFs provide temporary fast accessible storage, the network of buses performs data transports between the FU's and RF's, and sockets interface FU's and RF's to transport buses [3]. At first, the minimum structure of architecture known as minimal.adf is used which describes a minimalistic architecture containing minimum resource that TCE compiler can perform to compile C code. So minimal.adf architecture is mandatory architecture and new architectures are formed by adding or modifying custom FU with this minimal.adf architecture. Figure 49 shows the TTA structure of minimal.adf.

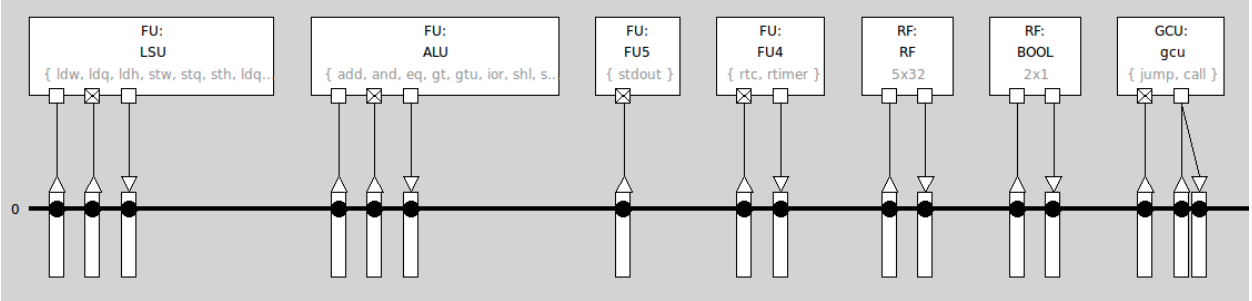


Figure 49 : Structure of minimal.adf architecture.

Instead of copying whole FUs, duplicating the specific operation of that FU will reduce the total cycle count [26]. For this reason, moderate.adf is developed by including its resources with minimal architecture. In order to increase the performance of the processor, new FUs and RFs are added to minimal.adf file and these new architectures are listed in table II. I developed hierarchy of processors for LT codec and its performances are tabulated in terms of cycle counts, time counts and resource utilization. There are various ways to increase the performance of the processor. For example increasing the width of RFs, duplicating the FUs, increasing the number of transport buses, modifying the design architectures and generating the custom FU for specific operation are popular useful techniques for improving the performance of the

processor. However, in this thesis I emphasized on the modification of LT codec input design structure and generating the custom FU for LT codec architecture. Other techniques are explained elaborately in the ref [26]. After finishing the simulation with minimal.adf by using ttasim, the result shows cycle execution counts, time required for simulation and processor utilization which are tabulated in table III.

Table II: Resources of all architecture definition files (ADFs)

Name of ADFs	Resource Name	No.	Description
minimal	LSU	1	FU with operation:ldh,ldhu,ldq,ldqu,ldw,sth,stq,stw
	ALU	1	FU with operation:add,and,eq,gt,gtu,ior,shl,shr,shru,sub,xor
	RF	1	Includes 5x32 bit registers, 1 read and 1 write port
	IO	1	FU with operation: stdout
	TIMER	1	FU with operation: rtc,rtimer
	Boolean RF	1	Includes 2x1 bit registers, 1 read and 1 write port
	GCU	1	Global Control Unit of the Processor
	Transport Bus	1	Fully connected transport bus
moderate	FU_1	2	FU with operation : ldw
	FU_2	2	FU with operation : stw
	FU_3	2	FU with operation : add
	FU_4	1	FU with operation : ldq
Custom	Random	1	FU with operation : random number generator
Encoder	CUS_ENC	1	FU with operation : LT encoding operation
Decoder	DEGREE	1	FU with operation : LT degree distribution, edges information
Decoder_llr	DEGREE	1	FU with operation : LT degree distribution, edges information
	LLR	1	FU with operation : tanh function generation
LT_CODEC	Encoder_Decoder	1	FU with operation : LT encoder and decoder

Therefore, table III shows the implementation result of minimal, moderate and custom architectures of LT codec. From this table, it can be shown that the minimal architecture does not offer good performance. It consumes huge cycle counts and takes more time for simulation. By using this architecture, ADD, LDW and STW consume maximum cycles. Therefore, this architecture can be moderated by

duplicating specific operations like ADD, LDW and STW as separate FUs. A new architecture is formed named as moderate.adf that shows good performance compared to the minimal architecture. This way of improvement is not much explained in this thesis. However, it is discussed earlier that the RNG is very important in this LT encoder and decoder operation. In HLL, default C random function was used to generate this random number. I therefore, generated one new FU name as RANDOM that generates the random number and use this FU in architecture named as custom.adf. Result shows that this custom FU takes only 230 cycle counts and reduces almost 84,900 cycles compared to moderate architecture. Using this custom.adf architecture LT codec takes 195,431,136 cycles and 1,954,311 ms time for implementation. Still this is not sufficient reduction of cycle count for implementing LTcodec. I need to develop more efficient processor.

It is mentioned earlier that there are several ways to improve the performance of the processor. At first I step by step modified the input design of LT codec. For example, the random number generator is widely used in encoder and channel noise generator. If this RNG is included as part of input design then it will consume (84,900/230) almost 370 cycles per function call as compared to RNG is included as part of compiler design (architecture definition file). So it can be easily shown that if there are huge calling of RNG function in HLL then it will consume huge cycle counts. One possible solution of this problem is to design uniform random number generator. But it is very difficult to generate uniform RNG by satisfying the functionality of the encoder and decoder. I have modified the input design depending upon the expectation of random number. For example, in order to generating the degree distribution in encoding part *rand()* is used through its prescribed manner. On the other hand, for noise generation, I have used LUTs instead of RNG.

Similarly it is mentioned earlier that the decoding process of LT codec is based on the iterative manner. Now we need to design a decoder that will take less iteration and this iteration depends on the degree distribution and number of redundant bit to

decode the encoded signal. However in this thesis, satisfying the functionality of LT codec I modify the degree distribution for reducing the cycles and simulation time. Later I will show the design of custom FU for LT decoder. Now I am going to explain the cost statement for different parts of the LT codec.

Before discussing this thing, I will explain the implementation of *printf()* command for printing values using this TCE tool. This implementation is not like the operation of any standard compiler.

Table III: Comparison of cycle counts and resource utilization of LT codec for minimal, moderate and custom ADFs.

Name of Arch(.adf)	Cycle & Time Count	Other Parameters		Operation executed in function units			
	Time (ms) 1958958	Name	Number	Name of FUs	Name of Operations	Number of executions	% of Utilization
minimal		Tran. Bus	1	LSU	LDQ	189945	0.1
	Cycle 195,895,926	Registers in Register Files	16		LDW	11124970	5.7
					STW	11626063	5.9
					STQ	171904	0.1
					LDQU	79160	0
				ALU	ADD	21431038	10.9
					SUB	803811	0.4
					AND	2858751	1.5
					EQ	2754354	1.4
					IOR	360951	0.2
					XOR	2365888	1.2
moderate	Time (ms) 1,955,159	Tran. Bus	1	LSU	STQ	171904	0.09
		Registers in Register Files	16		LDQU	79160	0.04
	Cycle 195,516,036			ALU	SUB	803811	0.41
					AND	2858751	1.46
					EQ	2754354	1.41
					IOR	360951	0.18
					XOR	2365888	1.21
					SHL	4024325	2.06
					SHR	1784612	0.91
				FU_1	LDW	7009118	3.58
				FU_2	STW	7177559	3.67
				FU_1	LDW	4115852	2.11
				FU_2	STW	4448504	2.28
				FU_3	ADD	12295030	6.29
				FU_3	ADD	9136008	4.67
				FU_4	LDQ	189945	0.1
custom	Time (ms) 1,954,311	Tran. Bus	1	RANDOM	RAND	230	0.000118
	Cycle 195,431,136	RFs	16				

Since TCE is operating system free platform so, *printf ()* implementation does not follow the straightforward approach. To make this job easy it is required to include an

operation name as STDOUT. This operation reads its input from the bus connected to the architecture, expected it to be a 8-bit char and writes the char verbatim to the simulator host's standard output. Software floating point support is necessary (*swfp* flag) because this tells the compiler to link the program with the floating point emulation library. Therefore, *printf()* function includes support for printing floating point values and our architecture does not contain floating point function units. This operation consumes huge cycles that is shown in the next simulation results. After verifying the system, this FU should be removed completely.

Table IV shows the simulation result for the LT encoder using different architectures. According to the previous discussion for using the printf command the Encoder processor takes more than 1,583,000 cycles compared to the without printf processor operation. However, at first I have simulated the LT encoder using minimal.adf architecture and it takes huge time and cycles because of missing custom FU. Then in Encoder.adf I have included one custom FU named as CUS_ENC to transfer the major operation of encoding algorithm to the compiler part (hardware architecture). From table IV it can be shown that the this custom operation takes only 230 operations and reduces the clock cycles almost 7,717,027. This shows the significant improvement in performance.

Table IV: Comparison of cycle counts and resource utilization of LT encoder for Encoder and minimal.ADFs

Name of Arch(.adf)	Cycle Count	Time Count (ms)	Operation executed in function units			
Encoder	23,946 (Without Print Operation)	238	Name of FUs	Name of Operations	Number of executions	% of Utilization
			CUS_ENC	CUS_ENC	230	1
			ALU	ADD	2499	10.5
Encoder	1,606,946 (With Print Operation)	16068	FU_3	CUS_ENC	230	0.01
			ALU	ADD	202425	12.5
minimal	7,740,973	77409	ALU	ADD	792444	10.23

After generating the processor for LT encoder, I will generate efficient LT decoder processors. In table III it is shown that for minimal.adf architecture LT codec takes highest number of cycles and from the theory of LT decoding algorithm, decoding of

LT codec is very much complex compared to encoder algorithm. Before designing the custom FU for implementation of LT decoder, I will explain the main bottleneck of decoding algorithm. In decoding algorithm, soft decoding procedure has been used through the check node and variable node operations. So VNU operation, it requires to know that how many edges are formed for each variable node that means it will tell the degree distribution of the message signal. Similarly, in CNU operation it will require to know that how many variable nodes are connected with each check node. That means the edge information of the check nodes. It is mandatory to find the single edge check node (degree 1 value of check node per update), so it is required to index the edges of the check nodes. Figure 50 shows the pictorial information of this decoding scenario. So to make the custom FU for LT decoder, I need to include these three information to this custom FU and use the required output properly fetching from this custom FU. The name of this custom FU is DEGREE. Moreover, in the decoding end, the encoded signal should be taken from the DEGREE FU.

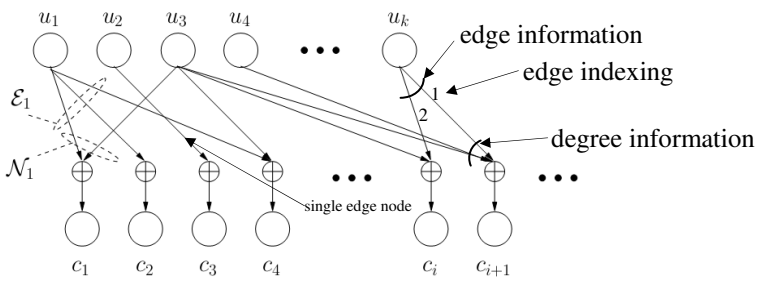


Figure 50: LT codec tanner graph for understanding the algorithm of LT decoder.

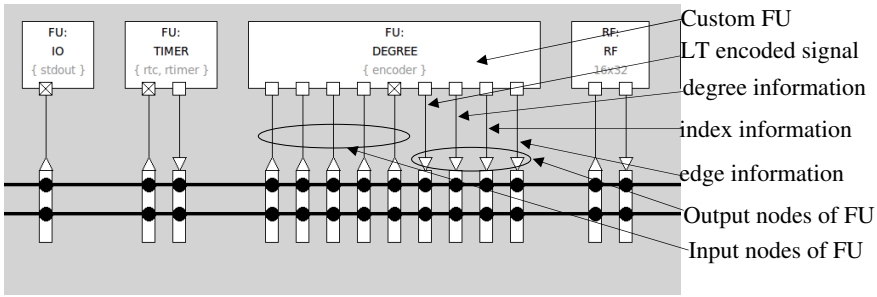


Figure 51: Architecture of custom function unit (DEGREE) for LT decoding application.

Figure 51 represents the structure of custom FU DEGREE and this FU is for decoding algorithm of LT codec. This DEGREE function unit gives four outputs those are labeled in the figure 51. Now degree, edge and index information generations are the part of compiler that means architecture through this FU. As a result the new ADF file Decoder will take less cycle counts for implementing the decoding operation and in this ADF architecture, whole encoding operations: generation of encoded signal, degree, index and edge information are part of the DEGREE FU. So I can remove the coding related to activities of DEGREE FU from the main input design written in C language. The custom FU DEGREE is written in C++ programming language. So, this is a powerful technique used in TCE tool. Table V shows the simulation result of LT decoder using Decoder ADF. Result shows that Decoder.adf configuration takes 184,541,996 cycles which is less than 10,889,140 cycles compared to the result of custom.adf architecture. From table V, it can be shown that DEGREE FU takes only 358 cycles when its operations are as a part of ADF architecture. Behind this operation the processor improves its efficiency by reducing the 10,889,140 cycles compared to custom.adf. Still, it is not sufficient in terms of cycle reduction. Therefore, I need to modify more.

Table V: Cycle counts and resource utilization of LT decoder for Decoder ADF

Name of Arch(.adf)	Cycle Count	Time Count (ms)	Operation executed in function units			
			Name of FUs	Name of Operations	Number of executions	% of Utilization
Decoder	184,554,925 (With Print Operation)	1,845,549	DEGREE	DEGREE	358	~0
			ALU	ADD	18171801	10
			LSU	LDW	9437612	5.1
Decoder	184,541,996	1,845,418	FU_3	DEGREE	358	~0
			ALU	ADD	2433650	1.3
			LSU	LDW	170573	0.1

According to the sum product algorithm, in CNU and VNU operation 'tanh' is used for sign identification. Therefore, I make a custom FU for 'tanh' function which is included in the architecture named as Decoder_llr. Table VI shows the result of this processor. From the comparison of table V and VI, LLR custom FU reduces the 163,425,299 compared to Decoder.adf processor. LLR itself consumes only 1380 cycles.

Table VI: Cycle counts and resource utilization of LT decoder for Decoder_llr ADF

Name of Arch(.adf)	Cycle Count	Time Count (ms)	Operation executed in function units			
			Name of FUs	Name of Operations	Number of executions	% of Utilization
Decoder_llr	21,129,626 (With Print Operation)	211,296	FU_3	DEGREE	358	0.001
			LLR	LLR	1380	0.006
			ALU	ADD	2185800	10.3
			LSU	LDW	1123666	5.3
Decoder_llr	21,116,697	211,166	FU_3	DEGREE	358	0.001
			LLR	LLR	1380	0.006
			ALU	ADD	2184203	10.3
			LSU	LDW	1122914	5.3

Yet, it is not sufficient the status of cycle count. According to table VI, it takes more that 21M cycles. But I want to reduce cycle count more. If I analysis the decoding part of input design, the whole complexity of decoding algorithm drops to the number of iterations of the message passing algorithm. Moreover, this number of iterations depends on the degree distribution of encoded signal. For constant degree distribution, error (**e**) of figure 48 is inversely proportional to the number of iterations. I actually in this thesis, focused on the implementation of the encoder and the decoding so, I

slightly modify the degree distribution to ensure the error (**e**) is zero and calculate the cycle count w.r.t. number of iteration. Table VII shows the result of this analysis. Here I have used the same architecture Decoder_llr and mimimal.adf that for simulating with different iteration number. Table VII shows the comparative result between two different architectures. For example for 7 iterations, minimal.adf took huge cycles because of input design. In this input design, I have included channel noise and there is no optimization of degree distribution. Moreover, the minimal.adf architecture is a simple processor structure.

Table VII: Comparison of cycle counts of LT decoder using two ADFs for different iterations.

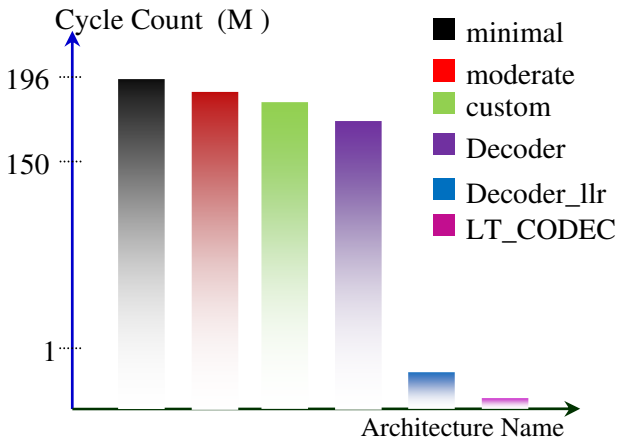
	Decoder_llr.adf architecture		minimal.adf architecture	
# of Iterations	Cycle Count	Time Count (ms)	Cycle Count	Time Count (ms)
1	6,581,637	65,816	168,851,862	1,688,518
2	9,488,649	94,886	532,749,722	5,327,497
3	12,395,661	123,956	943,238,922	9,432,389
4	15,302,673	153,026	1,378,119,387	13,781,193
5	18,209,685	182,096	1,835,020,463	18,350,204
6	21,116,697	211,166	2,306,948,480	23,069,484
7	24,023,709	240,237	2,789,089,682	27,890,896

Up to this point, Decoder_llr architecture takes minimum cycles to process the LT decoder. This architecture can be further modified by generating a custom FU using Encoder.adf and Decoder_llr.adf architectures. The name of this FU is Encoder_Decoder. Using this FU the final architecture is formed as LT_CODEC.adf. Table VIII shows the final result using this architecture. It takes very less cycle counts compared to all other architectures. When an operation is included as function of input design, it will take more cycles to generate the TTA instructions for this particular operation. TTA compiler will translate this specific operation instructions by using ALU and LSU FUs. On the other hand, when the specific operation is included as a part of custom FU then the TCE compiler can easily generate the TTA instructions independently. This is explained in the code generation technique of TCE tool [28]. However, figure 52 show the complete scenario of all architectures. After designing

this architecture, TCE will generate the complete processor for specific application input design in VHDL HDL.

Table VIII: Cycle counts and resource utilization of LT decoder for LT_CODEC.adf

Name of Arch(.adf)	Cycle Count	Time Count (ms)	Operation executed in function units			
			Name of FUs	Name of Operations	Number of executions	% of Utilization
LT_CODEC	4,466	43	Encoder_Decoder	Encoding &Decoding	1	0.02
			ALU	ADD	666	15
			LSU	LDW	305	7



Architecture Name	Cycle Count
minimal	195,895,926
moderate	195,516,036
custom	195,431,136
Decoder	184,541,996
Decoder_llr	21,116,697
LT_CODEC	4,466

Figure 52: Comparative performance of different architectures for LTcodec implementation.

These are the step by step procedures for generating the application specific processor like LT codec. According to the performance of the processor, LT_OCDEC processor shows very good performance compared to the other architectures. Moreover, these architectures can be further modified by duplicating the custom FUs, adding more data BUS or changing the RFs. However, after generating the optimized processor as HDL formation, it will be applied in prototyping board, or chip design procedures for getting the real information about timing, area or power reports. In the next section, I will discuss the simulation result using Tensilica tool.

6.2 SimulationResult Using Tensilica Tool

To compile an application in XX, we required to inform Xplorer the project to compile the processor configuration to compile the project on and the build target. A set of build properties like compiler, assembler and linker contains in a build target. In this work, we took the “release” version of the target library using level 3 optimization and apply FLIX & TIE instructions. Figure 53 shows the configuration overview of the ltcodec_tie processor configuration. From figure 53, this processor is developed using TIE instruction set for LT codec input design and then add this TIE instruction with core processor named as XRC_D2SA.

Now I am compiling the LTcodec input design as reference code along with its library for each of the sixteen target cores and then run a profile execution.

Configuration Overview	
User Name	chosun_ice_edu/sub2
Core Name	ltcodec_tie
Core Description	XRC_D2SA
Configuration Detail	
TIE sources for configuration	ltcodec.tdb contains ltcodec.tie
Xtensa ISA version	LX4.0
Instruction options	
16-bit MAC with 40 bit Accumulator	no
MUL 32	no
32 bit integer divider	no
Single Precision FP	no
Double Precision FP Accelerator	no
Synchronize instruction	no
Conditional store synchronize instruction	no
MUL 16	yes
CLAMPS	yes
NSA/NSAU	yes
MIN/MAX and MINU/MAXU	yes
SEXT	yes
Boolean Registers	yes
Number of Coprocessor(NCP)	3
Enable Density Instruction	yes
Enable Processor ID	yes
Zero-overhead loop instruction	yes
TIE arbitrary byte enables	yes

Figure 53: Processor configuration of ltcodec_tie architecture

Table IX: Comparison of cycle counts for different configurations of Tensilca tool.

Active Processor Configuration	Total cycles	Required Time (s)
DC_C_106micro	229,213,917	163.71
DC_C_108mini	219,797,553	171.82
DC_C_212GP	204,964,527	164.23
DC_C_233L	204,968,307	170.19
DC_C_330HiFi	202,604,066	165.85
DC_C_545CK	201,013,597	180.24
DC_C_570T	170,170,153	153.52
DC_D_106micro	229,213,920	162.71
DC_D_108mini	219,797,557	170.46
DC_D_212GP	204,964,531	163.87
DC_D_233L	204,968,281	169.39
DC_D_330HiFi	202,604,071	166.68
DC_D_545CK	202,604,071	179.79
DC_D_570T	170,170,158	154.26
XRC_D2MR	164,231,379	137.86
XRC_D2MR_FLIX	162,629,766	135.66
XRC_D2SA	208,465,165	157.37
XRC_D2SA_FLIX	206,444,710	202.20

Table IX represents the comparison of cycle counts for all processor configurations. As shown in figure 53, the configuration components are designed according to the implementation of input design. Based on this, ConnXD2 category processor shows very good result compared to the other processor configurations. If we study the cycle consumed by different operations using TCE tool, there are huge addition and logical operations taken by the LT codec design. Due to this reason, ConnXD2 type processor is suitable for simulating this LT encoder and decoder. From table IX, We can see that, without custom instruction operation XRC_D2MR is the best in comparison to other processors. Moreover, in Diamond controller processor, 570T configuration outperforms compare to others. We see that, 570T processor contains many DSP instruction extensions and SIMD execution units. If we see the disassembly information of input function, it is easily possible to find the step-by-step cycle consumptions by main and children functions as per their configuration details. We are not going to discuss all these architectural analysis. As it is mentioned earlier that ConnX D2 architecture is suitable for communication and for its rich hardware

resources, XRC_D2MR configuration without TIE or FLIX instruction, takes 164,231,379 total cycles for LT codec application. From its profile status, *main* function consumes highest 7,585,908 cycles and if we see the disassembly profile of *main* function, it takes many load, add, move and logical operations. So, when we think in terms of hardware, these operations are rewiring certain bits from input to output. For this reason, we develop TIE and FLIX instructions and include these custom instructions to the processor. Table IX shows the result of all target processor in terms of cycles. Significant improvement in terms of cycle counts was found and from this table, the XRC_D2MR_FLIX configuration took 162,629,766 cycles and *main* function took only 5,984,295 cycles which reduces 1,601,613 cycles compared to without FLIX operation. These architectures can be further modified by introducing the custom TIE instructions. I have generated TIE instruction by using automatic TIE generation techniques as mentioned in Figure 46. Now I will show the behavior of iteration vs cycle counts of LT codec implementation.

Table X: Simulation for different number of iteration using Tensilica tool

# of Iterations	XRC_D2MR_MAC		DC_C_106micro	
	Cycle Count	Time Count (s)	Cycle Count	Time Count (s)
1	5,204,861	4.43	6,983,593	4.97
2	19,182,518	15.60	26,034,519	18.31
3	35,128,884	27.71	48,036,337	33.79
4	52,840,982	42.21	72,696,444	50.98
5	71,951,305	57.18	99,135,487	69.74
6	92,160,200	74.41	127,678,043	90.72
7	115,082,566	92.73	159,731,114	113.43
8	164,837,807	128.70	194,314,345	137.03
10	189,915,708	151.39	264,271,320	185.73
16	340,940,055	283.37	475,626,960	335.57

It is mentioned earlier that the decoding complexities depend on the number of iterations required for recovering message from encoded signal. For XX it takes 9 iterations for successfully decoding the encoded signal. However, it is possible to reduce the number of required iterations by modifying the degree distribution in the encoder. Table X was simulated for fixed degree distributions using highest and

lowest configurations. Therefore, number of cycles are increasing exponentially with respect to the number of iterations. So it is very important to trade off between several issues: degree distribution, architecture structure of processor configuration, architecture of LT encoder and decoder, and finally the status of BEC. Because, the value of δ depends on the characteristics of the channel and the average number of degree connected with variable node depends on the value of δ . For example, according to the table X, for fixed value of δ , XRC_D2MR takes more that 340M cycles for 16 iterations on the other hand for diamond controller 106mico, it takes more than 475M cycles for 16 iterations. Moreover, simulating time behaves same as the manner of cycle counts.

6.3 Simulation Result Using OpenRisc Tool

For OpenRisc processor, *".cfg"* file contains the default configurations and a set of simulation environments, which are similar to the actual hardware situation. For RTL simulator, the verilog files of all IP cores are included by using MAKE file. So once the environment is configured then the simulator generated the *".log"* files under *"out"* and *"run"* folder. The minimal architecture of reference design is shown in table XI. In the OpenRisc processor, the reference design is compiled using OpenRisc tool chain (*or32-elf*) and a memory image is generated (*.vmem*). Then this program image is used in simulation to fill the RAM. Next, the verilog RTL sources check, compile, and simulate the result. Therefore, the OpenRISC processor will generate all the required signals to execute the operation.

There is no GUI for processor configuration in OpenRisc tool. So this reference design can be modified by setting the enable value 0/1 in the configuration file. For example in DMMU, *entry size* means the instruction size in bytes, the typical value of *entry size* is 64. *SIM* section of this configuration is one of the major parts in this configuration. This section specifies the behavior of the *or1ksim* processor. Under this section, it includes the operations like *verbose* used for printing extra message, *debug*

used for debugging, *profile*, *mprofile* used for memory profiling, *exe_log* etc. Similarly, CPU section ensures the operations like *ver* used for version, *sr* used for supervision register, *sbuf_len* used for length of store buffer etc. *PM* section is used for power management. UART section is used for creating an interactive terminal window like xterminal window. By setting or modifying the above parameters, new processor can be designed through observing their performances.

Table XI: Resources of OpenRisc processor for reference design

Resource Name	No.	Description	Processor
IMMU	1	Instruction Memory management Unit	OpenRisc
DMMU	1	Data Memory Management Unit	
IC	1	Instruction Cache	
DC	1	Data Cache	
CPU	1	Central Processing Unit	
PM	1	Power Management	
UART	1	Universal Asynchronous Receiver/Transmitter	

Table XII: Simulation result by using OpenRisc processor encoder and decoder.

OpenRISC Processor			
Encoder		Decoder	
cycle	Time (ns)	cycle	Time (ns)
142,015	6,174,570	153,353	6,712,850

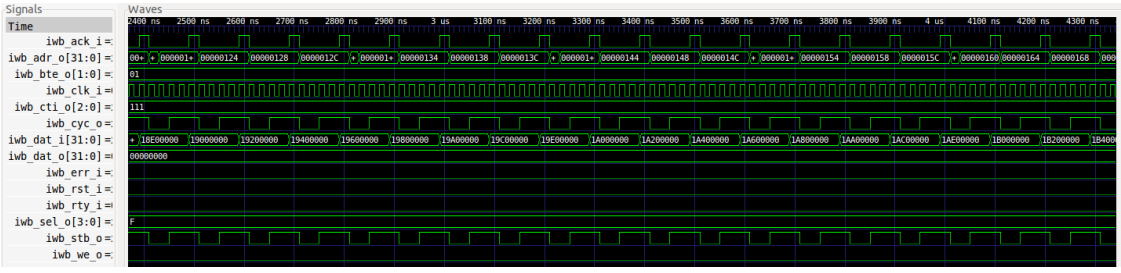


Figure 54: Different signal waveforms of instruction wishbone bus for OpenRisc-1200 core.

As it is mentioned, earlier that custom operation or instruction generation is one of the powerful techniques to reduce the cycle count. In OpenRisc processor tool, I did not find such option like designing custom FU in TCE or TIE and FLIX instruction generation technique in Tensilica tool. Therefore, in OpenRisc tool only modifying

the CPU configuration is not sufficient to reduce the cycle count. For implementing the technique of sum product algorithm, it is required to use the *sign* function (*tanh* or $tanh^{-1}$) in LT decoding algorithm. In OpenRisc C compiler it does not support to include the "*math.h*" header file. Therefore, I modify the decoding architecture of LTcodec design as per requirements of OR C compiler by including the LUTs. But these LUTs are not efficient because of random number generator. For each new simulation this LUT should be changed due to change of RNG. But for implementing the LT encoder, it does not require any mathematical operation. So it is easily synthesized by OpenRisc core. However table XII represents the simulation result using this processor. Here I have simulated encoder and decoder independently due to the missing support of *math.h* header file. While simulation, by enabling the option *VCD* = 1, value change dump (VCD) file had been generated under '*out*' folder. Then 'signal.wav' file has been loaded and we can see the output waveform of OpenRISC processor instruction wishbone bus by GTK wave tool using "*or1200-lttest.vcd*" file. It is also possible to get the wave form of other signals like uart, ram, data wishbone bus etc. Figure 54 shows different signal of instruction wishbone bus for OpenRisc-1200 core. In this figure iwb_clk_i means instruction_wishbone_clock_input signal. Similarly, iwb_ack_i means acknowledgement signal.

There are some limitations for simulating LT codec design using OpenRisc processor. I successfully completed the implementation of encoder but in the decoding part implementation didn't work properly. Due to the problem of header file, I mentioned earlier that LUTs had been used there and these LUTs have been changed in each simulation because of random degree distribution. So it is not possible to calculate the error calculation of the LT codec. Since there is no option to transfer load from input design to compiler or simulator, so it is not possible to include the custom FU like TCE or custom instruction set like TIE and FLIX in OpenRisc processor. Only modification can be achievable by designing LTcodec architecture as input design or changing the CPU architecture of the processor. But the effect of changing

CPU or simulation architecture shows very less impact on cycles count or simulation time. For this reason I didn't represent the CPU architecture modification in this thesis, although I have done this by changing setting the enable condition of different parameters under CPU section in reference configuration.

6.4 Comparison between All LT Codec Processsors

Now, it is necessary to mention that we already developed hierarchy of different architectures for LT codec by using TCE, Tensilica and OpenRisc tools. First, I will show the comparison between TCE and Tensilica tool for LT codec implementation. Then, the comparison between TCE, Tensilica and OpenRisc will be displayed. Table XIII shows the comparison between TCE and Tensilica processor.

Table XIII compares the performance results of Xtensa Xplorer and TCE tools. While simulating the instruction set simulator of TCE, tool run time count (RTC) is measured in millisecond and clock frequency is 100 MHz. From this table LT_CODEC.adf architecture takes minimum cycles compared to other architectures of TCE and Tensilica tools for implementing LT encoder and decoder. Moreover, this architecture took only 43 ms which is very less compared to the Tensilica tool. If we analysis the structure of XX core, it satisfies the class of RISC processor including the five and seven stage pipeline design. In this design, five-stage pipelining had been used for implementation. On the other hand, TCE tool is for implementing input design on TTA. It is mentioned earlier that the TTA structure has more benefits compared to the OTA processor domain. In OTA domain, it takes separate instructions for executing the instructions using ISS environment. For this reason Xtensa tool takes more cycles for implementation. However, the simulation speed is very high compared to the TCE tool. From table XIII, it can easily be calculated that TCE executes almost 100 K cycles per second using 100 MHz clock. However, Tensilica runs 1 M cycles per second using ConnX D2 engine. Now to make a fair comparison with three tools I have simulated encoder part of LT codec by using these

tools. Table XIV shows this comparison result, which is responsible for getting the scenario of these three tools.

Table XIII: Comparison of cycle counts for the TCE and Tensilica processors

TCE			Tensilica		
Architecture Name	Time(ms)	Cycle Count	Cycle Count	Time(s)	Architecture Name
custom	1,954,311	195,431,136	204,968,307	170.19	DC_C_233L
Decoder	1,845,419	184,541,996	202,604,071	179.79	DC_D_545CK
Decoder_llr	211,166	21,116,697	170,170,158	154.26	DC_D_570T
LT_CODEC	43	4,466	162,629,766	135.66	XRC_D2MR_FLIX

Table XIV: Comparison of cycle counts for the TCE, Tensilica and OpenRISC processors

TCE			Tensilica			OpenRISC	
Architecture Name	Time(ms)	Cycle Count	Cycle Count	Time(s)	Architecture Name	Cycle	Time (ns)
Encoder	238	23,946	142,557	0.11	XRC_D2MR	142,015	6,174,570
minimal	77,409	7,740,973	212,886	0.20	DC_D_570T		

From this table, it can be shown that Tensilica tool takes more cycles than others and the performance of the TCE is very good compared to others. Like Tensilica, OpenRisc takes separate cycles for executing the instructions, which is a common behavior of the OTA class processor tools. On the other hand, for TCE tool it is occurred as the side effect of data transport. However, all the architectures of these three tools can be further modified by using their own techniques. Besides this, the modifications of LT codec i.e. degree distribution, number of decoding iteration, or input and encoded message length have huge influence on this hardware throughput. Within these three tools, Tensilica tool is very easy in terms of use and optimization. In this thesis, I have used the Optimization level 3, automatic TIE and FLIX options of Tensilica tool. Moreover, the modification of configuration parameters of XX is not sufficient for designing the high performance LT codec design. Similarly, for TCE tool still, it can be modified by introducing more buses (presently I have used 9 buses), duplicating FUs, RFs and adding more efficient custom FUs etc. Therefore, an efficient trade off is required between all these observations to satisfy an excellent processor based on the input application.

Chapter 7

Conclusions

The step by step techniques of application specific processor design using TCE, Tensilica and OpenRISC tools have been discussed elaborately in the previous chapters. Finally in the result chapter, the comparisons of these three tools are presented in different aspects. In this chapter, the summary of whole thesis and some future ideas will be presented for extending of this thesis work.

7.1 Summary

The whole work of thesis can be divided into three parts: efficient processor selection, state of the art input design selection and finally generation of processor for that input design. Figure 55 shows the pictorial presentation of this thesis activity. This figure represents the algorithmic architecture for LT codec ASIP implementation. Accodring to the figure 55, processor platform selection is an important block in this design flow. For that reason, in the first couple of chapters I have discussed what kind of processor we should select. For explain this thing, I have started from RISC class processor and tried to explain the development of other efficient processor by removing the step by step shortcomings of RISC, Superpilelined and finally VLIW processor. Therefore, for designing application specific system TTA is promising processor family for getting high speed response. After selecting the efficient processor class, for generating application specific processor, we required to take a state-of-the-art system as input design. Nowadays the fountain code is very promising in the area of channel coding. Under this fountain code class we have selected LT codec channel coding technique compatible for BEC. Many researchers are interested due to comparatively simple and efficient manner of LT codec. Although due to some problems of LT codec some other fountain codes like Raptor code has been developed.

However, in this thesis we have elaborately discussed regarding the implementation issues of this LT encoder and decoder.

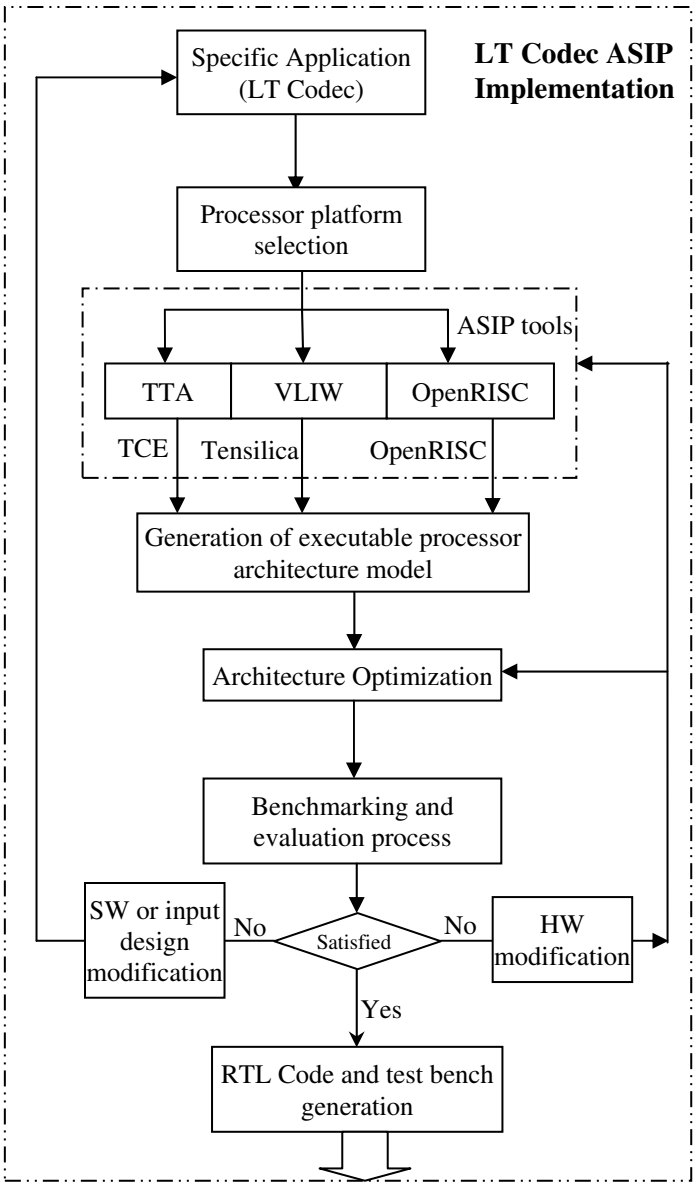


Figure 55: Design Flow of this thesis work.

For processor generation, we took three tools like TCE, Tensilica and OpenRISC. TCE is working for developing the TTA based processor. OpenRISC tool is executing under the concept of pure pipelined RISC processor. On the other hand, XX of

Tensilica shows the behavior like VLIW processor. As we discussed earlier that TTA is very suitable for applying custom FU to the architecture. Therefore, I have designed different custom FU for LT encoder and decoder. Similarly, in Tensilica tool, the processor configuration can be modified as per the input application in various ways. In this thesis, TIE and FLIX technique are applied to improve the performance of processor in terms of cycle count. Finally the performance of the OpenRISC processor has been studied. I find some limitations while using the OpenRISC processor. For simulating the input design written in HLL, it does not support many of the header files. As a result, there should take some alternative solutions like LUTs or other functional program based on mathematical operations for generating the processor. However, the response of the processor are not solely depends on the processor architecture. This performance also depends on the input design architecture. Therefore, besides the designing of custom processor part, we need to design LT codec as a reference input efficiently. We have discussed this proposed design technique in chapter 5. In this thesis work, there are some observations I find during simulation time. There are many reconfigurable techniques for every tool. It is not possible to take all these optimization techniques. For example, I have used the Optimization level 3, automatic TIE and FLIX options for Tensilica tool. Moreover, the modification of configuration parameters of XX is not sufficient for designing the high performance LT codec design. Similarly, for TCE tool still, it can be modified by introducing more buses (presently I have used 9 buses), duplicating FUs, RFs and adding more efficient custom FU etc. Therefore, an efficient trade off is required between all these observations to satisfy an excellent processor based on the input application.

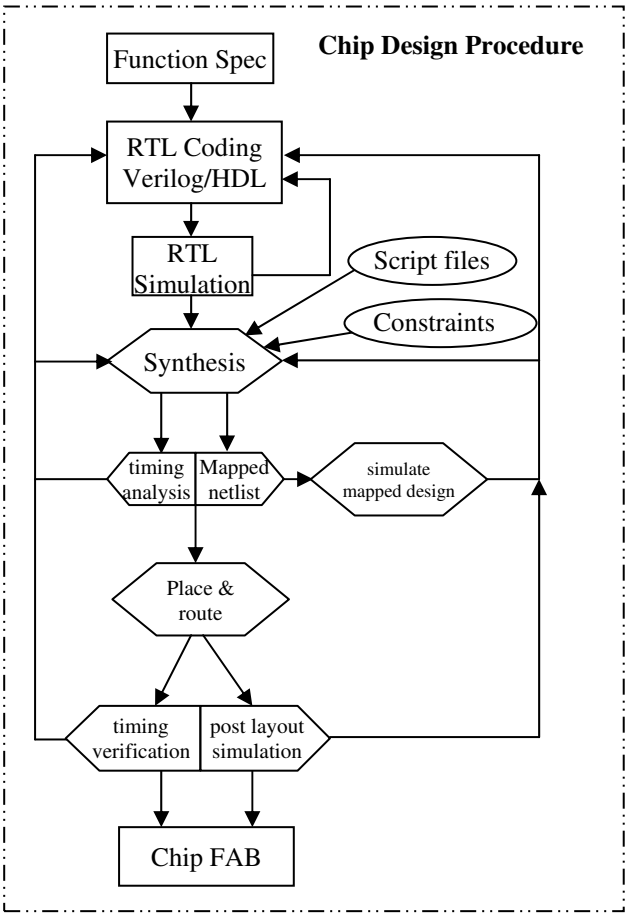


Figure 56: Design Flow of Chip design procedure.

7.2 Future Work

Currently I have used only three tools for getting the application specific processor of LT codec. In addition, from this comparison I found that this LT codec processor by TCE tool is good in terms of cycle count and required time. However, some other efficient tools like LISA, Coware etc. are required to make comparison with this current one. To make the LT codec processor efficient, it is also required to apply more optimization on the degree distribution of the LT encoder. It is already explained that, the whole complexity of LT codec depends on this degree distribution i.e. the maximum degree value in encoding part. Based on it, in the decoding part it requires

more iteration. Therefore, it is also be a part of future work to generate more efficient degree distribution.

The processors I have generated by using these three tools are not the ultimate goal of System on Chip (SoC) design. The first part of SoC (system design) has been done through this thesis work presented in figure 55. The second part of SoC (Chip design) has been remaining as shown in figure 56. Although I have done chip design procedures for LT codec but I did not use the RTL code generated from ASIP tools. Therefore, as mentioned in figure 55, at the end of this design flow, the target processors are generated in HDL form and it should be the input of figure 56. That means, the next step is to apply this HDL module into the chip design procedure. After checking the logic simulation, these modules should be synthesized by Synopsys or other tools using TSMC or Samsung DB files. Then the real scenario in term of area, power and time (although these parameters are also shown in the ASIP tool but those reports are not practical) will be found and finally we come to know which processor is very suitable for commercial use.

Appendix I

Architecture and Bypass Complexity

The definitions of Connectivity Graph are written below [4]:

A1. The connectivity graph CG of a processor is a bipartite graph CG (S_n, D_n, E), where:

1. $S_n = \{S_{n_1}, S_{n_2}, ..., S_{n_m}\}$ is a set of source nodes. All producers of values are treated as source nodes.
2. $D_n = \{D_{n_1}, D_{n_2}, ..., D_{n_n}\}$ is a set of destination nodes. This node is considered as consumers of operand and result values.
3. $E \subset S_n \times D_n$ is a set of directed edges.

A2. The architectural connectivity complexity, AC_{compl} of a connectivity graph CG (S,D,E), is defined as a (#S,#D,#E), where:

1. *#Sn* is the number of source nodes.
2. *#Dn* is the number of destination nodes.
3. *#E* is the number of edges contained in the graph *CG*.

For example, the architectural complexity of non-pipelined processor is given below:

$$AC_{compl}(\text{non-pipelined}) = (N+5, N+5, 3N+4)$$

where N is the number of general purpose processors.

A3. The bus complexity of a single bus, B_{compl} is defined as a 2-tuple (#*RC*, #*WC*), where

1. #RC is the number of read connections.
2. #WC is the number of write connection ports attached to the bus.

A4. The data path complexity, DPcompl, of a processor data path is defined as a 5-tuple (#*Bus*, #R_c, #W_c, #*RP*, #Regs), where

1. #*Bus* is the number of data buses.
2. #R_c the number of read connections in the data path.
3. #W_c the number of write connections in the data path.
4. #*RP* the number of ports on the RF and
5. #*Regs* the total number of registers required, including the general purpose registers and registers to hold immediate but exclusive internal FU registers.

A5. RC_{max} is the maximum number of read connections to any bus, and WP_{max} is the maximum number of write ports to any register.

Appendix II

Belief Propagation (BP) Algorithm [36]

BP is widely used for identifying the marginal probability in Markov models. For this reason, it is widely used in statistical interference, pattern recognition, Artificial intelligence and recently in forward error correction. Belief propagation is an inference algorithm for a particular kind of factorized joint probability distribution. The distribution is represented as a graph and the algorithm proceeds by passing messages along the edges of the graph according to a set of message-passing rules. Therefore, when it is required to solve the modeling problem then it is best to portray as a directed and undirected model but it cannot be translated or compiled into a factor graph. For example, consider the ubiquitous problem of computing marginal probability of a graphical model with N variables $\underline{x} = (x_1, ..., x_N)$ taking the values in a finite alphabet X. For conventional computing algorithm, it will take a time of order $|\mathrm{X}|^N$. If factor graph FG is applied then the computation complexity can be reduced dramatically. This recursive procedure can be known as message passing algorithm. Message passing algorithms operate on messages associated with edges of the FG, and update them recursively through local computations done at the vertices of the graph. Figure A2.1 shows the generation technique of FG

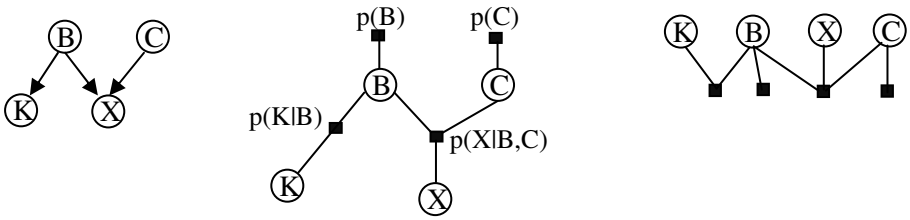


Figure A2.1: Generation technique of FG.

B is expressed by probability p(B), similarly we have unary factor C is expressed by p(C). Then we have a ternary factor X giving the conditional probability p(X|B,C) and finally the binary factor K depends on B p(K|B). This Bipartite graph can be alternatively written as the right of figure A2. Therefore, a factor graph can express compilations in both directed and undirected GMs. Figure **A**2.2 shows an example of FG. In this figure the round nodes represents the variable nodes and the square node corresponds the factor/function. The distribution corresponding to this graph is factorized as :

$$p(x_1, x_2, x_3, x_4) = \frac{1}{Z}\psi_a(x_1, x_2) \times \psi_b(x_1, x_3, x_4) \times \psi_c(x_2, x_4)$$

Suppose $x_1, x_2, \ldots, x_n$ be the variables of a finite domain D. Subsets $V(a) \subset \{1, \ldots\ldots, n\}$ are indexed by $a \in C$, where $|C| = m$. Given a subset $S \subseteq \{1, 2, \ldots\ldots, n\}$, we define $\mathbf{X}_S := \{x_i \mid i \in S\}$. Consider a probability distribution p over $x_1, x_2, \ldots, x_n$ that can be factorized as

$$p(x_1, x_2, \ldots\ldots, x_n) = \frac{1}{Z}\prod_{i=1}^{n}\psi_i(x_i)\prod_{a \in C}\psi_a(\mathbf{X}_{V(a)}) \tag{A2}$$

where $\psi_i(x_i)$ and $\psi_a(\mathbf{X}_{V(a)})$ are non negative real functions, referred to as compatibility functions and

$Z := \sum_{x_1, \ldots\ldots\ldots, x_n}\left[\prod_{i=1}^{n}\psi_i(x_i)\prod_{a \in C}\psi_a(\mathbf{X}_{V(a)})\right]$ is the normalized constant or partition factor [36]. A factor graph has represented this probability explained in equation A2 through a bipartite graph with V variables and C (set of $V(a)$) factors or functions. There is an edge between a variable node i and function node a if and only if $i \in V(a)$. We define also $C(i) := \{a \in C : i \in V(a)\}$[36]. Now if we want to compute the marginal probability of any variable i, as following

$$p(x_i) = \sum_{x_1 \in D} \ldots\ldots \sum_{x_{1-1} \in D} \sum_{x_{1+1} \in D} \ldots\ldots \sum_{x_n \in D} p(x_1, \ldots\ldots, x_n) \tag{A2.1}$$

Now, the question is how efficiently we calculate this marginal probability. The BP algorithm is an efficient algorithm for computing the marginal probability distribution of each variable of non-loop tree factor graph.

Let us draw the factor graph as in Figure A2.3, i.e., as a tree T rooted in x_i. Then, the children of x_i are the factors which contain x_i. The essential idea is to use the distributive property of the sum and product operations to compute independent terms for each sub tree recursively. This recursion can be cast as a message-passing algorithm, in which messages are passed up the tree.

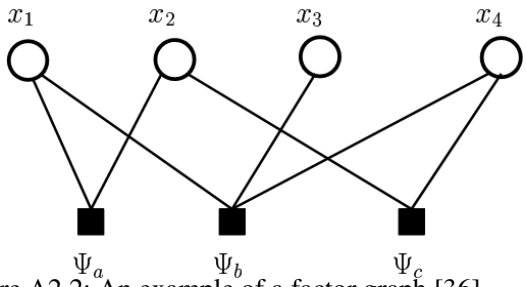


Figure A2.2: An example of a factor graph [36].

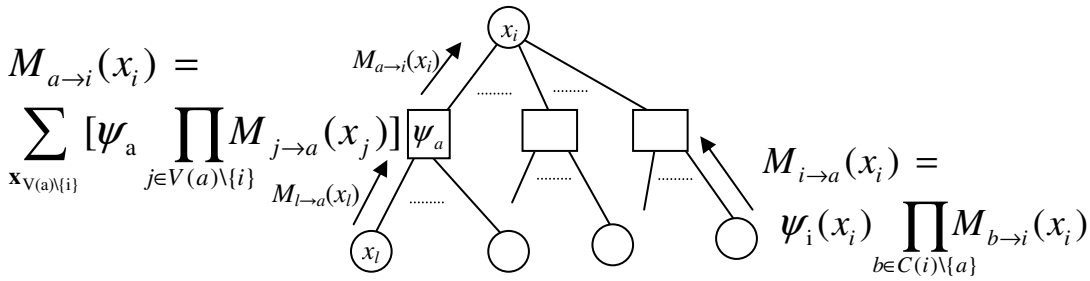


Figure A2.3: Cycle free Factor Graph with a recursive marginalization.

Let we assume that the vector $M_{i\to a}$ denote the message passed by variable node i to function node a. Similarly, the quantity $M_{a\to i}$ denotes the message passes from function node a to variable node i. Therefore, the messages from variable to function and function to variable nodes are updates as the following way [36]:

$$M_{a\to i}(x_i) \ \alpha \sum_{\mathbf{x}_{V(a)\setminus\{i\}}} [\psi_a(\mathbf{x}_{V(a)}) \prod_{j\in V(a)\setminus\{i\}} M_{j\to a}(x_j)]. \tag{A2.2}$$

$$M_{i\to a}(x_i) \ \alpha \ \psi_i(x_i) \prod_{b\in C(i)\setminus\{a\}} M_{b\to i}(x_i). \tag{A2.3}$$

It can be shown that for open FG, these updates will be converging after a linear number of iterations. After this convergence, the local marginal distribution at variable node and factor nodes can be computed as below:

$$F_i(x_i) \ \ \alpha \ \ \psi_i(x_i) \prod_{b\in C(i)} \hat{M}_{b\to i}(x_i) \tag{A2.4}$$

$$F_a(x_{V(a)}) \ \ \alpha \ \ \psi_a(x_{V(a)}) \prod_{j\in V(a)} \hat{M}_{j\to i}(x_j) \tag{A2.5}$$

Nowadays the BP algorithm can be used in error control coding like LT codes, LDPC or Raptor codes etc. It shows excellent result for error recovery when data are transmitted through BEC model.

References

[1] Dake Liu, “Embedded DSP Processor Design: Application Specific Instruction Set Processor,” M.K. Publishers, Elsevier,pp. 20-200, 2007.

[2] Predrag Radosavljevic, “Channel Equalization Algorithms for MIMO Downlink and ASIP Architectures,” Master’s Thesis, Rice University, Texas 2004.

[3] Otto Esko, “ASIP Integration and Verification flow for FPGA,” Master’s Thesis, Tampere University of Technology, Tampare Finland, may 2011.

[4] C. Hendrik, "Transport Triggered Architectures Design and Evaluation," Ph.D thesis, Technical University of Delft, 1995.

[5] Joseph Cavanagh,”Verilog HDL Digital Design and Modeling,” CRC press, pp. 650-699, 2007.

[6] W. Shi, H. Ren, T. Cao,W. Chen, B. Su, and H. Lu, “DSS: Applying Asynchronous Techniques to Architectures Exploiting ILP at Compile Time”, International Conference on Computer Design, pp. 321-327, Changsha China, 2010.

[7] V. Guzma, P. Jääskeläinen, P. Kellomäki, and J. Takala, “Impact of software bypassing on instruction level parallelism and register file traffic,” In Proc. Int. Workshop Emnedded Computer Syst.: Architecture, Modeling and simulation, pages 23-32, 2008.

[8] H. Jenkač,and T. Mayer, “Soft Decoding of LT-Codes for Wireless Broadcast,” In Proc. IST Mobile Summit, Germany 2005.

[9] J. Byers, M. Luby, and M. Mitzenmacher, “A digital fountain approach to asynchronous reliable multicast,” IEEE journal on selected Areas in Communications, vol. 20, pp. 1528-1540, 2002.

[10] C. Howland, and A. Blanksby, “A 220 mW 1 Gb/s 1024-bit rate ½ low density parity check code decoder,” IEEE conference on Custom Integrated Circuits, pp. 293-296, 2001.

[11] T. Zhang, and K. K. Parhi, “VLSI implementation-oriented (3,k)-regular low-density parity check codes,” IEEE workshop on Signal Processing Systems, pp. 25-36, 2001.

[12] C. E. Shanon, “A Mathematical Theory of Communication,” The Bell System Technical Journal, Vol. 27, pp 379-423, 623-653, Oct. 1948.

[13] Dino Sejdinovi¢, “Topics in Fountain Coding” Master’s Thesis, University of Bristol, 2009.

[14] G . Joshi, J. B. Rhim, J. Sun, and D. Wang, “Fountain Codes,” Notes on Principles of digital Communication II, MIT, Dec. 2010.

[15] T. Richardson, and R. Urbanke, “Modern Coding Theory,” Cambridge University Press, pp. 4-80, 2008.

[16] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. A. Spielman, “Efficient erasure correcting codes,”IEEE Trans. Inform. Theory, vol. 47, no. 2, pp. 569–584, 2001.

[17] M. Luby, “LT codes,” in Proc. IEEE Symp. Found. Comp. Sci., Vancouver, pp. 271–280, Nov. 2002.

[18] D.J.C. MacKay, “Fountain Codes,” IEE Proceedings – Communication, Vol. 152(6), pp. 1062-1068, 2005.

[19] Han Wang, “Hardware Designs for LT Coding,” Master’s Thesis, Technical University of Delft, The Netherlands 2006.

[20] K. Zhang, X. Huang, and C. Shen, “Soft Decoder Architecture of LT Codes,” IEEE workshop on Signal Processing Systems, pp. 210-215, 2008.

[21] T. Brandon, R. Hang, G. Block, V. C. Gaudet, B. Cockburn, S. Howard, C. Giasson, K. Boyle, P. Goud, S. S. Zeinoddin, A. Rapley, S. Bates, D. Elliott, C. Schlegel, “A scalable LDPC decoder ASIC architecture with bit-serial message exchange,” INTEGRATION the VLSI journal, vol. 41, pp. 385-398, 2008.

[22] TTA-based Co-design Environment v1.5. User Manual, Tampere University of Technology, Finland 2006.

[23] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, “Codesign Toolset for Application-Specific Instruction-Set Processors,” Proc. In: Conference on Multimedia on Mobile Devices, USA, SPIE Vol. 6507, pp. 1-11, 2007.

[24] P. Jääskeläinen, "From Parallel Programs to Customized Parallel Processors," doctoral dissertation, Tampere University of Technology, 2012.

[25] P. Jääskeläinen, "Instruction Set Simulator for Transport Triggered Architectures," Master of Science thesis, Tampere University of Technology, 2005.

[26] S. Alam and G. Choi, "Response of Transport Triggered Architectures for High-speed Processor Design," IEICE Electronics Express , Vol. 10, No. 5, pp. 1-6, March 2013.

[27] Metsähalme, "Instruction Scheduler Framework for Transport Triggered Architectures," Master of Science thesis, Tampere University of Technology, 2008.

[28] S. Alam and G. Choi, "Custom Code Generation Technique for ASIPs from High-level Language," IEICE Electronics Express (Submitted 2013).

[29] Tensilica Product, "Xtensa 7," Product Brief.

[30] Tensilica Product, "ConnX D2 DSP Engine," 2012 , http://www.tensilica.com/uploads/pdf/connx_d2_pb.pdf.

[31] Tensilica Product, "Tensilica Diamond Standard Controller," Data Book, 2012.

[32] J. Nurmi, Processor Design— System-on-Chip Computing for ASICs and FPGAs, Springer, The Netherlands, 2007.

[33] K. Anantha Ganesh Karikar, "Automatic Verification of Microprocessor designs using Random Simulation," Master's thesis, Uppsala University, Sweden 2012.

[34] D. Lampret, "OpenRISC 1000 Architecture Manual," OpenCores (2012).

[35] S. Alam and G. Choi, "Design and Implementation of LT Codec Architecture with Optimized Degree Distribution," IEICE Electronics Express (Accepted 2013).

[36] E. N. Maneva, "Belief propagation algorithms for constraint satisfaction problems," Ph.D thesis, California Institute of Technology, Berkeley 2006.

Acknowledgement

First and foremost, I am really thankful to almightily Allah for His enormous help to complete my thesis work successfully. Without His blessings, it is not possible for me to carry out this thesis work and to concentrate in writing with full devotion and consistency.

Then I would like to show my wholehearted gratitude and immense regard to my honorable supervisor, Prof. GoangSeog Choi for his valuable support, precious guidance and important suggestion to my work. During my lab work, he gave some novel ideas regarding Application Specific Instruction-set Processor (ASIP) and Chip Design procedure those were truly steered me to accomplish this task.

I am grateful to my supervisor that he has assigned me to participate in Multi Project Wafer Design project, 2012. Under this project, I went to Electronics and Telecommunication research Institute (ETRI) for getting training on Chip Design process that was beneficial for me to understand the ASIC as well as ASIP design. Moreover, I would like to give thanks to Engineers of Advanced Design Technology (ADT) and officials of ETRI for their continuous guidance in Chip Design process.

I was truly thankful to my co-supervisor Prof. Goo-Rak Kwon for his all kinds of support throughout my Master's program. For his kind consideration, I was able to complete my course and thesis work successfully.

Next, I was pleased to thesis evaluation committee members Professor Jae-Young Pyun and Professor Young-Sik Kim for their intellectual comments and important ideas regarding the modification of my thesis work those are really favorable for finishing task. Furthermore, I was also pleased to the anonymous reviewers of Electronics Express journal for their valuable comments and suggestions regarding the improvement the work.

After that, I acknowledged the invaluable test support from the Department of Computer Systems of Tampere University of Technology for developing the free tool

and supporting documents on transport triggered architecture. It was really beneficial for understanding the ASIP design procedure.

Besides this, I wish to thank all the members of SoC Design Lab and Digital Media Computing Lab for their kind help and support during my study period. The members of these labs have been a real family to me. Thanks a lot!

I am grateful to the Chosun University for Research Assistantship (RA) and the Korean Government for selecting me through Global IT Talent Scholarship Program under National IT industry Promotion Agency (NIPA), without this support it is not possible for me to study in Korea. Besides this, the Bangladesh Government through my job place Khulna University, Bangladesh was really kind to allow me to study outside and giving me study leave. I wish to give thanks to my departmental head and my colleagues including the entire management staff of Khulna University for letting me to grab this opportunity. I earnestly desire that this seed of kindness would someday soon germinate into a harvest of technology.

Finally I would like to bestow my extended thank to my family- my father, mother and all my siblings and sibling-in-law for giving me encourage and keeping me in touch which are very supportive to continue study in Korea. At the same time, there were many other people, who played vital role to make my stay successful in Korea. From bottom of my heart, I say Thanks a lot!

South Korea, May 16, 2013

S. M. Shamsul Alam

알람 샴술 석사학위 논문을 인준함

위원장 조선대학교 교수 변 재 영 印

위 원 조선대학교 교수 김 영 식 印

위 원 조선대학교 교수 최 광 석 印

2013년 5월

조 선 대 학 교 대 학 원

Graduate School of Chosun University

Gwangju, South Korea

CERTIFICATE OF APPROVAL

MASTER'S THESIS

This is to certify that the master's thesis of

S. M. Shamsul Alam

has been approved by examining committee for the thesis requirement for the Master's degree in Engineering

Committee Chairperson .....

Prof. Jae-Young Pyun

Committee Member .....

Prof. Young-Sik Kim

Committee Member .....

Prof. GoangSeog Choi