ARCHITECTURE DESIGN AND EVALUATION OF LDPC DECODER ON TTA BASED CODESIGN ENVIRONMENT

Sudeep Kanur Chandra Shekar

Master of Science Thesis
Supervisor: Johan Lilius
Advisors: Kristian Nybom & Sebastien Lafond
Department of Information Technologies
Åbo Akademi University
August 2013
ABSTRACT

High quality digital video transmission requires efficient and reliable data communication over broadcasting channels as there is a risk of data corruption associated during transmission. The near channel performance of Low Density Parity Check Codes (LDPC) has motivated its use in second generation Digital Video Broadcasting (DVB) standards for mobile, cable, satellite and terrestrial channels as an error correction code. But iterative decoding of LDPC codes provides significant implementation challenges as the complexity grows with the code size. This problem can be mitigated by exploiting the modular nature of the iterative decoding scheme for efficient parallel implementation.

Transport Triggered Architecture (TTA) provides a processor template that exploits operation style parallelism and parallelism at data transport level. TTA-Based Co-design Environment (TCE) provides the necessary toolset to design a TTA processor. In addition, the toolset also provides the means to design application specific processors to accelerate the execution of the application and implement the processor on reconfigurable logic platform such as Field Programmable Gate Arrays (FPGAs) with ease. This work leverages TCE toolset to implement iterative decoding scheme such as reduced minimum sum algorithm on field programmable gate arrays. It also presents the throughput gains and evaluates the capabilities of the TTA architecture and the TCE toolset for the design of application specific instruction set processors.

Keywords: FPGA, TCE, Low Density Parity Check Codes, DVB-T2, Transport triggered architecture
CONTENTS

Abstract i

Contents ii

List of Figures iv

Glossary vi

1 Introduction 1
   1.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
   1.2 Goal of this thesis . . . . . . . . . . . . . . . . . . . . . . . . . . 3
   1.3 Thesis structure . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2 Second generation Digital Video Broadcasting - Terrestrial 4
   2.1 DVB-T2 Transmission Standard . . . . . . . . . . . . . . . . . . . 4
   2.2 Decoding of DVB-T2 frames . . . . . . . . . . . . . . . . . . . . . 7

3 Low Density Parity Check codes 8
   3.1 Approaching channel capacity . . . . . . . . . . . . . . . . . . . . 8
   3.2 Linear Block Codes . . . . . . . . . . . . . . . . . . . . . . . . . . 11
   3.3 Encoding techniques . . . . . . . . . . . . . . . . . . . . . . . . . 14
   3.4 Decoding techniques . . . . . . . . . . . . . . . . . . . . . . . . . 15
      3.4.1 Minimum Sum Algorithm . . . . . . . . . . . . . . . . . . . . 16
      3.4.2 Reduced Minimum Sum Algorithm . . . . . . . . . . . . . . 18
      3.4.3 M kernel . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

4 Application Specific Instruction-set Processor design using TCE toolset 23
   4.1 Transport Triggered Architecture (TTA) . . . . . . . . . . . . . . 24
      4.1.1 Hardware Architecture . . . . . . . . . . . . . . . . . . . . . 25
      4.1.2 Software Design . . . . . . . . . . . . . . . . . . . . . . . . . 27
      4.1.3 Computation Example . . . . . . . . . . . . . . . . . . . . . 28
   4.2 TTA based Codesign Environment (TCE) . . . . . . . . . . . . . . 33
      4.2.1 TCE structure . . . . . . . . . . . . . . . . . . . . . . . . . . 33
      4.2.2 Design Flow . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3 Custom ASIP design on FPGAs ........................................... 38
4.3.1 Field Programmable Gate Arrays ................................. 40
4.3.2 Integrating with FPGA ............................................. 42

5 Decoder Design ............................................................. 44
5.1 Design procedure ....................................................... 44
5.2 Basic Architecture ....................................................... 47
5.3 Designing custom processor ......................................... 53
5.3.1 Code analysis ......................................................... 54
5.3.2 Parallelising check-node stage .................................. 57

6 Conclusion ..................................................................... 64
6.1 Summary of Results ..................................................... 64
6.2 Conclusion .................................................................. 66
6.3 Future Work .............................................................. 68

Bibliography .................................................................... 69

A Appendix ........................................................................ 74
A.1 Alist Format .............................................................. 74
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Generic Architecture of DVB-T2 transmitter system</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Architecture of Forward Correction Unit at decoder</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>Model of a Binary Symmetric Channel</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Comparison of performance (code rate vs. probability of error) for various algebraic codes for a BSC model with $p = 0.74$. Squares are repetition codes and Hamming codes. Other points are Reed-Muller and BCH codes [1].</td>
<td>9</td>
</tr>
<tr>
<td>3.3</td>
<td>Model for additive white gaussian noise (AWGN) channel</td>
<td>11</td>
</tr>
<tr>
<td>3.4</td>
<td>Representation of a strictly systematic linear block</td>
<td>12</td>
</tr>
<tr>
<td>3.5</td>
<td>Relationship between linear equations and parity check matrix for (7,4) Hamming code</td>
<td>12</td>
</tr>
<tr>
<td>3.6</td>
<td>Tanner graph for (7,4) Hamming code</td>
<td>13</td>
</tr>
<tr>
<td>3.7</td>
<td>Generalised representation of Tanner graph</td>
<td>17</td>
</tr>
<tr>
<td>3.8</td>
<td>Parity check matrix of rate 1/2 DVB-T2 LDPC code of size $32000 \times 64000$</td>
<td>20</td>
</tr>
<tr>
<td>3.9</td>
<td>Conceptual representation of sub matrices for a LDPC code</td>
<td>21</td>
</tr>
<tr>
<td>3.10</td>
<td>Rearranged matrix representation of rate 1/2 DVB-T2 matrix of size $32000 \times 64000$</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>General Architecture of a TTA processor</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Structure of a functional unit (FU)</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>Instruction format of TTA machine</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>TTA instruction pipeline a) Example of a three stage pipeline b) Example of operation with latency of three</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>TTA machine with single bus</td>
<td>30</td>
</tr>
<tr>
<td>4.6</td>
<td>TTA machine with three data bases</td>
<td>31</td>
</tr>
<tr>
<td>4.7</td>
<td>Optimised TTA machine with three data bases</td>
<td>32</td>
</tr>
<tr>
<td>4.8</td>
<td>TTA machine with a custom unit</td>
<td>32</td>
</tr>
<tr>
<td>4.9</td>
<td>Software architecture of TCE toolset. Shaded regions represent tools, while unshaded regions represent file formats [2]</td>
<td>34</td>
</tr>
<tr>
<td>4.10</td>
<td>Initialisation Phase [2]</td>
<td>36</td>
</tr>
</tbody>
</table>
4.14 Design flow for design of custom operations .......................... 39
4.15 Architecture of Stratix III FPGAs ................................. 40
4.16 High level block diagram of adaptive logic modules (ALM) ........... 41
4.17 Concise design flow for custom ASIP design on FPGAs ............... 42

5.1 Overview of the DE3 FPGA development board ......................... 45
5.2 The initial processor (IP) configuration .................................. 48
5.3 Custom memory architecture combining M9K and M144K memory blocks .................................................. 49
5.4 Comparison of throughput rates for MSA RMSA algorithms on various basic processor configuration for DVB-T2 short matrix ........... 52
5.5 Comparison of throughput rates for all the three algorithms on various basic processor configuration for DVB-T2 long matrix .......... 53
5.6 Custom processor design with \textit{minmod} unit ........................... 55
5.7 Block diagram of \textit{minmod} unit ........................................ 56
5.8 Comparison of throughput rates for all the three algorithms on various processor configuration using \textit{minmod} unit for DVB-T2 long matrix .......................... 56
5.9 Comparison of throughput rates for MSA RMSA algorithms on various processor configuration using minmod unit for DVB-T2 short matrix .................................................. 57
5.10 Custom processor design with \textit{sclmmod} unit ............................ 57
5.11 Block diagram of \textit{sclmmod} unit ........................................ 58
5.12 Comparison of throughput rates for all the three algorithms on various processor configuration using \textit{sclmmod} unit for DVB-T2 long matrix .......................... 59
5.13 Comparison of throughput rates for MSA RMSA algorithms on various processor configuration using sclmmod unit for DVB-T2 short matrix .................................................. 60
5.14 Additional units added to the basic TTA processor used for decoding LDPC codes .................................................. 60
5.15 Dependency graph of \textit{cnus} instruction ............................... 61
5.16 Comparison of throughput rates of RMSA and M-kernel algorithms on various processor configuration using \textit{cnus} unit for DVB-T2 long matrix .................................................. 62
5.17 Data path design of the \textit{cnus} unit ........................................ 63

6.1 Comparison of throughput rates for all the three algorithms on various processor configuration for DVB-T2 long matrix .................. 65
GLOSSARY

AWGN  Additive White Gaussian Noise
ASIC  Application Specific Integrated Circuit
ASIP  Application Specific Instruction-set Processor
DSP  Digital Signal Processor
DVB  Digital Video Broadcasting
DVB-C  Digital Video Broadcasting, Cable
DVB-C2  Second Generation Digital Video Broadcasting, Cable
DVB-H  Digital Video Broadcasting, Handheld
DVB-S  Digital Video Broadcasting, Satellite
DVB-SH  Digital Video Broadcasting, Satellite Handheld
DVB-S2  Second Generation Digital Video Broadcasting, Satellite
DVB-T  Digital Video Broadcasting, Terrestrial
DVB-T2  Second Generation Digital Video Broadcasting, Terrestrial
FEC  Forward Error Correction
FPGA  Field Programmable Gate Array
IRA  Irregular and Repeat Accumulate
LDPC  Low Density Parity Check codes
LLR  Log Likelihood Ratio
LSU  Load Store Unit

MSA  Minimum Sum Algorithm

PLP  Physical Layer Pipes

RMSA Reduced Minimum Sum Algorithm

RTL  Register Transfer Level

SFN  Single Frequency Network

TCE  TTA based Codesign Environment

TTA  Transport Triggered Architecture

VHDL Very-high-speed integrated circuits Hardware Description Language

VLIW  Very Long Instruction Word
1 INTRODUCTION

1.1 Overview

Digital video broadcasting is seeing growth in popularity with the introduction of high
definition television over recent years. The growth has created greater needs for higher
transmission rates and coding efficiency. Broadcasting channels offer much higher
data rates and efficiency when compared to other transmission channels such as cables
and DSL. Broadcasting channels can also hold multiple channels of different qualities
of video data within existing capacities of the channel.

Of various standards regulating digital content broadcasting, Digital Video Broad-
casting in one set of open standards maintained by the DVB group consisting of an
industry consortium of over 270 members. DVB standards allow transmission of data
through approaches including satellite, terrestrial, cable and microwave channels. The
first generation of DVB introduced standards for satellite (DVB-S), cable (DVB-C),
terrestrial (DVB-T), handheld devices (DVB-H) and for handheld devices with satel-
lite channel (DVB-SH). The standards were improved and the second generation of
DVB were introduced for satellites (DVB-S2), terrestrial (DVB-T2) and cables (DVB-
C2). With widespread acceptance of DVB-T2 standards and gradual conversion from
previous standards to the current one, DVB-T2 has spanned to 54 countries [3].

The efficient and reliable digital data communication over broadcasting channels
faces an inherent problem that information may be altered or lost during transmis-
sion due to channel noise. For transportation of compressed video, any information
loss could cause significant visual distortion and loss of audio video synchronisation.
Multipoint communications further complicate the problem of error recovery from re-
transmission. Forward Error Correction (FEC) techniques have been developed to
counter such issues by redundantly encoding the source data using error correcting
codes (ECC) such that the messages can be retrieved back in the receiver in case of
noisy channels without the need of retransmission. DVB first generation systems have
introduced FEC methods such the Reed Solomon (RS) codes in DVB-T and DVB-C standards, but with data rates approaching limits of the channel capacity, capacity-approaching codes like Low Density Parity Check (LDPC) Codes were used. Second Generation systems use LDPC along with BCH codes to achieve high efficiency and reliability.

The encoding process of LDPC codes are relatively straight forward and is performed by the transmitting systems, but the decoding of the LDPC codes is a NP complete problem. In addition the standard specifies two different frame sizes and two set of code rates to chose from. High throughput along with size, weight and power (SWaP) constraints on electronics have presented an opportunity to try various implementation platforms. The work in [4] and [5] presents IP cores for DVB-S2 standards on an ASIC, while [6] present the same on flexible platforms such as field programmable gate arrays (FPGAs). Software Defined Radio (SDR) and GPU implementation have also been implemented for the standard [7]. While these implementations have achieved real time throughput rates, they come at the cost of increased design time as the IP cores are hand-coded in all the cases or at increased size of the hardware as in the case of GPUs and SDR. This work seeks motivation from this problem to implement decoding of LDPC codes without compromising the real time data throughput requirement.

Transport Triggered Architecture (TTA) are special cases of Very Long Instruction Word (VLIW) processor architecture that have proved to provide significant speed up compared to operation triggered architectures [8]. TTA provides a processor template with customisable data-paths without the cost of inflexibility often encountered in design of hardware accelerators thus decreasing the complexity and design time. The modular nature of TTA makes it suitable for designing decoders for LDPC as several decoding algorithms are inherently parallel and modular in structure [9].

TTA based Codesign Environment (TCE) is a toolset for the TTA architecture developed by the Department of Computer Science at Tampere University of Technology [10]. TCE aids in the designing of TTA processors that supports C/C++ and OpenCL programming languages. TCE also allows the user to compile the program, simulate the processor, analyse the performance and generate HDL code for the processors for ASIC or FPGA implementation. TCE supports Altera FPGA platform and can generate HDL which can natively integrate with the Altera design platform for FPGAs.
1.2 Goal of this thesis

The thesis is motivated by the possibility to exploit the modular nature of TTA architecture in the design of LDPC decoders without trying to compromise real time throughput requirements and flexibility demanded by such decoders. The goal is to evaluate the implementation of a LDPC decoder with TTA architecture using TCE toolchain on FPGA. Area and power consumptions on FPGA are measured along with data throughput rates. The flexibility of the design is tested against various standard and non standard parity matrices for two variations of the minimum sum decoding algorithm. In this way this work also tries to evaluate the ease of implementation of modular and data intensive algorithms with TCE toolchain and tries to analyse the capabilities of the toolchain.

1.3 Thesis structure

The thesis is structured in a top-down fashion where the reader is given an overview where this work fits and explanation is provided in context of this work. The chapters in this thesis are structured as:

Chapter 2 describes the second generation Digital Video Broadcasting standard as applied to terrestrial channel from the perspective of this work.

Chapter 3 introduces Low Density Parity Check codes. Encoding and decoding schemes are discussed in this chapter.

Chapter 4 gives insight into Transport Triggered Architecture architecture, TTA based Codesign Environment toolchain and Field Programmable Gate Arrays.

Chapter 5 explores various aspects of the toolchain and the algorithm. It also details processors developed in this process and presents measurements made with respect to throughput, size and power.

Chapter 6 analyses the results and concludes the thesis.
2 SECOND GENERATION DIGITAL VIDEO BROADCASTING - TERRESTRIAL

LDPC codes have been extensively used as error correction codes in the second generation digital video broadcasting. The second generation digital video broadcasting was introduced to increase the transmission capacity of the broadcasting channels and support variety of new features. It is known to give an increased capacity of 50% when compared with its first generation counterpart (DVB-T) in the UK and about 67% in single frequency network (SFN) operation mode [11]. The new standards were created for terrestrial, satellite, cable and mobile broadcasting channels. Along with definitions for forward error correction, the standard covered other features and modes summarised in the table 2.1. The second generation digital video broadcasting standard is being widely adopted worldwide. The architecture and the frame structure of a second generation digital video broadcasting transmitter system is given in detail in [11] and [12]. This section provides an simplified overview of the system and attempts to give an overview where this work fits.

2.1 DVB-T2 Transmission Standard

The generic model of a second generation digital video broadcasting terrestrial system is shown in the Fig. 2.1. The system input could be one or more MPEG-2 Transport
<table>
<thead>
<tr>
<th>Features</th>
<th>DVB-S2</th>
<th>DVB-T2</th>
<th>DVB-C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Interface</td>
<td>Multiple Transport and Generic Streams</td>
<td>Multiple Transport and Generic Streams</td>
<td>Multiple Transport and Generic Streams</td>
</tr>
<tr>
<td>Modes</td>
<td>Variable Coding &amp; Modulation and Adaptive Coding &amp; Modulation</td>
<td>Variable Coding &amp; Modulation</td>
<td>Variable Coding &amp; Modulation and Adaptive Coding &amp; Modulation</td>
</tr>
<tr>
<td>Forward Error Correction</td>
<td>LDPC + BCH 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 5/6, 8/9, 9/10</td>
<td>LDPC + BCH 1/2, 3/5, 2/3, 3/4, 4/5, 5/6</td>
<td>LDPC + BCH 1/2, 2/3, 3/4, 4/5, 5/6, 8/9, 9/10</td>
</tr>
<tr>
<td>Modulation</td>
<td>Single carrier QPSK with Multiple Streams</td>
<td>OFDM</td>
<td>absolute OFDM</td>
</tr>
<tr>
<td>Modulation schemes</td>
<td>QPSK, 8PSK, 16APSK, 32APSK</td>
<td>QPSK, 16QAM, 64QAM, 256QAM</td>
<td>16- to 4096-QAM</td>
</tr>
<tr>
<td>Guard Interval</td>
<td>Not applicable</td>
<td>1/4, 19/256, 1/8, 19/128, 1/16, 1/32, 1/128</td>
<td>1/64 or 1/128</td>
</tr>
<tr>
<td>Fourier Transform Size</td>
<td>Not Applicable</td>
<td>1k, 2k, 4k, 8k, 16k, 32k DFT</td>
<td>4k Inverse FFT</td>
</tr>
<tr>
<td>Pilots</td>
<td>Pilot symbols</td>
<td>Scattered and Continual Pilots</td>
<td>Scattered and Continual Pilots</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison between various second generation DVB standards.
streams or Generic Streams and system outputs a single signal to be transmitted over the network. The input preprocessor block divides the Transport Streams (TS) into one or more logical data input streams that form the input to the system. The preprocessor, which is not part of the DVB-T2 physical layer, outputs data in the form of Physical Layer Pipes (PLP) to the input processing system. The input processing system maps the PLPs into internal bit format and perform stream synchronisation by guaranteeing constant bit rate and frame to frame delay. The subsystem also deletes null packets, apply CRC-8 encoding and insert baseband frame header resulting in baseband frames (BBFRAMES). Channel encoding on scrambled BBFRAMEs are done by forward correction (FEC) unit, a subsystem of bit interleaved coding & modulation block, to produce forward error corrected frames (FECFRAMEs). Channel encoding is done in two steps, the inner coding is done using LDPC codes and outer coding is done using Bose-Chaudhuri-Hocquenghem (BCH) codes. In the same module, the FECFRAMEs are bit interleaved using parity interleaving and cell twist interleaving techniques. The frames are then mapped to a coded and modulated FEC block by first demultiplexing the input bits into parallel cell words and then mapping these cell words into quadrature phase shift keying (QPSK) or quadrature amplitude modulation (QAM) constellation points. It also performs cell and time interleaving to improve performance against channel distortions on FECFRAMEs.

The frame builder subsystem assembles the cells produced by time interleaver for each of the PLPs, into arrays of active orthogonal frequency division multiplexing (OFDM) cells. These OFDM cells correspond to each of the OFDM symbols that make up the overall frame structure. The OFDM generation module takes the cells produced by the frame builder, as frequency domain coefficients, to insert the relevant reference information, known as pilots. Pilots allow the receiver to compensate for the distortions introduced by the transmission channel, and to produce from this the basis for the time domain signal for transmission. It also inserts guard intervals to produce the completed T2 signal.
2.2 Decoding of DVB-T2 frames

The demodulation of the signal at the receiver is the reverse of the process mentioned above, and the standard has set aside several recommendations for its implementation. However, the standard does not enforce the implementation of receiving process, as there are several other ways to implement the same. The forward error correction subsystem is of interest in this work, as this subsystem employs LDPC decoder for inner channel decoding. The block diagram of the forward error correction subsystem at the receiver side that forms a part of bit interleaved decoding & demodulation module is shown in Fig. 2.2. The input to this subsystem are probability values of a certain bit being transmitted expressed as Log Likelihood Ratios (LLRs), described in detail in section 3.4, which is obtained by de-mapping of rotated constellation. A positive LLR would indicate that bit was more probably transmitted as 1, a negative value indicates that transmission was probably a 0. These LLRs are de-interleaved and fed to LDPC decoder to remove effects of channel noise on the bit stream and recover useful information. Across various transmission standards, the implementations of LDPC decoder vary only in the code rates used. Hence, this work can be extended to other second generation standards without major modifications.
3 **Low Density Parity Check codes**

Low-Density Parity-Check Codes were first discovered by Robert Gallager in 1962, where he reported its performance on various channel models and introduced an iterative decoding scheme [13]. However due to the lack of processing capabilities, LDPC codes remained largely forgotten. It was not until 1995-96 that these codes were rediscovered simultaneously and independently by two different communities [14] [15]. LDPC codes have since gained popularity for their near channel limit performance and have been adapted in many applications ranging from deep space missions to terrestrial broadcasting [16] [17] [18].

3.1 **Approaching channel capacity**

A communication channel is the medium used to transmit the signal from transmitter to receiver systems. It may be a pair of wires, coaxial cables, a band of radio frequencies, light beams etc. The channel affects the transmitted signal by distorting it with noise, making reception of original message difficult. The profile of transmitted signal can be obtained from received signal if we model the channel in-between. A communication channel is said to be discrete in nature if the input and output signals to the channel are discrete symbols. A simple model of a discrete channel called Binary Symmetric Channel is shown in Fig. 3.1. The outcome of this channel can be two symbols. First, the signals arrive are affected by noise with the error probability $p$. Second, the signals arrive are undistorted with probability $q = 1 - p$ at the receiver. A continuous channel with bandwidth $B$ is modelled with thermal white noise additive in nature. Such a continuous channel is called Additive White Gaussian Noise (AWGN) channel and the model is closer to reality.

Forward error correction techniques are used to introduce redundancy to the information during transmission such that original message can be retrieved from the received signal distorted by the channel noise. To decode a received signal with arbit-
For a BSC channel, the channel capacity is defined as

\[ C_{BSC}(p) = 1 - h(p) \]  

(3.1)
where \( h(p) \) is the binary entropy function defined as

\[
h(p) = -p \log(p) - (1 - p) \log(1 - p)
\]  

(3.2)

Fig. 3.2 shows the theoretical limit for BSC channel and performance of various standard codes against bit error rate. The LDPC codes used here is characterised by a parity check matrix of size \( 10000 \times 20000 \). From the Fig. 3.2, it can be seen that LDPC codes outperform other codes and achieve near channel limit performance with sufficiently large code rate.

For an AWGN channel, the capacity is defined as

\[
C_{AWGN} = B \cdot \log_2(1 + \frac{S}{N})
\]  

(3.3)

where a signal of strength \( S \) is distorted by white thermal noise \( N \) when propagating through channel with bandwidth \( B \). For a band-limited communication system of bandwidth \( B \), and in the presence of white thermal noise, the noise power \( N \) is given by \( N = N_0B \), where \( N_0 \) is the power spectral density of noise in the channel. There is an equivalent expression for signal-to-noise ratio described by average bit energy \( E_b \) and transmission rate \( R \). For \( R = C \) then,

\[
\frac{E_b}{N_0} = \frac{S}{N_0R} = \frac{S}{N_0C}
\]  

(3.4)

\[
\frac{C}{B} = \log_2 \left( 1 + \frac{E_b}{N_0} \cdot \frac{C}{B} \right),
\]  

(3.5)

\[
2^{C/B} = 1 + \frac{E_b}{N_0} \cdot \left( \frac{C}{B} \right)
\]  

(3.6)

For a particular rate of transmission \( R \),

\[
2^{R/B} \leq 1 + \frac{E_b}{N_0} \cdot \left( \frac{R}{B} \right)
\]  

(3.7)

Eq. 3.7 is depicted in the Fig. 3.3 which shows two regions, one of practical use and another of impractical use separated by curve \( R = C \). Rearranging Eq. 3.7 and applying limits \( B \rightarrow \infty \) gives the Shannon limit at \( \left( \frac{E_b}{N_0} \right)_{dB} = -1.59 \) dB as \( R/B \rightarrow 0 \). The work in [14] and [20] present the channel approaching capacity of LDPC codes for AWGN channels against various standard codes. LDPC codes are known to approach
3.2 Linear Block Codes

The introduction of redundancy and its impact on performance was discussed in the previous section. The information symbols can be mapped either by dividing them into independent blocks or by generating them as a function of present and previous inputs and outputs. The former codes are known as block codes, while the latter are known as convolutional codes. If the mapping is a linear function on block, such codes are called linear block codes.

Let us consider a source emitting messages $s$ which can be divided into blocks of length $k$ symbols. The message symbols can be arbitrary, but belongs to a known alphabet of size $2^l$, where $l$ is the number of bits in each symbol. A systematic linear block code is defined as a set of codewords such that every codeword consists of all $k$ information symbols. A strictly systematic linear block code is defined as a set of codewords such that every codeword has redundancy added either at its beginning or at its end. Such a codeword $x$ has a length of $n$ symbols with $k$ information symbols and $n - k$ parity symbols where $n \geq k$. A representation of strictly systematic linear
Figure 3.4: Representation of a strictly systematic linear block

Figure 3.5: Relationship between linear equations and parity check matrix for (7,4) Hamming code

block codeword is shown in Fig. 3.4. The rate of a code is defined as

$$R = \frac{k}{n},$$

and gives the fraction of useful information in the transmitted code. A higher code rate signifies efficient utilisation of the bandwidth for transmitting useful information and is desirable. However, a higher code rate also means less redundancy in the information and hence reduces the error correction performance of the code.

In linear block codes, the relationship between the information symbols and the parity symbols are represented in terms of linear equations. These equations can be represented in the form of a matrix called parity matrix, denoted as $H$. Given a $m \times n$ parity matrix $H$ with $m$ rows and $n$ columns, one can note that the length of the codeword $x$ is the same as the number of columns $n$ in the matrix. One can also deduce the number of parity symbols using the relation $m = n - k$. The concept of parity check matrix is illustrated using a (7,4) Hamming code where, $n = 7$ and $k = 4$ with binary symbols. The linear equations describing the relationship between message bits and parity bits as well as its corresponding parity check matrix is given in Fig. 3.5.

The parity matrix of LDPC codes are constructed using $m \times n$ sparse matrix, where
A sparse matrix has a large number zero elements compared to non-zero elements and the number of non-zero entries in $H$ grow linearly as $O(n)$. Hence the name Low Density Parity Check codes. LDPC codes being linear block codes exhibit the property that the sum of any two codeword is a valid codeword in the code, and there exists a codeword which consists of all zero symbols. For a codeword $C$, the linear block codes in general exhibit the relation,

$$H \cdot x^T = 0^T$$  \hspace{1cm} (3.9)\

Let the weight of rows $m$ and weight of columns $n$ of $H$ be given as $w_m$ and $w_n$ respectively, where weight of a row or column is defined as number of ones present in that row or column. Then a LDPC code satisfies the condition $m \gg w_m$ and $n \gg w_n$. A code is called $(w_m, w_n)$-regular LDPC code if $w_m$ & $w_n$ are constants and irregular LDPC code otherwise. The second generation digital video broadcasting systems use irregular codes as it have been proven that irregular codes are closer to channel capacity than regular codes [21] [22].

Tanner generalised the LDPC codes and showed that they can be effectively be represented by bipartite graphs i.e a graph made from connecting nodes with edges consisting of only two categories of nodes and all edges may connect only to these nodes [23]. The bipartite graphs are constructed from the parity check matrix $H$ by connecting check nodes with variable nodes. It is drawn according to the following rule: a check node $i$ can connect to variable node $j$ whenever the element $h_{ij}$ of $H$ is 1, where $i$ & $j$ represent rows and columns of parity matrix. These graphs, known as Tanner graphs, represent the entire LDPC code and can help in understanding decoding algorithms for LDPC codes. Fig. 3.6 shows the tanner graph of the parity check matrix.
for (7,4) Hamming code. The row indices of the parity check matrix is given by \( r_i \) and column indices are given by \( x_i \). Note that for a given code \( x \), there are \( n \) variable nodes and \( n - k \) check nodes.

### 3.3 Encoding techniques

Encoding is the process of mapping information \( s \) of size \( k \) onto codeword \( x \) of size \( n \) by adding \( n - k \) parity bits. An \((n, k)\) block code for a channel \( Q \) is a list of \( S = 2^k \) codewords

\[
\{x^{(1)}, x^{(2)}, \ldots, x^{(2^k)}\}, \quad x^{(s)} \in A^n_X
\]

where \( A^n_X \) is the input alphabet, each of length \( n \). Using this code we can encode any binary input sequence \( s \) of length \( k \) among \( 2^k \) possible sequences to a unique codeword \( x \) of length \( n \). LDPC codes can be encoded using several methods. Let us consider a parity check matrix \( H \) with full rank. By applying Gauss-Jordan elimination, the matrix can be arranged as

\[
H = [P|I_{n-k}], \quad (3.10)
\]

where \( I_{n-k} \) is a \((n - k) \times (n - k)\) identity matrix and \( P \) is a \((n - k) \times k\) parity sub-matrix. The \( k \times n \) generator matrix can be found by

\[
G = [I_k|P^T] \quad (3.11)
\]

The mapping can be done by

\[
s \cdot G = x \quad (3.12)
\]

In general, for a full rank \( H \) matrix, it can be divided into two sub-matrices \( H_s \) and \( H_p \) with size \((n - k) \times k\) and \((n - k) \times (n - k)\) respectively as \( H = [H_s|H_p] \). Correspondingly, the codewords can also be divided into information and parity symbols as \( x = (x_s|x_p) \). Eq. 3.9 now becomes

\[
H_s \cdot x_s + H_p \cdot x_p = 0 \quad (3.13)
\]

Assuming \( H_p \) is square and invertible, then parity symbols \( x_p \) can be found out by,

\[
x_p = H_p^{-1} \cdot H_s \cdot x_s \quad (3.14)
\]
The relation $H_p^{-1} \cdot H_s$ is predetermined, speeding up mapping process.

The LDPC codes used in DVB-T2 are quasi cycle irregular repeat accumulate (IRA) codes with structure, $H = [A|B]$, where $A$ is $(n-k) \times k$ sparse matrix and $B$ is $(n-k) \times (n-k)$ staircase lower triangular matrix. Due to this structure, these codes offer easy encoding of information bits without the use of a generator matrix or the need of inversion operation on matrix. Encoding can be performed directly from the parity check matrix and this technique is mentioned in detail in [12].

### 3.4 Decoding techniques

Decoding is the process of finding the most likely transmitted codeword given the channel output $y$. Formally, for a block code $(n, k)$, decoding is a process of mapping from a set of length $n$ strings of the channel outputs $y$, $A^n_y$, to codeword label $\hat{x}$. The value $\hat{x}$ is the estimated value of the channel input $x$ corrected for channel distortion. The value $\hat{s} = \{0, 1, 2, \ldots, 2^k\}$, which is the estimated value of original message can thus be obtained from $\hat{x}$. An optimal decoder for a channel code is one which minimises the probability of block error caused due to the channel noise. It decodes a channel output $y$ to the input $x$ that has maximum posterior probability $P(y|x)$ given by Bayes’ theorem as

$$P(x|y) = \frac{P(y|x) \cdot P(x)}{\sum_{x'} P(y|x') \cdot P(x')} \quad (3.15)$$

where $P(y|x)$ is the conditional probability of $y$ given $x$ and $x' \in A_X$. If the alphabets are binary in nature, i.e. $A_X = A_Y = \{0, 1\}$, then the conditional probabilities for $x = 1$ and $x = 0$ can be given from Eq. 3.15 as,

$$P(x = 1|y) = \frac{P(y|x = 1) \cdot P(x)}{P(y|x)} \quad (3.16)$$

$$P(x = 0|y) = \frac{P(y|x = 0) \cdot P(x)}{P(y|x)} \quad (3.17)$$

This posterior probability is determined by three factors: the prior probability $P(x)$, the data dependent term $P(y|x)$ called likelihood and the normalising factor $P(y, x)$ called joint probability. The value $\hat{x}$ is decided as $\hat{x} = 0$ if $P(x = 0|y) > P(x = 1|y)$ and $\hat{x} = 1$ otherwise. Assuming prior probabilities to be equal i.e. $P(x = 0) = P(x = 1) = 0.5$, deciding $\hat{x}$ depends only on likelihoods expressed in the form of ratio shown
in Eq. 3.18. The decision is now made as \( \hat{x} = 0 \) if \( \bar{\Lambda} > 1 \) and \( \hat{x} = 1 \) otherwise.

\[
\bar{\Lambda} = \frac{P(y|x = 0)}{P(y|x = 1)}, \quad \bar{\Lambda} \geq 0
\] (3.18)

Encoding of LDPC codes and linear block codes in general is a straightforward in nature, but the decoding problem of finding maximum likelihood of \( s \) is a NP-complete problem [24]. For LDPC codes, Gallager proposed an iterative probabilistic decoding scheme based on message or belief propagation [13]. The Sum Product algorithm is a well known iterative decoding algorithm based on belief propagation. The details about the algorithm and its performance on various channels can be found in [25]. The Minimum Sum algorithm, an approximation of the Sum Product algorithm, simplifies the implementation of the algorithm on the hardware with reduced performance. This algorithm will be discussed here using Tanner graph as a tool for explanation [26].

### 3.4.1 Minimum Sum Algorithm

Minimum Sum algorithm (MSA) works by passing messages or beliefs between the check nodes and variable nodes of the tanner graph as shown in Fig. 3.7. MSA operates in logarithm domain and messages passed between nodes known as Log Likelihood Ratios (LLRs) are given by preceding receiver stage. LLR is similar to Eq. 3.18 and for \( 1 \leq i \leq n \), it is given by,

\[
\Lambda(x_i) = \log\left(\frac{P(x_i = 0|y_i)}{P(x_i = 1|y_i)}\right), \quad -\infty \leq \Lambda \leq \infty \] (3.19)

From Eq. 3.19, LLRs exhibit the property that as \( \Lambda \to \infty \), the probability of 0 being a correct value is 1 and \( \hat{x} \) is decided as 0. Similarly as \( \Lambda \to -\infty \), \( \hat{x} \) is decided as 1. At \( \Lambda = 0 \), \( \hat{x} \) can be equiprobable.

Let us consider a tanner graph shown in Fig. 3.7 where \( C(v) \) denotes the set of check nodes which connect to variable nodes \( v \). Similarly let \( V(c) \) denote the set of variable nodes that connect to check nodes \( c \). Let \( C(v) \setminus c \) represent all members of set \( C(v) \) except \( c \), while \( V(c) \setminus v \) represent all members of set \( V(c) \) expect \( v \). The MSA decoding algorithm for \( j \) iterations can be summarised in following steps.

1. **Initialisation**: Each codeword symbol \( y \) represented by the variable nodes in the

---

\(^1\text{The time required for solving NP-complete problems increases very quickly as the size of the problem grows. Not a desirable property for decoding codes with large length } n.\)
Figure 3.7: Generalised representation of Tanner graph

tanner graph are initialised by *a priori* LLR. For each variable node $v$, send the messages

$$\Lambda^{(0)}_{v \rightarrow c} = \Lambda^{(0)}$$

2. **Check node update**: For each check node $c$ and for each $v \in V(c)$, compute

$$\Lambda^{(j)}_{c \rightarrow v} = (\prod_{v' \in V(c) \setminus v} \text{sign}(\Lambda^{(j-1)}_{v' \rightarrow c})) \times \min_{v' \in V(c) \setminus v} |\Lambda^{(j-1)}_{v' \rightarrow c}|$$  \hspace{1cm} (3.20)

3. **Variable node update**: For each variable node $v$, send the message

$$\Lambda^{(j)}_{v \rightarrow c} = \Lambda^{(0)} + \sum_{c' \in C(v) \setminus c} \Lambda^{(j)}_{c' \rightarrow v}$$  \hspace{1cm} (3.21)

and compute

$$\Lambda^{(j)}_{v} = \Lambda^{(0)} + \sum_{c \in C(v)} \Lambda^{(j)}_{c \rightarrow v}$$  \hspace{1cm} (3.22)

4. **Decision**: Obtain intermediate $\hat{x}_i = 0$ if $\Lambda^{(j)}_{v}(x_i) \geq 0$ and $\hat{x}_i = 1$ otherwise. Check for the condition $H \cdot \hat{x}^T = 0$. If the condition is satisfied $\hat{x}$ is a valid codeword and $\hat{s}$ can be extracted, else goto step 2 and iterate until iteration limit is reached.

Careful observation of the algorithm at step 2 reveals that the check node computation of a given check node is independent of others in a given iteration. This property can be used to parallelise all the check node computations in a given iteration. This parallelism can also be seen in the variable node computation at step 3. Hence, in an ideal case, all the check nodes and variable nodes can be made to run in parallel with messages passing between them.
3.4.2 Reduced Minimum Sum Algorithm

The variable node update stage at step 3 in section 3.4.1 can be rewritten as

$$\Lambda^{(j)}_{v \rightarrow c} = \Lambda^{(j)}_v - \Lambda^{(j)}_{c \rightarrow v}$$  \hspace{1cm} (3.23)

The variable node message $\Lambda^{(j)}_{v \rightarrow c}$ can be computed from $\Lambda^{(j)}_v$ and $\Lambda^{(j)}_{c \rightarrow v}$, allowing us to merge step 2 and 3, where $\Lambda^{(j)}_{c \rightarrow v}$ can be directly computed from $\Lambda^{(j-1)}_v$ and $\Lambda^{(j-1)}_{c \rightarrow v}$. We can rewrite MSA algorithm as follows.

1. **Initialisation**: Each codeword symbol $y$ represented by the variable nodes in the tanner graph are initialised by a priori LLR. For each variable node $v$, assign

   $$\Lambda^{(0)}_v = \Lambda^{(0)}$$ and $$\Lambda^{(0)}_{c \rightarrow v} = 0$$

2. **Check node update**: For each check node $c$ and for each $v \in V(c)$, compute

   $$\Lambda^{(j)}_{c \rightarrow v} = (\prod_{v' \in V(c) \setminus v} \text{sign}(\Lambda^{(j-1)}_v - \Lambda^{(j-1)}_{c \rightarrow v'})) \times \min_{v' \in V(c) \setminus v} |\Lambda^{(j-1)}_v - \Lambda^{(j-1)}_{c \rightarrow v'}|$$  \hspace{1cm} (3.24)

3. **Variable node update**: For each variable node $v$, compute

   $$\Lambda^{(j)}_v = \Lambda^{(0)} + \sum_{c \in C(v)} \Lambda^{(j)}_{c \rightarrow v}$$  \hspace{1cm} (3.25)

4. **Decision**: Obtain intermediate $\hat{x}_i = 0$ if $\Lambda^{(j)}_v(x_i) \geq 0$ and $\hat{x}_i = 1$ otherwise. Check for the condition $H \cdot \hat{x}^T = 0$. If the condition is satisfied $\hat{x}$ is a valid codeword and $\hat{x}$ can be extracted, else goto step 2 and iterate until iteration limit is reached.

The rearrangement of the equations makes this algorithm perform faster and has less memory footprint than the original algorithm. This algorithm is henceforth referred to as Reduced Minimum Sum Algorithm (RMSA) [27]. The original algorithm needs to store $n \times d_v$ items of variable nodes to check node messages ($\Lambda_{v \rightarrow c}$), $d_v$ being average variable node degree i.e. average weight of the columns of parity check matrix $H$. In contrast, the reduced form is independent of variable nodes to check node messages and uses only $n$ items of variable node messages ($\Lambda_v$) for storage.
Memory footprint is further reduced as only addressing from check nodes to variable nodes needs to be stored unlike the original algorithm where addressing from variable nodes to check nodes are also stored. However, the major disadvantage of this algorithm is that variable node update stage cannot be parallelised anymore due non-sequential memory access which depends on adjacent variable nodes. As both algorithms are equivalent, there is no change in the performance of the algorithm. Hence in situations where memory footprint is more important than throughput performance, this algorithm can be used.

3.4.3 M kernel

The $H$ matrices of DVB-T2 LDPC codes have other properties beyond being IRA type that aid in further reduction of memory footprint. Some periodicity constraints were put on construction of the $H$ matrix allowing significant reduction on storage requirement without degrading performance. The $H$ matrix can be rearranged and be divided into sub-matrices of size $M \times M$, such that each sub-matrix consists of identity matrices which are either shifted or cyclically shifted. Certain sub-matrices have two diagonals in them and the factor $M = 360$ is common to all rates of LDPC matrices for DVB-S2, DVB-T2 and DVB-C2 standards. The parity check matrix of rate 1/2 DVB-T2 LDPC code characterised by sparse matrix of size $32400 \times 64800$ is shown in Fig. 3.8. Each blue dot in the figure is the position of bit 1 in the matrix. Fig. 3.9 gives conceptual representation of a sub matrix for size $360 \times 360$ for a DVB second generation LDPC code. Note that many sub matrices are empty showing the sparse nature of the matrix.

Let us consider a parity check matrix $H$ with size $(n - k) \times n$. As shown in section 3.3, the matrix can be arranged in form of $[A|B]$ of size $(n - k) \times k$ and $(n - k) \times (n - k)$. The $H$ matrix can be divided into $M \times M$ sub-matrices with $Q_b \times Q$ sub-matrices in row and columns respectively, where $Q = \frac{n}{360}$, $Q_b = \frac{(n-k)}{360}$, $Q_a = \frac{k}{360}$ and $Q = Q_a + Q_b$. Of all the sub-matrices, there are at most $\max(w_r) \cdot Q_b$ sub-matrices that are non-zero. This is because a single row in an irregular sparse matrix can have a row weight of at most $\max(w_r)$, where $w_r$ is the row weight of each row of the sparse matrix. As the sub-matrices are a collection of these elements and are identity matrices which are either shifted or cyclically shifted, there can only be at most $\max(w_r)$ of such non zero sub-matrices in a given row span of 360 rows. These non-zero matrices can be represented in a form of a tuple $(\mathcal{C}, \mathcal{D})$, where $\mathcal{D}$ gives the
amount by which an $M \times M$ identity matrix needs to be either shifted or cyclically shifted to get the sub-matrix indexed by $C$. Let $\mathcal{T}$ be a matrix consisting of tuples $(C_{p,q}, D_{p,q})$ that represent every non-zero sub-matrix of $H$ as shown,

$$
\mathcal{T} = \begin{bmatrix}
(C_{1,1}, D_{1,1}) & \cdots & (C_{1,q}, D_{1,q}) \\
\vdots & \ddots & \vdots \\
(C_{p,1}, D_{p,1}) & \cdots & (C_{p,q}, D_{p,q})
\end{bmatrix}, \quad 1 \leq q \leq \max(w_r), 1 \leq p \leq Q_b \tag{3.26}
$$

From Eq. 3.26, the column index $j$ of every $l^{th}$ non-zero element at row $i$ can be found out by,

$$
j = \begin{cases} 
C_{\lceil i/360 \rceil, l} \times 360 + \left[ D_{\lceil i/360 \rceil, l} + (i \text{ mod } 360) \right] \text{ mod } 360 & \text{if } C_{\lceil i/360 \rceil} < Q_b \\
C_{\lceil i/360 \rceil, l} \times 360 - D_{\lceil i/360 \rceil, l} + i \text{ mod } 360 & \text{otherwise}
\end{cases} \tag{3.27}
$$

where $1 \leq l \leq \max(w_r)$, $1 \leq i \leq (n - k)$.

As an example, let us consider a rate 1/2 DVB-T2 matrix characterised by the parity check matrix shown in Fig. 3.8. The matrix after rearranging is shown in Fig. 3.10. For this matrix, $n = 64000$, $k = 32400$, $\max(w_r) = 7$, $Q = 180$ and $Q_a = Q_b = 90$. Let
us consider the second row of $T$ with elements,

$$T = (10, 310), (10, 299), (20, 74), (37, 0), (41, 206), (90, 0), (91, 0)$$

The first element of the tuples give the column indices of the non-zero sub-matrix and those elements less than $Q_a$ belong to information matrix while others belong to the parity matrix. The second elements give the amount they are shifted or cyclically shifted downwards. The sub-matrices belonging to information matrices are cyclically shifted while the sub-matrices belonging to parity matrices are shifted downwards. Eq. 3.27 takes care of shifting and rotation. Note that the column index 10 is repeated twice. This indicates that the column $10 \times 360 + 1 = 3601$ to $10 \times 360 + 360 = 3960$ has two diagonals shifted downwards 310 and 299 times respectively. The two diagonals are separated out for the ease of representation and calculation.

The tuples belong to the second row of $T$, hence they represent sub-matrices with row indices $361 \leq i \leq 720$. Using Eq. 3.27, we the get indices of a non-zero element $j$ for $i = 361$ as $3911, 3900, 7275, 13321, 14967, 32401, 32761$. Storing $T$ instead of
Figure 3.10: Rearranged matrix representation of rate 1/2 DVB-T2 matrix of size $32000 \times 64000$

$H$ and calculation of nodes during the check node and the variable node computation stage of the RMSA algorithm reduces the memory footprint further, but the computation burden reduces the throughput. The $M \times M$ structure can also be used to group check node units together to form a semi-parallel architecture with each kernel processing $360 \times 360$ matrix. Hence the name $M$ kernel.
Applications with high throughput performance use Application Specific Integrated Circuit (ASIC) or Digital Signal Processors (DSP) for their realisation. While ASICs give highest throughput among the two, they are associated with high non-recurring engineering cost, manufacturing cost, design time and are suited for high volume production only. DSPs on the other hand are general purpose ASICs that have architectures targeting typical signal processing applications. These platforms provide cost-effective solutions and low design time for signal processing applications, but applications with special requirements suffer from architecture imposed constraints. This calls for a custom design based on the requirements of the application.

Application Specific Instruction-set Processors (ASIP) provide a middle ground between ASICs and DSPs. They form a viable platform for the implementation of custom applications demanding high throughput and low cost design. Unlike DSPs, ASIPs are co-designed with software application that execute on the target and mitigates architectural limitation. However, designing a new processor is a time demanding task, especially the verification of processor correctness is a time consuming task. Similar problems are faced when optimising a given architecture according to custom needs for better performance or when the effects of optimisation needs to be seen.

Tool assisted design exploration alleviates this problem and in addition provides useful data regarding the cost of executing application in the terms of area, throughput and energy. This also allows the analysis and exploration of hundreds of designs with cost estimates that can prove useful during design process. Such automation decreases the non-recurring engineering costs and design time making them ideal for applications with special architecture designs. Transport triggered architecture (TTA) and its corresponding toolset, TTA based Co-design Environment (TCE) is motivated to provide
a tool assisted design exploration for designing of ASIPs.

4.1 Transport Triggered Architecture (TTA)

Very Long Instruction Word (VLIW) processors were introduced to exploit the instruction level parallelism (ILP) i.e. executing more than one instructions in parallel. A VLIW architecture consists of several processor units which execute large instruction words fetched from instruction memory or cache in parallel. The processor unit design is kept simple by shifting the complexity of instruction scheduling and parallel dispatch to the compiler. This makes processor perform faster and keep the overall hardware cost to a minimum. The basic instruction set is kept simple as in RISC architectures and the compiler groups these simple instructions together so that multiple instruction units can be kept busy at the same time. Thus, the compiler has the responsibility to come up with a schedule from a given application and rearrange it for parallel execution.

VLIW architecture has a significant dependency on the compiler for its performance. Recent advances in compiler technology and general advances in semiconductor technology has alleviated this problem. However, VLIW suffers from other architectural limitations such as increased code size arising from aggressive scheduling policies, large memory bandwidth and register file size. Operations requiring one or two instructions under utilise the communication bandwidth, and the complexity of communication network itself increases with large number of processors. Further, addition of new processor units or functional units is difficult and requires redesign of the architecture. These issues led to the design of TTAs [9].

TTAs change the programming paradigm from *operation triggering* to *transport triggering*, where operations are a consequence of data transports between various functional units or processing units. TTAs can be seen as super class of VLIW which exploit parallelism not only at instruction level, but also at data level. TTA extend the concept of RISC architecture by further reducing the instruction set to a single instruction where other operations are a consequence of move instruction and makes data transports completely visible in the architecture.
4.1.1 Hardware Architecture

A TTA processor can be depicted as set of Functional Units (FU) interconnected through a network. The functional units may be units which perform arithmetic or logical operations, Register Files (RU) for storing data, Special Functional Units (SFU) for custom operations and units for fetching and storing data from data and instruction memory. The functional units may consist of one or more input and output ports. The network consists of buses and sockets which form connection path between various ports of the FU. The processor has separate instruction and data memory. The general architecture of a TTA processor is shown in Fig. 4.1.

Functional units operate on data and they receive it through input ports and send the data through output ports. There is no limitation on the number of input and output ports a FU can have and is decided by the operation a FU performs. For instance, a FU which performs addition may have two input ports corresponding to two input data and one output port corresponding to the result. The FUs can also be designed to perform more than one operation, in such cases one of the input port may be dedicated to opcode to select operation in the FU. However, one input port in every FU is designated as trigger port that triggers the operation on data whenever data is written to it.

The architecture of FU with one trigger port (T), one input port (I) and one output port (O) is shown in Fig. 4.2. The FUs are internally pipelined and the data as well as opcodes can be stored at the previous cycles. The FU triggers whenever data is written to the trigger port. The FUs consists of an internal combinatorial or sequential logic that runs synchronously with the instruction stream specified by the compiler. The FU
advances one step every time an instruction is issued.

The consequence of internal pipeline in a FU is that it allows the results to be stored until the next usage of the FU. This eliminates the need of storing all the results into a register file as they can be passed directly to the next FU. Also several operations does not generate results for the registers, eg: jump and calls. This has motivated the design of Register File (RF) as a FU. A RF can have multiple but limited number of read and write ports so that multiple FUs can read and write at the same clock cycle. The utilisation of a RF is determined by the compiler at the compile time reducing the complexity of the RF. Special Functional Units (SFUs) can be easily added with custom operations such as multiply and accumulate as they behave like any other FUs. A dedicated FU called Load Store Unit (LSU) is used to fetch data from data memory and Instruction Fetch Unit (IFU) is used to fetch data from instruction memory.

All the ports originating from and terminating to the FUs are connected to the interconnection network through the sockets. An input socket consists of multiplexors which feed data from the buses to the FUs while an output socket consists of de-multiplexors setting results of the FUs on the buses. Sockets provide a convenient way to form the network with buses as it is not mandatory that a given socket should connect with every available bus. Input sockets are tagged using destination IDs. A socket connecting four move buses can have four destination IDs. These destination
IDs are compared in the socket and if the IDs match to that of the socket, then a corresponding socket is selected and data is moved to the FU. Output sockets work in similar fashion but with source IDs instead and they set results on the buses.

### 4.1.2 Software Design

Operations in TTA occurs as a side effect of data transport unlike RISC where operations to be performed are specified in the instruction. An operation in TTA typically consists of moving an operand data to the FU, triggering the FU and transporting the results from the FU. These moves are classified into three categories: *operand*, *trigger* and *result* moves. *Operand* moves are responsible for moving input data of an operation to the operand register of a FU, *trigger* moves also transport input data to the FU, but doing so triggers operation. Consequently, *operand* moves should either precede or should take place in the same cycle as the *trigger* move. *Result* moves transport final data from a FU to another FU or a RF and takes place after the *trigger* move.

The operations can be understood better by comparing it with a typical RISC instruction. An add instruction, where data from register *r1* and *r2* is added and stored in register *r3* in a RISC platform is given by

```plaintext
add r3, r1, r2;
```

Listing 4.1: ADD instruction in a RISC platform

While the same operation on a TTA architecture is shown in the listing 4.2. The first instruction is of an *operand* move type where data from register *r1* is moved to *in1* port of the FU *add*. The second instruction is of a *trigger* move type as the movement is to the trigger port of FU while the final instruction is of a *result* move type. The trigger port is suffixed with ‘t’.

```plaintext
r1 -> add.in1;
r2 -> add.in2t;
add.out1 -> r3;
```

Listing 4.2: ADD instruction in a TTA platform

The instruction format, shown in Fig. 4.3, consists of immediate extension tag, move slots and immediate extension field. There are as many move slots as there are
buses and the Fig. 4.3 shows instruction format for two move buses. Each move slot consists of guard field (grd) which allows the FU to either delay the operation using *global lock* or cancel the operation using *squashing*. The destination and source IDs are used to select appropriate sockets on each bus that transfer the data. Each move slot can also be used to represent immediate value extensions, in such cases, sockets are not allowed to decode and instead all IDs and guards form a long immediate. The immediate extension tag is used to determine which move slots are used as immediate instruction and which are used as data transports. The immediate extension field is also available for the construction of long immediate instructions.

TTA allows pipelining in two levels: the first level inside a FU, as discussed in the previous section and the second at an instruction level. A three-stage pipelining is typically provided at an instruction level where three stages consists of instruction fetch (IF), decode (DC) and move (MV). The pipeline is shown in Fig. 4.4. The instruction fetch stage fetches instruction from the cache or the memory, decode stage transports destination and source IDs to the sockets where it is decoded, while in move stage the actual transportation of data between buses and FUs take place. As the functional units can be pipelined, the execution adds latencies in the pipeline shown in Fig. 4.4. The decode and move instruction can be combined into a single decode-move (DC-MV) instruction to provide a two stage pipeline.

### 4.1.3 Computation Example

A detailed understanding of TTA can be obtained by a computation example. Let us consider a typical RISC instruction shown in listing 4.3 with no preference to the write
Figure 4.4: TTA instruction pipeline a) Example of a three stage pipeline b) Example of operation with latency of three order.

Listing 4.3: Example set of RISC instructions

```
READ_IO(a);
READ_IO(b);
c = a*a + b*b;
d = a + b*b;
WRITE_IO(c);
WRITE_IO(d);
```

A TTA architecture with an adder, a multiplier, a RF unit and an Input and Output (IO) functional unit connected to a single transport bus is considered for this example. The architecture is shown in fig. 4.5. The trigger port of individual functional unit is depicted with an asterisk on respective ports. The IO FU behaves similar to a RF unit where port 0 is used to read data and port 1 is used to write data to external IO. The instructions for both RF and IO units specify the address to be read or written into. The instructions can be translated into TTA instruction format as shown in the listing 4.4.
Figure 4.5: TTA machine with single bus

The instructions are executed in sequential order, starting with moving data from IO functional units to register files. The data is then moved to the multiplier first and then to the adder. The resultant data from the adder is moved to the IO function unit. This functions are repeated for both output 'c' and 'd'.

Listing 4.4: Sequential TTA instructions

Ignoring the initial fetch and decode cycles of the pipeline and assuming single cycle execution of the instruction, the above program executes in 14 cycles. The FUs can store operand data in their internal register, hence the program can be optimised by avoiding data movement to the RF unit and moving data directly to the FU whenever
possible. The execution of the instruction can be further increased by using parallel data movement. Let us consider the same architecture, but with three transport buses to support parallel data movement. The architecture with multiple bus is shown in Fig. 4.6.

Table 4.1 gives the assembly code for parallel execution. Short immediate operations are used to move constants to the FU. The immediate operands are determined at compile time and is stored in the instruction memory. Addition of multiple path for data movement has decreased the execution time to 5 cycles. It can be seen that the utilisation of all the three buses are not constant. Addition of another bus will not improve the performance any further, but instead may introduce delay due to extra overhead needed to address the bus. The RF unit is not used and the data is directly transferred between the units. Also, one can also notice that not all the three buses need connection to all the FUs. The connection can be minimised reducing the power consumption of the hardware. The optimised hardware is shown in Fig. 4.7.
The program execution can be further accelerated by introducing a custom execution unit. The custom unit is designed with two input ports and two output ports to accept inputs $a$ and $b$ and output results $c$ and $d$. The unit itself can be implemented in numerous ways ranging from simple combinatorial circuit consisting of adders, multipliers and multiplexors to processors as long as the interfaces meet the TTA requirements. The custom unit can also be a TTA processor, providing a scope for a hierarchical design. However, this requires a toolchain that provide separation between hierarchies and generate instructions for each processor. Such design also requires a hardware platform that provides capabilities to program individual instruction memories. The TTA architecture with customised unit with single bus and an IO unit is shown in Fig. 4.8.

The code listing 4.5 above gives the program for the custom TTA architecture. It is assumed that the custom unit takes single cycle to produce results. If the execution time of the custom unit is more than a cycle, no-operation instructions are introduced.
It can be seen that at the cost of hardware design time and complexity, the program execution time is decreased to 4 cycles. Although there is no substantial increase in performance when compared to a multi-bus case, this example provides the ease at which customised hardware units can be interfaced with rest of the processor.

4.2 TTA based Codesign Environment (TCE)

The flexible and customisable feature of TTAs can be exploited using TTA based Codesign Environment (TCE) toolset [10]. The toolset is developed at Tampere University of Technology and is released as an open source. The toolset encompasses numerous commandline and graphical utilities providing designers the capabilities ranging from hardware design exploration, software compilation to cycle accurate simulation. The structure and design flow of the tool will be discussed here.

4.2.1 TCE structure

The structure of the toolset is shown in the Fig. 4.9 which gives the utilities and data formats used. The toolset can be divided into three primary groups: hardware generation, program compilation and instruction simulation. A custom processor can be designed from existing FUs using ProDe, ProGe, tcecc, PIG and ttasim utilities.

The designer specifies the functional units of the ASIP and their interconnects in a graphical tool called Processor Designer (ProDe). ProDe then builds an Architectural Definition File (ADF) with these specification. The ADF file along with a program described in a high level language forms input to the TCE compiler (tcecc). The compiler is a retargetable LLVM-based compiler that can compile programs in C and C++. The output of tcecc is an assembly language object in TTA Program Exchange Format (TPEF). The Program Image Generator converts such an object file into an instruction memory that is ready for deployment. An Implementation Definition File (IDF) is also
created by ProDe which references the implementation of each FU in the *Hardware Database* (HDB).

The HDB is a SQL-based database that contains entries of *Hardware Description Language* (HDL) description of the functional units. There may be several implementations satisfying the same architecture interface to meet the needs such as low-power design or ones tailored for specific synthesis target. The processor architecture described in ADF is converted into a HDL with the help of IDF by *Processor Generator* (ProGe) utility. ProGe can be configured either to integrate processor to a target such as Field Programmable Gate Arrays (FPGAs), or generate scripts to run on simulators such as GHDL, Modelsim etc. For a designer interested in designing custom hardware
units as hardware accelerators, the toolset provides *Hardware Database Editor* (HDB) to modify the entries in HDB. Information about the HDL entity needs to be filled in, such as the entity name and the names of each input, output and clock signal, so as to allow ProGe to automatically generate a HDL description of the interconnection network which incorporates the unit.

The hardware machine and the program can be simulated using the instruction set simulator *ttasim* or by its graphical wrapper *Proxim*. The simulator takes ADF and TPEF file as input and simulates with an accuracy of an instruction cycle. The simulator assumes single cycle fetch and execution of the instructions from instruction memory and does not takes memory stalls into account. To introduce custom instructions associated with hardware accelerators, *Operation Set Editor* (OSEd) is used. The editor can make changes to *Operation Set Abstraction Layer* (OSAL), that stores the instructions used in TTA processors. OSAL lists the name, operand count and properties of each instruction. Properties could include simulation time in cycles and *trigger semantics* i.e. legal replacements of an operation with a combination of other operations. For example, a *less-than-or-equal* operation could be computed as *not greater-than*, or $a + b$ as $b + a$ when it is convenient. The simulator behaviour of each operation is defined by writing a C++ function that performs the operation.

For designs targeting reconfigurable logic such as FPGAs, TCE toolset provides the *Platform Integrator* tool for effortless ASIP design flow. The tool provides two main functionalities. The first one is to integrate TTA processor right into FPGA board as a standalone processing unit. The second option is to wrap TTA processor into an IP block which can be used in system on chip designs. The *Platform Integrator* currently supports only *Altera* FPGA boards. The toolset and FPGA boards are discussed in section 4.3.

### 4.2.2 Design Flow

The design flow of TCE consists of four phases that tie different utilities and data formats of TCE together to create and deploy TTA machines. The first phase of the design flow, *Initialisation Phase*, provides input sequential program and initial processor architecture to the design flow as shown in Fig. 4.10. The initial sequential code input to TCE is generated by a 3rd party frontend compiler such as LLVM, and is not shipped with TCE. Program described in multiple compilation units can be linked together by TPEF linker to produce a single TPEF object. ProDe tool is used to design
either the starting point architecture for design exploration, or the final target architecture for scheduling.

In Design Exploration Phase, the tool assists the designer in discovering optimal processor configuration. It is a process in which several variations of an user-defined starting point architecture are simulated and costs are estimated. The exploration is semi-automatic and alternatively, the designer may chose manual design exploration. The manual mode requires designer to run instruction scheduler and instruction set simulator manually for initial architecture. The statistics produced after exploration can help in modifying architecture to suit the needs of the application.

The design exploration phase is shown in Fig. 4.11. This phase is semi-automatic as it needs an initial architecture configuration from the designer. The Explorer removes resources from the initial architecture and sends the modified architecture to the code generation and analysis phase. A Cost Estimator uses a predefined processor cost database and estimates the cost of executing the given program in the modified architecture, in terms of physical area of processor, energy consumption and maximum speed. Explorer then finds an optimal processor architecture after evaluating many variations of processor modifications. Sequential simulation is performed once per design exploration to provide profiling information for the instruction scheduler. The tool automatically invokes parallel simulation and outputs ADF, IDF for each variation of the target architecture. The phase culminates by providing the designer a list of
characteristics of all explored processor configurations to choose from.

The Code Generation and Analysis phase, shown in Fig. 4.12, is the most demanding and important part of TCE design flow. The scheduler or compiler backend converts sequential programs to parallel program to utilise the given architecture. This makes manual programming of TTAs redundant and assists the designer in semi-automatic design exploration. The analysis part of this phase includes active use of ttasim and Proxim for the simulation of scheduled program to obtain data about processor utilisation that can be used in cost estimation.

Processor and Program Image Generation phase forms the final phase and is shown in Fig. 4.13. This includes the generation of HDL files and processor compatible binary of scheduled programs for the designed TTA processor using ProGe utility and Program Image Generator respectively. The synthesis of the processor and porting of design on ASICs or any other platform is out of the scope of TCE.
4.3 Custom ASIP design on FPGAs

Designing a custom operation which is able to meet a specific requirement is one of the primary goals of TCE toolset. The toolset increases the design productivity by increasing the design abstraction and separating the custom operation architecture and design implementation. This separation allows the custom operations to be tested and evaluated without the need of the RTL implementation of the custom operation. The design flow of custom operation is shown in the Fig. 4.14. A custom operation candidate forms the input of the design flow. The candidate is chosen from analytics obtained after profiling the application. Valuable insights can also be obtained by analysing the data dependency graphs extracted during the compile stage. After the candidate is
chosen, Operation Set Editor (OSEd) is used to create a custom operation definition which includes only the name of the operation and the number of inputs and output operands. The simulation model is created using high level languages such as C and C++ and it usually involves functions from the main code that needs to be accelerated.

The custom operation is wrapped in a functional unit and is added to the processor architecture. The operation latency is defined at this stage. The toolset gives options to change the latency during the iterations if the latency is not known during the initial stages of design. The custom operation is utilised in the software by calling the operation via TCE specific operation macros or intrinsics. The custom operation is then compiled and simulated. The improvement on execution count can be obtained from simulations and the operations can be improved based on these metrics.

Once the custom operation is decided to be included in the processor, an implementation of this is required. This requires the designer to write Register Transfer Level (RTL) code in hardware descriptive language (HDL) such as VHDL (Very High speed integrated circuit Hardware Descriptive Language) to target an ASIC or reconfigurable circuits such as Field Programmable Gate Arrays (FPGAs).
4.3.1 Field Programmable Gate Arrays

Field Programmable Gate Arrays are integrated circuits with reprogrammable logic arrays that enable configuration of logic network after its manufacture, unlike Application Specific Integrated Circuits (ASIC). The capability of being field programmable is obtained due to its composition of logic elements or arrays. The structure of logic elements is specific to the vendor and the device family. This work uses the Stratix III device family from Altera FPGA vendor. The architecture of the Stratix III device family is shown in Fig. 4.15 [28]. The device consists of arrays of macroscopic building blocks called logic array blocks (LAB) (not shown in figure) each composed of basic building blocks known as adaptive logic modules (ALMs) that can be configured to implement arithmetic, logical and register functions. Specifically, each LAB consists of ten ALMs, carry and arithmetic chains for arithmetic functions, register chain connection lines between various registers within ALMs and interconnect and control lines between ALMs.

The ALMs contain a variety of lookup table based resources that can be divided between two combinatorial adaptive lookup tables (ALUTs) and two registers with up to eight inputs. In addition, each ALM consists of two programmable registers, two dedicated full adders, a carry chain, a shared arithmetic chain, register chain that can implement arithmetic and logic operation on any combinations of two functions. Each ALM drives all types of interconnects: local, row, column, carry chain, shared arithmetic chain, register chain and direct link interconnects. A high-level block diagram of Stratix III ALM is shown in Fig. 4.16 [29].
Figure 4.16: High level block diagram of adaptive logic modules (ALM)

The device also consists of embedded memory blocks in three different sizes: 320 bit memory logic array blocks (MLABs), 9-Kbit M9K blocks, and 144-Kbit M144K blocks. Each memory block can independently be configured to be a single- or dual-port RAM, FIFO, ROM or shift register. Multiple blocks of the same type can be stitched together to form a bigger block with minimal timing penalty. The MLABs have been optimised to implement filter delays, small first-in-first-out (FIFO) buffers, and shift registers. The M9K blocks are used for general purpose memory applications, and M144K blocks are ideal for processor code storage, packet buffering, and video frame buffering [30].

The logic elements of FPGA are connected together by a dedicated interconnection network. The logical elements are driven by a hierarchical clock networks, multiple phase-locked loops (PLLs) and delay-locked loops (DLLs) forming a complete clock management system. PLLs provide option to program unique, customisable clock frequency with inherent jitter filtration and fine granularity control over multiply, divide ratios and dynamic phase shift reconfiguration capabilities. A DLL provides a process, voltage and temperature compensated delay that can be used to phase shift the read clock from an external memory to align it with the center of the data valid window.

External signals are extracted and applied through I/Os arranged in I/O banks for easy configurability. Numerous I/O features assist in high-speed data transfer into and out of devices, few including single-ended, non-voltage, and voltage referenced
I/O standards; programmable output current strength, slew rate, delay, bus hold and pull up resistor; serial, parallel, dynamic and differential on-chip termination among many other features. The I/Os also support a wide range of industry I/O standards for easy interfacing between variety of devices [31]. The logic elements, interconnection, clock circuitry and I/O modules are packaged in a single die for various speed grades, commercial and industry grades with varying pin-counts [32]. The reconfiguration ability of FPGAs provide an excellent platform to prototype ASICs before manufacture and is extensively used in this work to test TTA machines and evaluate TCE toolset.

### 4.3.2 Integrating with FPGA

The custom operation is then integrated with the target platform using Platform Integrator toolset. Prior to integration, the TTA unit tester tool can be used to test if
the output of the RTL code is equal to its simulation model. The custom operation can also be added to a hardware database with the hdbeditor tool for later design reuse. The rest of the design flow is shown in Fig. 4.17. The design flow is the concise representation of the flow discussed in section 4.2.2. The design flow discussed here includes the Platform Integrator to integrate custom operations and FPGA synthesis tool to synthesise the TTA processor on an FPGA fabric.
5 Decoder Design

The design and exploration of the processors for the LDPC decoder were carried out in the version 1.6 of the TCE toolset. The processors were ported on DE3 FPGA development board, shown in Fig. 5.1 [33]. The development board houses a Stratix III (EP3S150F1152C2) FPGA clocked at 50 Mhz. The FPGA consists of 142,000 logic elements, 6,390 Kbits of total embedded memory, 384 $18 \times 18$ multiplier blocks and 8 phase lock loops. It also consists of a SO-DIMM slot that can be used to mount an external DDR2 RAM of size up to 4 GB. A 128 MB DDR2 RAM was used in the current setup. Version 13 of Quartus II integrated development environment (IDE) was used to synthesise the HDL code from TCE toolset and port it on FPGA. The FPGA is configured using built in USB blaster circuit for programming and user API control.

5.1 Design procedure

For the design of the processors, five different parity matrix candidates were chosen to ease the development and verification process. Hamming $(7,4)$ code with the parity matrix shown in Fig. 3.5 and a $8 \times 12$ parity matrix were chosen to help development process. A $1000 \times 2000$ parity matrix was chosen to check the functionality and scaling ability of the decoder on a medium sized parity matrix. A rate $\frac{1}{2}$ LDPC code of frame size 16,200 bits and 64,800 bits specified by the second generation DVB standards were used. The second generation DVB standard specify six different code rates for LDPC listed in table 2.1, for short frame size of 16,200 bits and long frame size of 64,800 bits. Among all the different code rates, a rate $\frac{1}{2}$ LDPC code was chosen because it represents largest parity check matrix of size $9000 \times 16200$ for the short frame size and $32400 \times 64800$ for the long frame size. The results shown here only include parity check matrices of size $9000 \times 16200$, henceforth called as $DVB-T2$ short and $32400 \times 64800$, henceforth called as $DVB-T2$ long, and the remaining matrices were used only for development process to check the functionality on a smaller scale.
Figure 5.1: Overview of the DE3 FPGA development board
The parity check matrix is represented using *alist* format, a data structure that consists of size of the matrix, biggest row and column weights, lists of weights of each row and column and a lists of row and column indexes which are connected to each other [34]. Appendix A.1 lists the structure of the alist format.

Simulations were performed as follows. Random data was first encoded into a code word with LDPC code. Care was taken to avoid messages with all zeros or all ones for simulation. The code word was passed through an AWGN channel simulator which corrupted the data. LDPC decoding was then performed in an attempt to recover the original data. The decoded data was compared with the source data in order to check the correctness of the decoder. An open source library was also used to implement source message generation, encoding and channel simulation [35]. The decoder output was also simultaneously verified with the output given by the decoder of the open source library to verify the correctness of the implementation.

The performance of each processor was estimated based on its throughput and area consumption. The throughput was measured for worst case performance for all the decoder algorithm i.e. the step 4 of MSA and RMSA algorithm described in section 3.4.1 and 3.4.2 was modified such that the algorithm iterated for the maximum number of iterations defined. The maximum iteration limit affects the error correction performance of the decoder expressed in terms of the bit error rate (BER) vs. signal to noise ratio (SNR) curve. Higher iteration limits provide better error correction capability, but comes at the cost of decreased throughput. A study of error correction performance for various iteration limits is carried out in [36]. For hardware implementations, an iteration limit between 20-50 provide significant error correction performance. As such the design of the TTA processor is based on the throughput relative to other TTA processor implementations, an iterative limit of 20 was chosen taking simulation time into consideration without hurting error correction performance of the decoder.

The area consumption of the FPGA is measured in terms of utilisation of logic, memory, input output pins, PLLs/DLLs and other FPGA specific resources. Distilling these into a single metric is a difficult problem as overshooting the utilisation of a single resource makes the porting of a HDL code on a FPGA an impossible task. For instance, considering the memory resources, Altera architecture does not combine M144K blocks with M9K blocks to form a single block of memory. Hence overshooting the utilisation of M9K or M144K memory block renders the entire HDL code unportable on a FPGA even if other resources are available. This problem has been
seen during the TTA processor design process. However, during design process, it was also seen that only few parameters, such as logic and memory utilisation particularly utilisation of M9K and M144K blocks varied. Hence processors were compared based on throughput and utilisation of logic, M9K memory blocks and M144 memory blocks.

To test the TTA processor on FPGA, the LLR values obtained from the receiver stage as well as the parity check matrix in alist format was stored on the on-chip memory of the FPGA. A timer was used on TTA processor to measure the decoder throughput with the resolution of 1 microsecond and the FPGA was clocked at 50 MHz. While the ttasim and Proxim simulator tools of TCE toolset assumes a clock frequency of 100 MHz, the simulated time scaled accurately to that executed on of FPGA. The accuracy is because the data and instruction memories are implemented on on-chip memories of FPGA with single cycle access and the simulator accurately models such systems. This property can be used to get an accurate idea of execution time on FPGA prior to porting on it. This property was also extensively used during the course of experimentation, as the author short circuited the FPGA board, and due to this all the processor architectures could not be tested on FPGA. The measurements were made again on the simulator assuming a clock rate of 100 MHz and was scaled down by a factor of two. The HDL code generated by TCE tool was synthesised for a frequency of 50 MHz to get an exact data on resource utilisation on FPGA.

5.2 Basic Architecture

The initial processor configuration for the decoder consists of an Arithmetic and Logic Unit (ALU), a Load Store Unit, an Input Output Unit (IOU), a boolean register file (BOOL), a timer (RTC), a jump and branch control unit (GCU) and a Register File (RF) connected to a single transport bus. The processor is shown in Fig. 5.2. The ALU consists of arithmetic operations such as 'add' and 'subtract', and logical operations such as 'and', 'or', 'xor', signed and unsigned shifts and cyclical shifts in left and right direction, and comparison operations. The LSU can fetch and store data from the data memory in word, half word, quarter word and byte sizes. The IOU is used to print the data to stdout. The BOOL is a $2 \times 1$ bits register while RF is a $5 \times 32$ bit register. The throughput rates for DVB-T2 short and DVB-T2 long parity matrices for MSA algorithm is given in the table 5.3.
From the FPGA utilisation data given in table 5.1, it can be seen that the processor does not fit in the FPGA. Excessive consumption of memory by the parity check matrices of DVB-T2 long and DVB-T2 short codes are the reason for this. To reduce the memory footprint, RMSA algorithm and M-kernel algorithm for decoding LDPC codes was employed. The M-kernel algorithm was only implemented to DVB-T2 long matrix. The device utilisation for the RMSA and M-kernel algorithm is given in the table 5.1.

Reduction of size of parity check matrices does not help in fitting the design on FPGA because of the non-uniform utilisation of the memory resources by the synthesis tool. By default, Altera tools does not combine M9K and M144K memory blocks together to form a single block of memory. Hence a custom memory architecture for combining M9K and M144K blocks was built. The need for custom memory architecture is device specific. FPGAs with larger memories may not require such architecture. The memory architecture is shown in Fig. 5.3. The architecture divides given address range of data memory into blocks of equal size and allocates them to M9K and M144K memory blocks respectively. The higher order addresses are implemented in M144K memory blocks, while the lower order addresses are implemented in M9K memory blocks. The switching is done using most significant bit of the address range. This architecture has resulted in efficient usage of memory resources on the FPGA, as seen in table 5.2. The RMSA decoding algorithm for DVB-T2 short matrix and M-kernel
algorithm for DVB-T2 long matrix now fits on the FPGA.

Alternative approaches including interfacing DDR2 RAM to the processor and designing a custom external memory to fit the design of the decoder on FPGA were also tried. As described in section 4.2.1, the simulator models code and data memory with access times of single clock cycle. The LSU unit of the TTA processor is also modelled for memories with access times of single clock cycle. The HDL code for the LSU can be modified to allow interfacing memories with access times of more than single clock cycle. But the access times should be deterministic and known. A wrapper needs to be built to interface LSU with DDR2 memory controller and this was left out as a future exercise. Designing an external SRAM memory with single or integer access cycle was also carried out. The SRAM memory was to be interfaced to the expansion header. But, during the testing phase, the author short-circuited the board, forcing to use simulation platform instead. However, combining memory blocks were
<table>
<thead>
<tr>
<th>Processor Configuration</th>
<th>DVB Short</th>
<th>DVB Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial processor (IP)</td>
<td>134.25</td>
<td>93.06</td>
</tr>
<tr>
<td>IP + 1 RF</td>
<td>280.08</td>
<td>194.66</td>
</tr>
<tr>
<td>IP + 1 Bus</td>
<td>212.35</td>
<td>147.13</td>
</tr>
<tr>
<td>IP + 1 RF + 1 Bus</td>
<td>476.17</td>
<td>331.01</td>
</tr>
<tr>
<td>IP + 1 mul</td>
<td>538.49</td>
<td>387.65</td>
</tr>
<tr>
<td>IP + 1 mul + 1 RF</td>
<td>1125.98</td>
<td>810.96</td>
</tr>
<tr>
<td>IP + 1 mul + 1 Bus</td>
<td>846.73</td>
<td>608.54</td>
</tr>
<tr>
<td>IP + 1 mul + 1 RF + 1 Bus</td>
<td>1709.50</td>
<td>1221.87</td>
</tr>
</tbody>
</table>

Table 5.3: Throughput rates in bits/second from various processor designs for decoding LDPC codes using MSA algorithm.

<table>
<thead>
<tr>
<th>Processor Configuration</th>
<th>DVB Short</th>
<th>DVB Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial processor (IP)</td>
<td>110.71</td>
<td>76.86</td>
</tr>
<tr>
<td>IP + 1 RF</td>
<td>290.25</td>
<td>200.60</td>
</tr>
<tr>
<td>IP + 1 Bus</td>
<td>220.23</td>
<td>151.73</td>
</tr>
<tr>
<td>IP + 1 RF + 1 Bus</td>
<td>498.34</td>
<td>344.66</td>
</tr>
<tr>
<td>IP + 1 mul</td>
<td>618.05</td>
<td>427.51</td>
</tr>
<tr>
<td>IP + 1 mul + 1 RF</td>
<td>1378.67</td>
<td>969.11</td>
</tr>
<tr>
<td>IP + 1 mul + 1 Bus</td>
<td>1017.34</td>
<td>704.49</td>
</tr>
<tr>
<td>IP + 1 mul + 1 RF + 1 Bus</td>
<td>2256.80</td>
<td>1593.49</td>
</tr>
</tbody>
</table>

Table 5.4: Throughput rates in bits/second from various processor designs for decoding LDPC codes using RMSA algorithm.

carried out prior to designing external memory and the M-kernel algorithm for DVB-T2 long code as well as RMSA algorithm for DVB-T2 short code was tested on FPGA with execution times scaling to that of the simulation.

The initial processor configuration was tested for RMSA and M-kernel algorithm as well and the throughput rates are given in table 5.4 and table 5.5. The initial processor configuration was explored further with the addition of a multiplier unit, a RF unit and a transport bus and the throughputs are recorded for all possible processor combinations. The comparison of the throughput rates among the decoding algorithm for different processor configurations are presented in Fig. 5.4 for the DVB-T2 short code and in Fig. 5.5 and several observations can be made. The RMSA and M-kernel algorithm is faster than MSA algorithm, despite the lack of parallelism of computation of variable node messages. Absence of processing variable node to check node messages ($\Lambda_{v\rightarrow c}$) and fewer memory access can be attributed to speed. The extraction of parity check
<table>
<thead>
<tr>
<th>Processor Configuration</th>
<th>DVB Long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial processor (IP)</td>
<td>86.40</td>
</tr>
<tr>
<td>IP + 1 RF</td>
<td>173.78</td>
</tr>
<tr>
<td>IP + 1 Bus</td>
<td>136.11</td>
</tr>
<tr>
<td>IP + 1 RF + 1 Bus</td>
<td>296.66</td>
</tr>
<tr>
<td>IP + 1 mul</td>
<td>397.45</td>
</tr>
<tr>
<td>IP + 1 mul + 1 RF</td>
<td>655.08</td>
</tr>
<tr>
<td>IP + 1 mul + 1 Bus</td>
<td>839.02</td>
</tr>
<tr>
<td>IP + 1 mul + 1 RF + 1 Bus</td>
<td>1406.19</td>
</tr>
</tbody>
</table>

Table 5.5: Throughput rates in bits/second from various processor designs for decoding LDPC codes using M-kernel algorithm.

matrix from the cyclically shifted sub-matrices adds computation load on decoding of M-kernel, and is slower compared to RMSA algorithm.

Few anomalies can be seen from the Fig. 5.4 and Fig. 5.5. The MSA algorithm computes faster than RMSA and M-kernel algorithm for the initial processor. The slower execution speed of RMSA and M-kernel algorithm can be attributed to the lack of RF resources for non-sequential memory access required for variable node messages. This can be seen from the table 5.6, which lists the execution counts on RF unit and bus for various operations for initial processor configuration and the same with an additional RF unit. The execution count of operations with a processor configuration consisting of RF unit is less than that of initial processor configuration, suggesting lack of availability of registers has resulted in the use of data memory.

Addition of transport bus enables delivering instructions in parallel, increasing the throughput. However, increasing the amount of transport bus also increases the size of instruction memory. Adding RF units has more pronounced impact on throughput when compared to addition of the transport bus, but comes at the cost of increased power consumption and increased resource utilisation on FPGA. A considerable increase in the throughput can be seen after the addition of a dedicated multiplier unit. The ALU does not support multiplication operation, and all the multiplication operations are done using addition and logical operations. The speed up of M-kernel decoding algorithm over RMSA decoding algorithm for DVB-T2 long matrix on the initial processor configuration with addition of a multiplication unit and a bus (IP + 1 mul + 1 bus) can be attributed to the lack of sufficient registers and access of fewer data memory location by M-kernel algorithm.
The exploration of the initial processor configuration can be continued by adding more units. Several combinations of units can be added to increase the throughput. At this stage, the automatic design exploration feature provided by TCE toolset was used. TCE toolset provides Explore plugins for searching target processor configurations. The Grow machine plugin can be used to add resources to the machine until cycle count doesn’t go down anymore. However in the current version of the tool, the plugin is restricted to addition of RF units only.

The Minimize machine, on the other hand, removes the units from the machine until real time requirements of the applications are not reached anymore. The processor configuration generated by the plugin is heavily dependent on the seed design. For an application as large as decoding LDPC codes, the seed design for the plugin depends not only on the size of the parity matrix, but also on the choice of the initial units. The choice of initial units for the seed design can be made by looking into the scheduling and execution profile of the code on TTA. This can help to spot repeated patterns for which custom units can be designed. After an exhaustive search, explore plugins can be used to further increase the throughput of the design.
5.3 Designing custom processor

A custom processor was designed with the goal of minimising the design time and maximising the throughput. This involved exploiting both instruction level and task level parallelism from the application. The design process involved searching for patterns in the application and two approaches were taken to design the custom processor. The first approach involved profiling the application to find regions of the code that is executed often. These regions of code is then converted into a custom processor unit and the throughput is measured. The regions where parallelism can be exploited is favoured and measurements are carried out.

The second approach involved exploiting task level parallelism inherent in the algorithm. The checknode stage of all decoding algorithms are parallelised and the throughput is measured. In addition, the instructions patterns of the application was also analysed using data dependency graphs obtained from the TCE compiler during scheduling. Repeated instruction patterns are then converted into custom processors to obtain speed up. The final processor includes the custom processors derived from the
<table>
<thead>
<tr>
<th>Sockets</th>
<th>IP</th>
<th>IP + 1 RF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus</td>
<td>32190374609</td>
<td>14629130343</td>
</tr>
<tr>
<td>lsu_i1</td>
<td>5917801019</td>
<td>511981149</td>
</tr>
<tr>
<td>lsu_o1</td>
<td>3101403179</td>
<td>357650216</td>
</tr>
<tr>
<td>lsu_i2</td>
<td>2837681904</td>
<td>160940558</td>
</tr>
<tr>
<td>bool_i1</td>
<td>727162811</td>
<td>727162711</td>
</tr>
<tr>
<td>gcu_i1</td>
<td>685182913</td>
<td>652135088</td>
</tr>
<tr>
<td>gcu_i2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>gcu_o1</td>
<td>33112562</td>
<td>33112562</td>
</tr>
<tr>
<td>ALU_i1</td>
<td>9283893812</td>
<td>4949540487</td>
</tr>
<tr>
<td>ALU_i2</td>
<td>9111462456</td>
<td>4777109131</td>
</tr>
<tr>
<td>ALU_o1</td>
<td>9356966593</td>
<td>5030863906</td>
</tr>
<tr>
<td>IO_i1</td>
<td>64844</td>
<td>64844</td>
</tr>
<tr>
<td>rtc_rtimer_i1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>rtc_rtimer_o1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>RF_1_o1</td>
<td>11146209723</td>
<td>2451084795</td>
</tr>
<tr>
<td>RF_1_i1</td>
<td>3627124847</td>
<td>1417893269</td>
</tr>
<tr>
<td>RF_1_1_o1</td>
<td>-</td>
<td>2589929301</td>
</tr>
<tr>
<td>RF_1_1_i1</td>
<td>-</td>
<td>1432303103</td>
</tr>
</tbody>
</table>

Table 5.6: Trigger counts at all the sockets of the initial processor configuration with and without additional RF unit

these approaches that have resulted in significant increase in the throughput.

5.3.1 Code analysis

The application was profiled with Proxim tool to obtain segments of the code that had peak execution. The check-node computation stage described by Eq. 3.20 and Eq. 3.24 for all decoding algorithms showed peak execution. To understand the execution profile better, an algorithmic representation of Eq. 3.24 is shown in listing 5.1.

Line 5 of the listing 5.1 showed peak execution as it involves computation of minimum and absolute of all neighbouring variable nodes connected to a given check node. A hardware unit computing minimum and absolute of two variables, referred here as minmod, was developed and interfaced with the initial processor configuration with multiplier and two additional register files. The block diagram of the custom unit is shown in Fig. 5.6, while the TTA processor consisting of this unit is shown in Fig. 5.7. The custom unit is tailored for RMSA algorithm given in Eq. 3.24 that takes $\Lambda_v, \Lambda_{i \rightarrow v}$.
loop across all checknodes
loop for each node degree of checknodes
loop for each node degree checknodes
if (node weights are not equal)
   \[ L_{\text{check2var}} = \min(\mod(L_{\text{check2var}}), \mod(L_{\text{var2check}})) \]
   \[ \text{sign} \Leftarrow \text{getsign}(L_{\text{var2check}}) \]

Listing 5.1: Check node computation

![Custom processor design with minmod unit](image)

Figure 5.6: Custom processor design with minmod unit

and other values of \( \Lambda_{c \rightarrow v} \) to find minimum. The same unit is used for M-kernel decoding algorithm as well. The hardware unit can be reused for MSA algorithm by providing them with inputs \( \Lambda_{v \rightarrow c} \), zero and other values of \( \Lambda_{c \rightarrow v} \).

The design was extended by adding additional minmod units and transport buses. The comparison of throughput for various processor configuration consisting of minmod units are shown in Fig. 5.9 for DVB-T2 short matrix and in Fig. 5.8 for DVB-T2 long matrix. The minmod unit does not parallelise because of two reasons. The computation of sign, shown in line 6 of the listing 5.1 is not a part of the hardware unit. The second reason being that the computation takes place for all the neighbouring variable nodes only. This introduces a condition in the application where it has to check if a given node is a neighbour or not which is dynamic in nature not constant during compile time.

To exploit parallelism of the for loop of the application, line 4, 5 and 6 were converted into a custom unit. The block diagram of scalable hardware unit that computes minimum, absolute and sign of the messages is shown in Fig. 5.11 and is referred as sclmmmod here. In addition to \( \Lambda_{v}, \Lambda'_{c \rightarrow v} \) and other values of \( \Lambda_{c \rightarrow v} \), the unit also takes in the loop variables \( j \) and \( k \) to perform comparison inside the hardware unit and outputs the minimum value and sign of the \( (\Lambda_{v} - \Lambda'_{c \rightarrow v}) \). The processor with sclmmmod unit is shown in Fig. 5.10. The comparison of throughput for various processor configuration
Figure 5.7: Block diagram of \textit{minmod} unit

Figure 5.8: Comparison of throughput rates for all the three algorithms on various processor configuration using \textit{minmod} unit for DVB-T2 long matrix

consisting of \textit{minmod} units are shown in Fig. 5.13 for DVB-T2 short matrix and in Fig. 5.12 for DVB-T2 long matrix.

We can see that introduction of additional \textit{sclmm} unit does not increase the throughput suggesting that the second unit was not utilised during computation. In addition, the throughput decreases compared with previous \textit{minmod} custom unit. This is because the device needs several buses to feed in the input and when buses are not available, it introduces wait states decreasing the overall throughput of the processor. To mitigate this, additional buses were introduced which increased the throughput, but additional \textit{sclmm} unit remained unused. The node degree of each checknode is not a constant during compile time and is determined during runtime. Hence, the custom
5.3.2 Parallelising check-node stage

The design of the custom processor was pushed further by incorporating entire check-node update stage as a Special Function Unit (SFU). The check-node stage described in Eqn. 3.24 of the RMSA decoding algorithm is implemented in HDL language and is integrated with rest of the processor. The SFU is a state machine with a FSM based control path. The data path is shown in Fig. 5.17. The design of the SFU exploits the fact that the rearranged matrix shown in Fig. 3.10 has a fixed row weight $m = 7$, with
an exception of the first row where \( m = 6 \). Hence the SFU has 14 inputs corresponding to seven \( \Lambda_v^{(j-1)} \) and seven \( \Lambda_c^{(j-1)} \) represented in the figure as Lcv' and Lv respectively. The trigger is applied to Lcv'[7].

The design of the data and control path is inspired from the listing 5.2. The first stage corresponds to the first loop, where it finds the minimum value (minValfOut), its corresponding node (node) and next minimum value (nMinValfOut). The second stage corresponds to the second loop of the listing 5.2 where it performs node comparison and substitutes either minValfOut or nMinValfOut. It also performs syndrome check on each row of the parity matrix that is later combined in the outer loop. The SFU outputs eight values: seven corresponding to \( \Lambda_j^{c \rightarrow v'} \), shown as Lcv in the figure and one sbit used for syndrome checking. The SFU has a total block latency of 20 cycles.

The implementation of the FU was added to the HDB of the TCE toolchain and was tested using Hardware Database Tester for functional correctness with the simulation model. Additional units were added based on the FU utilisation data given by the toolset. Optimisations were carried out on processor using Simple IC Optimiser and MinimiseADF explore plugins of the toolset. The final processor consists of basic units such as arithmetic and logic unit, load store unit, input and output unit, jump and branch unit along with additional units and four register files of depth 20 and 32 bits wide. The additional units added to the basic unit is shown in Fig. 5.14 and consists of a timer, multiplier, two adders, shift right and left units, and cnus unit.
Processor configuration

Figure 5.12: Comparison of throughput rates for all the three algorithms on various processor configuration using \textit{sclmmod} unit for DVB-T2 long matrix

The TTA processor with custom unit for computing check node stage gives a peak throughput of 85315.5 bps for RMSA decoding algorithm. The same unit was also used for M-kernel algorithm. The non-zero row and column indices were extracted from $\mathcal{S}$ matrix in software. The extracted indices were then used to obtain check-node and variable node messages. The throughput rate for M-kernel algorithm was 10 times slower with 8694.7 bps. As noted in section 3.4.2, due to non-sequential memory access, the variable node update stage for RMSA and M-kernel algorithm cannot be parallelised and hence implementation of variable node update stage was carried out in software. Fig. 5.16 gives the throughput rate of RMSA and M-kernel algorithm for various processor configuration. It can be seen from Fig.5.16 that addition of $cnus$ unit does not alter the throughput indicating that additional $cnus$ units are not used.

The lack of parallelism can be explained with the help of dependency graphs of the instructions obtained prior to register allocation phase by the compiler. This intermediary stage is produced by the LLVM compiler and gives an overview of how instructions depend on each other. From the Fig. 5.15, it can be seen that every input and output of the $CNUS$ instruction is a consequence of arithmetic operation followed by memory operation or vice versa. This is due to the fact that in its simplest form,
every message originating from a variable node is indexed by another array consisting of its address location. These values are dynamic in nature and cannot be resolved during compile time making loop-unrolling and thereby parallelism not possible.

In addition to the problem of resolving a double pointer, the MSA algorithm needs additional seven inputs and outputs to compute syndrome. This brings the total tally of inputs to 21 and outputs to 14. Large number of input and outputs makes parallelism difficult. As feeding data from the memory takes more clock cycles than actual computation, the implementation of check node unit for MA algorithm was not done.

The data dependency graph was further analysed to find recurring instruction pat-
minVal = infinity; nMinVal = infinity; sbit = 0

loop for each node_degree in checknodes
    temp[node_degree] = Lc[node_degree] - Lcv[node_degree];
    if(mod(temp[node_degree]) < mod(minValfOut))
        nMinValfOut = mod(minValfOut);
        minValfOut = mod(temp[node_degree]);
        node = node_degree;
    else if(mod(temp[node_degree]) < mod(nMinValfOut))
        nMinValfOut = mod(temp[node_degree]);

loop for each node_degree in checknodes
    if (node != node degree of minimum value)
        output[node] = minValfOut * getsign(temp[node degree])
    else
        output[node] = nMinValfOut * getsign(temp[node degree])
    if (Lc[node] > 0)
        sbit ^= 1

Listing 5.2: Alternate way to compute check nodes

terns that can benefit from a custom instruction. Four instruction patterns were found recurring throughout the program: multiply and accumulate for integers, addition followed by logical AND operation, addition followed by half word load operation and addition followed by half word store operation. However, these patterns could not be simulated or implemented due to an existing bug in the TCE toolchain that fails to convert a commutative operation such as ADD and AND into an appropriate LLVM instruction template. LLVM accepts an immediate operation only as a right operand, but the conversion of custom instruction involving commutative operations creates im-

Figure 5.15: Dependency graph of cnus instruction
mediate operation with a left operand. This is a known issue of TCE toolchain and is expected to work for more complex instruction patterns.

Figure 5.16: Comparison of throughput rates of RMSA and M-kernel algorithms on various processor configuration using \textit{cnus} unit for DVB-T2 long matrix
Figure 5.17: Data path design of the cnus unit
6 Conclusion

The design of the LDPC decoder using TTA architecture involved exploring algorithms for various throughput and memory requirements. Minimum Sum Algorithm (MSA), Reduced Minimum Sum Algorithm (RMSA) and M-kernel algorithm were chosen based on their ability to parallelise and memory footprint. The design exploration was carried out with an initial processor giving a throughput of 93.06 bps for MSA algorithm and was extended until a maximum throughput of 85315.5 bps for RMSA algorithm was obtained.

6.1 Summary of Results

The design exploration was carried out in three steps. In the first step, the initial processor configuration was extended by addition of functional units, register files and transport buses to see their impact on throughput rates. The intention here was to search any existing patterns and limitations. It was found out that addition of buses helped execution of instructions in parallel, but beyond a certain point, further addition reduced the throughput due to the increase in instruction size. Addition of RF units also brought about increase in throughput, but this should be balanced with memory and power consumption. A dedicated multiplier unit brought about significant increase in throughput, but the same could not be said with addition of other units. As there exists an exhaustive lists of functional units shipped with TCE, it would not be possible to check all the combinations.

The second step involved profiling the code to find the areas of the code that took large amount of cycles to execute. The profiled code resulted in the design of two custom units namely minmod and sclmmad units. The impact on throughput was studied in section 5.3.1 and it was found out that due to the limitations from the algorithm, these units could not be parallelised. The third step involved designing a custom unit for the computation of messages for an entire check node. The limitations from the
algorithm was avoided by using a rearranged matrix shown in Fig. 3.10, which had the property of same value for node degrees across all check nodes. The aim was to parallelise check node computation. The custom unit provided a significant speed up but failed to parallelise due to large number of IO and memory operations required by the custom operation. The summary of results obtained from all the three steps are summarised in Fig. 6.1 and in Table 6.1 for DVB-T2 long matrix.

<table>
<thead>
<tr>
<th>Processor Configuration</th>
<th>MSA</th>
<th>RMSA</th>
<th>M-kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial processor (IP)</td>
<td>93.06</td>
<td>76.86</td>
<td>86.40</td>
</tr>
<tr>
<td>IP + mul + 1 RF + 1 Bus</td>
<td>1221.87</td>
<td>1593.49</td>
<td>1406.19</td>
</tr>
<tr>
<td>IP + mul + 2 RF + 1 Bus + 1 minmod</td>
<td>3445.97</td>
<td>4011.14</td>
<td>3256.69</td>
</tr>
<tr>
<td>IP + mul + 2 RF + 1 Bus + 1 sclmm</td>
<td>3241.41</td>
<td>4138.4</td>
<td>3069.75</td>
</tr>
<tr>
<td>Final TTA</td>
<td>-</td>
<td>85315.5</td>
<td>8694.7</td>
</tr>
</tbody>
</table>

Table 6.1: Throughput rates in bits/second from various processor designs for decoding LDPC codes.
Table 6.2: Comparison of normalised throughputs among CPU, Exynos 4412 SoC and TTA implementations given in kpbs/MHz

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>Exynos SoC</th>
<th>TTA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RMSA</td>
<td>M-kernel</td>
</tr>
<tr>
<td>RMSA</td>
<td>3.9852</td>
<td>1.0980</td>
<td>1.70631</td>
</tr>
</tbody>
</table>

### 6.2 Conclusion

A clear understanding of the throughput rates and device consumption can be obtained by comparing it with existing implementations. Two implementations are chosen: first on an Intel i7-950 CPU with four physical cores clocked at 3.06 GHz and second on Samsung Exynos 4412 SoC hosting four ARM Cortex A9 cores clocked at 1.6 GHz [36] [37]. The throughput rate on the CPU with decoding algorithm running on a single core with hyper-threading turned off is 12.1950 Mbps, while on Exynos 4412 SoC for single ARM Cortex A9 core is 1.7569 Mbps. The throughput rates of all implementations are normalised to 1 MHz and is given in Table 6.2.

From Table 6.2, it can be seen that RMSA algorithm on TTA processor outperforms Exynos 4412 SoC implementation. However, M-kernel algorithm is slower by a factor of ten. The retrieval of the parity check-matrix from the compressed representation results in the reduction of the throughput. The CPU implementation is faster by a factor of two for RMSA algorithm and by a factor of 22 for M-kernel algorithm. It should be noted that TTA implementation is rather straightforward and simple compared to CPU. Also power consumption and area utilisation is considerably less for TTA implementation.

The TTA implementation enjoys several benefits over traditional hand-coded HDL designs of LDPC decoders for FPGAs and ASIC implementations. TTA architecture provides a middle ground between completely software and hardware approaches, delivering the benefits of developing an application in high level languages and flexibility to add custom hardware accelerators at various hierarchies. This allows embedding syndrome check operation into the hardware without complicated data path and control path design resulting in an better decoder implementation. TTA architectures also provide a ready processor template to integrate custom hardware. This alleviates the need to design the control and data paths for integrating with the processor and provides greater flexibility and reduced design time [38]. This property has been
extensively used in designing custom processors for computationally intensive operations such as 1024-point fast Fourier transform and in reconfigurable video coding applications [39] [40] [41] [42] [43] [44].

However TTA suffers from three major shortcomings. A data parallel application often requires dedicated memory architecture to fetch data in parallel and share memories. TTA ships with a default LSU unit which assumes an integer delay in memory fetch and store, but the user is given freedom to implement any kind of memory hierarchy. While this makes TTA implementation independent, the user has the responsibility to implement the entire memory architecture for a data intensive parallel application, increasing the design time. The work in [45] proposes one viable parallel memory architecture for TTA architecture.

TTA architectures help the user exploit parallelism at instruction level. By looking into instruction patterns prior register allocation, the user can develop custom instructions combining often repeated instruction patterns into a single custom unit and map the software and hardware without using macros. This ability gives the user quickly design and deploy custom accelerators. However, a known issue of the toolchain restricts users from implementing simple instruction patterns involving operations based on commutation. The last short coming of the TTA chain pertains to the lack of tools for design space explorations. The existing Explore plugins only support addition of simple units such as register files or removal of components based on initial seed design. With a large database of hardware units, manual design exploration becomes cumbersome.

To conclude, this work presents a viable solution to implement the decoding of long LDPC codes using TTA processors. The simulation and implementation results for MSA, RMSA and M-kernel algorithm are given. For a given LDPC code, the performance was maximal for RMSA algorithm, however for an implementation with constrained memory, M-kernel algorithm can be used. While TTA architectures perform well for computationally intensive operations, for a memory intensive application such decoding long LDPC codes, TTA architectures perform well as the throughput rates are comparable with CPU and SoC implementations taking size, power and area consumption into account. Easier IP reuse coupled with the possibilities to further increase throughput rate by exploiting hierarchical and modular nature of TTA makes it suitable for software defined radio applications.
6.3 Future Work

A number of improvements can be made on the existing work to improve throughput and decoder performance. The M-kernel performance could be boosted by including extraction of indices from the compressed $\Sigma$ matrix as a hardware unit. The decoder performance of all the decoding algorithms can be improved by using floating point representations of LLRs instead of 8-bit integer representation. The impact of using floating point operations on throughput and the tradeoff can also be studied.

The extent of parallelising check node and variable node stages of the decoder can be further probed by implementing a parallel memory controller as suggested in [45]. The check nodes and variable nodes can be grouped in 360 as in M-kernel algorithm and provide only the tuples $(C_{p,q}, D_{p,q})$ to generate rest of the indexes and memory locations. This coarse grain parallelism can decrease the data transfer load on transport buses by pushing the memory load store units into the dedicated check node or variable node stage processor. The check node or variable node stage processor can be either a custom hardware design or another TTA processor at a lower hierarchy.
BIBLIOGRAPHY


[22] T.J. Richardson, M.A. Shokrollahi, and R.L. Urbanke. Design of capacity-
approaching irregular low-density parity-check codes. *Information Theory, IEEE


certain coding problems (corresp.). *Information Theory, IEEE Transactions on*,


[26] A. Anastasopoulos. A comparison between the sum-product and the min-sum
iterative detection algorithms based on density evolution. In *Global Telecommunica-

[27] Xiaofei Huang, Suquan Ding, Zhixing Yang, and Youshou Wu. Fast min-sum
algorithms for decoding of ldpc over gf(q). In *Information Theory Workshop,

handbook.pdf".

[29] Altera. Logic array blocks and adaptive logic modules in stratix iii devices. "http:
//www.altera.com/literature/hb/stx3/stx3_siii51002.pdf".

altera.com/literature/hb/stx3/stx3_siii51004.pdf".

stx3_siii51007.pdf".

literature/hb/stx3/stx3_siii51017.pdf".


inference.phy.cam.ac.uk/mackay/codes/alist.html".

cs.utoronto.ca/~radford/ldpc.software.html".

71


APPENDIX

A.1 Alist Format

A alist format is a data structure stored in a file. The data structure representation in C is shown below.

```c
typedef struct {
    int N, M;    /* size of the matrix */
    int biggest_num_m;    /* actual biggest sizes */
    int biggest_num_n;
    int *num_mlist;    /* weight of each row, m */
    int *num_nlist;    /* weight of each column n */
    int **mlist;    /* list of integer coordinates in the m direction where the 
                     non-zero entries are */
    int **nlist;    /* list of integer coordinates in the n direction where the 
                     non-zero entries are */
} alist_matrix;
```