TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY
Julkaisu 821 • Publication 821

Perttu Salmela

Implementations of Baseband Functions for Digital Receivers

Tampere 2009

Tampereen teknillinen yliopisto. Julkaisu 821
Tampere University of Technology. Publication 821

Perttu Salmela

Implementations of Baseband Functions for Digital Receivers

Thesis for the degree of Doctor of Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB104, at Tampere University of Technology, on the 7th of August 2009, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2009

ISBN 978-952-15-2187-4 (printed)
ISBN 978-952-15-2210-9 (PDF)
ISSN 1459-2045

ABSTRACT

With ever higher data rates, the complexity of baseband processing increases basically for two reasons. Firstly, the required processing rate is proportional to the bit rate and, secondly, with higher data rates, more demanding and sophisticated algorithms must be applied. For example, new wireless telecommunications systems like 3G long term evolution (LTE) can have even a 100 Mbps data rate and multiple-input multiple-output (MIMO) transmission methods are applied. Thus, the problem domain of implementation of baseband functions includes both addressing the high computational complexity and describing the implementations in a flexible way so that even complex algorithms can be used without extensive efforts.

In this Thesis, implementations and implementation methods of baseband processing functions are proposed. Computational complexity and flexibility of implementation are approached with application-specific processors (ASP). The computing demands can be met with high parallelism when parallelization of the targeted algorithm is possible, and the software description of the computation possesses flexibility. Especially, the error correction decoding, matrix decomposition, and symbol detection tasks of the baseband processing chain are targeted in this Thesis. Both processor implementations and implementations of assisting hardware units are presented. With all the presented principles and implementations, programmable ASPs are targeted even though other platforms could also be used.

As a result, the essential computational challenges and the design space of wireless receivers is clarified. The work in this Thesis shows how the computation of the addressed baseband functions can be implemented efficiently, and the work shows how they can be implemented when a programmable platform is targeted. The results show that the benefits of the programmability do not sacrifice efficiency of the implementation.

PREFACE

The work presented in this Thesis has been carried out in the Department of Computer Systems at Tampere University of Technology during the years 2003–2009 and partially at University of Maryland, College Park in 2006.

I would like to express my gratitude to my supervisor Prof. *Jarmo Takala* for his support, motivation, and efforts to make this research work possible. I would like to thank my thesis Reviewers, Prof. *Holger Blume* and Prof. *Joseph R. Cavallaro*, who provided valuable and insightful comments. Thanks to their constructive comments, this Thesis has achieved its final form. Also, I would like to thank Prof. *Shuvra S. Bhattacharyya* for his efforts to make my exchange visit possible and for his help and guidance in the US. I would like to thank also Prof. *Olli Silvén* for his encouraging support.

My friends and colleagues deserve also special thanks. Especially, the closest colleagues during the research work of this Thesis, *Harri Sorokin*, M.Sc., *Lassi Nurmi*, M.Sc., *Juho Antikainen*, M.Sc., Dr.Tech. *Tuomas Järvinen*, Dr.Tech. *Jari Nikara*, and Dr.Tech. *Vesa Lahtinen* deserve many thanks for many fruitful discussions, observant comments, and motivations. A lot of thanks are given to my colleagues *Pekka Jääskeläinen*, M.Sc., *Teemu Pitkänen*, M.Sc., and *Riku Uusikartano*, M.Sc., for their helpful assistance, constructive comments, and influential enthusiasm. In addition, I wish to express special thanks to Dr.Tech. *Adrian Burian* for his support, ideas, and productive collaboration. The whole research group in University of Maryland, College Park and, especially, Ph.D. *Chung-Ching Shen* and *Ruirui Gu*, M.Sc., deserve many thanks for their assistance and creative atmosphere.

I also want to thank *Teemu Sipilä*, M.Sc., and *Aki Happonen*, M.Sc., for their comments and industrial point of view. Yet, there are still far too many people to name in the Department of Computer Systems, who deserve thanks and with whom this work has been carried out. All the mentioned and many unmentioned people have had a great influence on this Thesis. Without that influence this work would not have been possible.

This Thesis was financially supported by the Nokia Foundation, the Ulla Tuominen Foundation, the Foundation of Advancement of Technology (TES), and the National Technology Agency of Finland (TEKES). The financial support is appreciated and acknowledged gratefully.

Finally, I express the deepest gratitude to the *Salmela* family and *Hanna* for the support and motivation and to all the friends for their influence on this Thesis.

Tampere, June 11, 2009

Perttu Salmela

TABLE OF CONTENTS

LIST OF PUBLICATIONS

This Thesis is a monograph, which contains some unpublished material, but is mainly based on the following publications. In the text, these publications are referred to as [P1], [P2],. . . , and [P9].

[P1] P. Salmela, J. Antikainen, T. Pitkänen, O. Silvén, and J. Takala, "3G long term evolution baseband processing with application-specific processors," *International Journal of Digital Multimedia Broadcasting*, Vol 2009, Article ID 503130, 13 p., 2009.

[P2] P. Salmela, H. Sorokin, and J. Takala, "A programmable max-log-MAP turbo decoder implementation," *VLSI Design*, Vol. 2008, Article ID 319095, 17 p., 2008.

[P3] P. Salmela, H. Sorokin, and J. Takala, "Low-complexity polynomials modulo integer with linearly incremented variable," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, Washington D.C., USA, Oct. 8–10, 2008, pp. 251–256.

[P4] P. Salmela, A. Burian, H. Sorokin, and J. Takala, "Complex-valued QR decomposition implementation for MIMO receivers," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Las Vegas, NV, USA, Mar. 30–Apr. 4, 2008, pp. 1433–1346.

[P5] P. Salmela, J. Antikainen, O. Silvén, and J. Takala, "Memory-based list updating for list sphere decoders," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, Shanghai, China, Oct. 17–19, 2007, pp. 633–638.

[P6] P. Salmela, R. Gu, S. S. Bhattacharyya, and J. Takala, "Efficient parallel memory organization for turbo decoders," in *Proceedings of the 15th Euro-*

pean Signal Processing Conference, Poznan, Poland, Sept. 3–7, 2007, pp. 831–835.

[P7] P. Salmela, T. Järvinen, and J. Takala, “Simplified max-log-MAP decoder structure,” in *Proceedings of SympoTIC’06 the Joint IST Workshop on Sensor Network & Symposium on Trends in Communications*, Bratislava, Slovakia, June 24–27, 2006, pp. 10–13.

[P8] P. Salmela, T. Järvinen, T. Sipilä, and J. Takala, “256-state rate 1/2 Viterbi decoder on TTA processor,” in *Proceedings of IEEE 16th International Conference on Application-specific Systems, Architectures and Processors*, Samos, Greece, July 23–25, 2005, pp. 370–375.

[P9] P. Salmela, T. Järvinen, T. Sipilä, and J. Takala, “Parallel memory access in turbo decoders,” in *Proceedings of the IEEE Symposium on Personal, Indoor, and Mobile Radio Communications*, Beijing, China, Sept. 7–10, 2003, pp. 2157–2161.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACS	Add Compare Select
ACSU	Add Compare Select Unit
ALU	Arithmetic Logic Unit
ASIC	Application-Specific Integrated Circuit
ASP	Application-Specific Processor
BCJR	Bahl, Cocke, Jelinek, and Raviv
BR	Boolean Register
CMP	CoMPare
CORDIC	COordinate Rotation DIgital Computer
CSSU	Compare-Select-Store Unit
CU	Control Unit
DFT	Discrete Fourier Transform
DP	Dual Precision
DSP	Digital Signal Processor
FEC	Forward Error Correction
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field-Programmable Gate Array

FU	Function Unit
FWL	Fractional Word Length
GE	Gate Equivalent
GPRS	General Packet Radio Service
HSDPA	High-Speed Downlink Packet Access
IDFT	Inverse Discrete Fourier Transform
IFFT	Inverse FFT
INC	INCrement by one
IP	Intellectual Property
IPC	Inter-Processor Communication
IR	long Immediate Register
IWL	Integer Word Length
LD	LoaD
LDPC	Low-Density Parity-Check
LMMSE	Linear Minimum Mean Square Error
LSB	Least Significant Bit
LSD	List Sphere Decoder
LSU	Load/Store Unit
LTE	Long Term Evolution
LU	Logic Unit
LUT	Look-Up Table
MAP	Maximum *A Posteriori*
MIMO	Multiple-Input Multiple-Output

ML	Maximum Likelihood
MSB	Most Significant Bit
MUX	Multiplexer
OFDM	Orthogonal Frequency Division Multiplexing
OPC	OPeration Code
PC	Program Counter
PCCC	Parallel Concatenated Convolutional Code
PE	Processing Element
PED	Partial Euclidean Distance
QAM	Quadrature Amplitude Modulation
RF	Register File
RISC	Reduced Instruction Set Computer
SDR	Software Defined Radio
SFU	Special Function Unit
SHU	SHift Unit
SIMD	Single Instruction Multiple Data
SISO	Soft-Input Soft-Output
SoC	System-on-Chip
SP	Single Precision
ST	STore
STAM	Symmetric Table Addition Method
STBC	Space Time Block Code
SUB	SUBtract

TCE	TTA-based Codesign Environment
TTA	Transport Triggered Architecture
UMTS	Universal Mobile Telecommunication System
VCP	Viterbi decoder Co-Processor
VLIW	Very Long Instruction Word
WL	Word Length
XOR	eXclusive-OR

LIST OF SYMBOLS

MIMO–OFDM and Symbol Detection

c	speed of light
$f_{carrier}$	carrier frequency
$\boldsymbol{H}$	channel matrix
K	list length of K-best LSD
$\boldsymbol{Q}$	Q matrix of QR decomposition of $\boldsymbol{H}$
$\boldsymbol{R}$	R matrix of QR decomposition of $\boldsymbol{H}$
$\boldsymbol{s}$	transmitted symbol vector
$\boldsymbol{s}'$	estimate of the transmitted symbol vector
t_{coh}	coherence time
v_r	speed of receiver
X_{T}	time domain signal
X_{F}	frequency domain signal
$\boldsymbol{y}$	received symbol vector

Viterbi Decoding

$B_k(i)$	branch metric of a state transition from ith state at kth trellis stage
i	sate index

k	trellis stage
N_{states}	number of the states of the encoder
p	index of survivor path information word
$P_k(i)$	path metric of state i at kth trellis stage
S_k^p	pth survivor path information word at kth trellis stage

Turbo Decoding

$\alpha_k(u)$	forward path metric of state u at kth trellis stage
a_0, a_1	operands of modified radix-2 ACSU
$\beta_k(u)$	backward path metric of state u at kth trellis stage
b_0, b_1	operands of modified radix-2 ACSU
c_{rw}	implementation dependent delay between read and write operations
C_{stage}	clock cycles per trellis stage
$d_k(u'u)$	branch metric of state transition (u', u) at kth trellis stage
$E_{decoding}$	efficiency as theoretical number of clock cycles per required clock cycles
f_c	clock frequency
$\boldsymbol{f}_{SISO}(\cdot, \cdot, \cdot)$	SISO component decoder
$\gamma_k^{00\ldots11}$	branch metrics corresponding to concatenated systematic and parity bits, $00\ldots11$, at kth stage
k	trellis stage index
K	code block length
$\boldsymbol{\lambda}^{in}$	extrinsic information input vector

$\boldsymbol{\lambda}^{out}$	new extrinsic information output vector
λ_k^{in}	extrinsic information input at kth trellis stage
λ_k^{out}	new extrinsic information output at kth trellis stage
$\boldsymbol{L}$	soft-bit estimate vector
L_k^0	intermediate term of derivation of L_k
L_k^1	intermediate term of derivation of L_k
L_k	soft output
L_{win}	window length
m_0, m_1	results of modified radix-2 ACSU
R	throughput
$s_k^{0 \ldots 6}$	intermediate terms of derivation of L_k and λ_k^{out}
sgn$(\cdot)$	signum function
$t_k^{0 \ldots 6}$	intermediate terms of derivation of L_k and λ_k^{out}
u, u'	next state, previous state
$U_{pred}(u)$	set of predecessor states of u
U_{sp}	set of state transitions corresponding with the same systematic and parity bit combination
$U_{suc}(u')$	set of successor states of u'
v, v'	next state, previous state
w, w'	next state, previous state
x^p	transmitted parity bit
x^s	transmitted systematic bit
y^p	received parity bit

y^s	received systematic bit
$\boldsymbol{y}^p$	received parity bit vector
$\boldsymbol{y}^s$	received systematic bit vector

Path Metric Normalization

c	constant term
c_j	jth bit of c
Δ_i	variable term of ith path metric
i	state index of path metric
j	bit index
m_i	ith path metric
$m_{i,j}$	jth bit of ith path metric
W_Δ	word length of Δ

Parallel Memory Access in Turbo Decoding

$B(i)$	sawtooth access sequence at ith step
B_{addr}	address of the accessed bank
B_{sel}	selected bank
c_{rw}	implementation dependent delay between read and write operations
C	number of columns of interleaving matrix
$col(j, K)$	column of jth element as the interleaving matrix is filled
e_j	jth element
f_c	clock frequency

i	index of access sequence generation
j	generated access sequence index
L_b	buffer length
K	code block length
L_{win}	window length
$M_{cols}(K)$	columns of interleaving matrix for K length code block
$M_{rows}(K)$	rows of interleaving matrix for K length code block
N_{banks}	number of parallel memory banks
$P_{inter}(\cdot,\cdot)$	inter-row permutation
$P_{intra}(\cdot,\cdot,\cdot)$	intra-row permutation
$\pi(\cdot,\cdot)$	interleaving function
R	number of rows of interleaving matrix
$row(j,K)$	row of jth element as the interleaving matrix is filled
T	number of parallel access interfaces
x_j^s	jth systematic bit

3G LTE Interleaving Pattern Generation

deg	degree of polynomial
a	first degree coefficient of $E(\cdot)$
a_h	hth coefficient of $P(\cdot)$
$E(\cdot)$	first degree polynomial
j	linearly incremented index
K	modulus

mod^*	modulo in limited domain
n	degree of polynomial
N	length of generated sequence
$P(\cdot)$	polynomial
$\Pi(\cdot,\cdot)$	experimented 3G LTE interleaving function
w_h	hth coefficient of $W(\cdot)$
$W(\cdot)$	polynomial
$y(\cdot)$	targeted function

Inverse Square Root Approximation and QR Decomposition

α	number of leading zero bits of x
α_i	ith bit of α
a_t	constant term of tth linear approximation
b_t	first order coefficient of tth linear approximation
c	N-bit subword of x
c_i	ith bit of c
$d_{i,t}$	sign of ith term of a sum presentation of the first order term of the linear approximation
$e_{i,t}$	exponent of the ith denominator of an approximation of the sum presentation of the first order term of the linear approximation
k	number of bits shifted in the last scaling
M	word length of u
N	word length of c
T_{QR}	time taken by QR decomposition

u	last M bits of x
u_i	ith bit of u
x	argument of inverse square root function

List Processing for Symbol Detection

$\mathcal{A}$	all symbol candidates
a	address of a node
b	jump latency
C_{branch}	clock cycles taken by heap filling and insertions with two insertion routines
C_{fill}	clock cycles taken by heap filling and insertions with separate filling routine
$C_{non-pipelined}$	clock cycles taken by insertion routine without software pipelining
$C_{pipelined}$	clock cycles taken by software pipelined insertion routine
d	heap node
e	list element
$f(n)$	clock cycles taken by n inserts
g	list element
k	number of times n elements are inserted in filling routine comparison
$\mathcal{L}$	list of n symbol candidates
m	depth of the heap
n	list length

$parent(d)$	parent node of node d
$val(d)$	value of node d
x	candidate symbol

Complexity and Power Estimations

C_{FFT}	clock cycles taken by the FFT task
C_{QR}	maximum clock cycles taken by the QR decomposition task
$C_{\mathrm{QR\_avg}}$	average clock cycles taken by the QR decomposition task
C_{LSD}	clock cycles taken by the LSD task
C_{Turbo}	clock cycles taken by the turbo decoding task
D_{FFT}	delay of the FFT task
D_{LSD}	delay of the LSD task
D_{QR}	delay of the QR task
D_{Turbo}	delay of the turbo decoding task
f_i	clock frequency for task i
i	task index
n	list length of the K-best LSD
P_i	required number of processors for task i
R	code rate
S_{tasks}	set of tasks
T_{FFT}	OFDM symbol time
U	total utilization
U_i	utilization of the processors processing task i

1. INTRODUCTION

In the field of telecommunications, there is a constant drive for ever higher data rates. For example, mobile telecommunications systems have evolved from the first digital 2G systems to 3G systems and descendants of the 3G are under development. The transitions between telecommunications system generations have never been sharp but there have been enhancements like general packet radio service (GPRS) or high-speed downlink packet access (HSDPA) and the 3G long term evolution (LTE) is coming in the near future for a smooth migration to higher data rates. Updating standards and higher data rates affect computation requirements of baseband processing, i.e., processing which operates at the same rate as the data is received or transmitted. In addition to only scaling proportionally to the data rate, the demands of baseband processing are also affected by the required more sophisticated algorithms for achieving the high data rates. For example, multiple-input multiple-output (MIMO) and orthogonal frequency division multiplexing (OFDM) techniques will be used in 3G LTE [16] and to utilize the available gain in transmission capacity, computationally demanding symbol detection methods must be used. Another demanding task is error correction with turbo decoding [21] both due to the computational complexity and due to the complex control of parallel processes.

Typically, the initial goals of the upcoming data rates and other system performance metrics are set before the technology to implement them is mature. The first implementations are being developed even if the new standards were not yet finalized. This is natural, since it is advantageous for commercial vendors to provide the first products as soon as possible. For example, rapid development may provide opportunities to patent developed solutions. It is also advantageous if the migration to higher data rates proceeds as fast as possible, since consumers have been willing to adopt products whose features are based on ever higher data rates.

The aforementioned progression results in several challenges to the development and implementation of baseband functions. In addition to the plain need for implementations of baseband functions, the work of this Thesis can be motivated by the following four challenges. The first challenge is the high computational load which is caused by high data rate. In other words, higher data rates require more computations to be carried out in the receiver than lower data rates. In addition to the computational complexity, the design complexity is also increased, i.e., the high data rates require complex algorithms and designing efficient implementations of such algorithms requires extensive efforts. This is the second challenge. As a third one, the design time should be as short as possible for the aforementioned requirement of rapid time-to-market and for the rapid updates of telecommunications standards with new transmission techniques. The fourth challenge is the requirement of efficiency. If the final products are targeted to high volume consumer markets, wasteful simple solutions using more resources than it is required in reality, would be inefficient and result in high costs.

There exists some approaches how the aforementioned four challenges can be faced. The high computational load is typically met with an application-specific hardware implementation. Moreover, the computing capacity of the application-specific hardware implementation typically originates from a high parallelism. Thus, if the computations can be parallelized, the parallelization provides one solution to achieve high computational capacity. On the other hand, if parallelization is impossible, sequential operations can be accelerated by shortening the critical path of the computing hardware implementation.

Describing complex algorithms is more convenient with higher level languages than with accurate hardware descriptions. This is natural, as there are less details to be specified in higher level presentation than in a lower level presentation. Higher level descriptions or their compiled descriptions can be, typically, simulated faster than hardware descriptions. This shortens the delay of a design cycle, which begins at a change in the desired functionality of the system and ends in the verification of the results. If the applied description language is a programming language and a programmable implementations are targeted, the functionality of the system can be changed afterward. For example, software defined radios (SDR) [75, 76] require programmable implementations. The idea of the SDR is that it can be adapted to several transmission techniques by re-programming. The whole baseband processing architecture would be re-programmable in an ideal SDR. However, area-efficient or power-efficient implementations require often detailed, highly optimized hardware descriptions as the design tools cannot always optimize high-level descriptions sufficiently. Therefore, it is advantageous to describe some parts of the targeted functions on low level as pure hardware.

In this Thesis, the aforementioned methods to meet the design challenges are put into practice by applying application-specific processors (ASP). In general, the ASP lies in the range between digital signal processors (DSP) and pure hardware implementations. The benefits of DSPs are their programmability and, therefore, flexibility and rapid design time. On the other hand, pure hardware implementations tend to achieve highest efficiency and computing capacity. Naturally, with ASPs, it is targeted to obtain the benefits of both DSPs and pure hardware implementations. As the applied ASP template is customizable, the level of parallelism of the processor can be adjusted. Thus, the first challenge of high computing capacity can be met with parallel computing resources assuming that the computations of the algorithm can be parallelized. Naturally, the processor is programmable, so the behavior can be described at higher level than with pure hardware implementation. Since the ASP is customizable, also low-level detailed hardware descriptions of kernel computations can be utilized as dedicated hardware units can be included in the data path of the processor.

1.1 Scope and Objective of Research

In this Thesis, implementations of certain baseband functions of the receiver are considered. The scope is exemplified with a simplified transmission system consisting of a transmitter, channel, and receiver shown in Fig. 1. There exist several important baseband functions. Some of them are present only when certain transmission techniques are applied. Different standards applying the same transmission techniques may still require different parameters of the respective baseband functions. In this Thesis, the error correction decoders, list processing and matrix processing required by MIMO symbol detection, and inverse square root function approximation required by matrix processing are considered in detail. As the implementations of the addressed baseband functions are developed, they are implemented with customizable ASPs or as hardware blocks, which can be included in the datapath of such ASPs.

The objective of this Thesis is to show how such baseband functions can be imple-

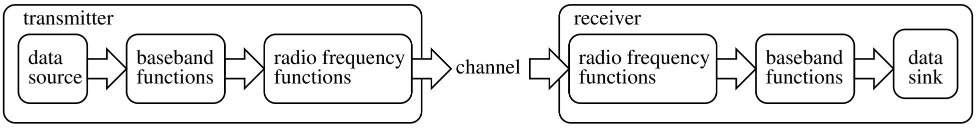


***Fig. 1.** A high-level block diagram showing the position of baseband functions in wireless digital transmission system.*

mented with the applied ASP template and to show that the price of programmability and flexibility does not exceed their benefits. In other words, a reasonable efficiency can be obtained even if the underlying architecture is programmable and its behavior is described at higher level than pure hardware implementations. When plain hardware blocks, which can be used in the data path of such ASPs, are presented, the objective is to show how the targeted function can be accelerated conveniently with such hardware blocks. In other words, the targeted function possesses a division into high-level control and low-level computation kernels and such division lends itself to the ASP implementations.

1.2 Main Contributions

The contributions in this Thesis are ASP or hardware implementations of particular functions, which are required in the baseband processing of the receiver. As the implementations are considered relatively deeply, there are implementations of assisting functions like parallel memory access methods even if its real target function is turbo decoding. To summarize the implementations, also a part of the whole baseband processing chain applying the ASP implementations is analyzed.

The main contributions of this Thesis can be stated as follows:

- Function units for Viterbi decoding. A set of four units are proposed per software pipelined radix-2 add compare select (ACS) task.
- A method for time-multiplexing modified ACS units (ACSU) for all the required computations of max-log-MAP algorithm.
- Function units for stage-parallel turbo decoding.
- Two parallel memory access methods for stage-parallel turbo decoding.

- Function unit for generating the interleaving pattern of 3G LTE turbo codes.
- A scalable low-complexity function unit for coarse approximation of inverse square root function.
- An application-specific processor for complex-valued QR decomposition targeted to MIMO systems.
- List processing function units for list sphere detection.
- Complexity and power estimations of a baseband processing chain consisting of fast Fourier transform (FFT), turbo decoder, QR decomposition, and K-best list sphere decoder (LSD).

Even if the number of presented baseband functions is limited, their implementations present a variety of methods and principles which could be applicable also with other functions or in other fields.

1.3 Author's Contribution

The author has acted as the first author of all the publications [P1]–[P9], on which this Thesis is based. For turbo decoding, the author has been responsible for finding the ways how parallel memory accesses can be established [P6, P9] and the author has carried out the required simulations and implemented the hardware units. The author has designed the turbo and Viterbi decoder processors and the accompanied accelerating units [P2, P8], programmed the parallel assembly codes, and implemented the processors. The applied time-multiplexing of computing resources [P7], normalization method, and the used partitioning of tasks to computing units were ideas of the author. The author was also responsible for the comparisons of turbo decoders in [P2].

Computation of polynomials modulo integer for interleaving pattern generation [P3] was derived by the author. Also the simulation model and the hardware unit were developed by the author. The memory and register based list processing units and the processor using heap for list processing [P5] were developed by the author.

The key idea of inverse square root approximation was invented by Dr. Tech. Adrian Burian, and the author extended the idea to a scalable form. The author also wrote the derivations, implemented the computation units, and prepared the comparisons. The inverse square root unit was used in QR decomposition processor [P4], which was designed and programmed by the author.

The analysis of the baseband processing chain and the processing requirements for the targeted data rate in [P1] are derived by the author. In [P1], processors developed by Juho Antikainen, M.Sc. and Teemu Pitkänen, M.Sc. are used as a part of the analyzed baseband processing chain.

The work reported in this Thesis has been reported earlier in [P1]–[P9]. For this reasons, many Chapters contain verbatim extracts of [P1]–[P9]. The extracts are under copyright of respective copyright holders. None of the publications [P1]–[P9] has been used in another academic Thesis.

1.4 Thesis Outline

The Chapter 2 provides the necessary background information and serves as a map of the addressed baseband functions. The Chapter shows where the functions reside in the receiver and explains their operation on a high level. In addition, the applied ASP template is introduced in Chapter 2.

The error correction decoders are considered in Chapters 3–6. First, a Viterbi decoder is presented in Chapter 3. In the succeeding chapters, a partial-stage turbo decoder, stage-parallel turbo decoder, and parallel memory accesses of turbo decoding are presented.

The Chapter 7 presents the inverse square root approximation method which is put into practice with the QR decomposition in the same Chapter. The Chapter 8 addresses list processing which is required by symbol detection. Before conclusions, the Chapter 9 summarizes the presented work by analyzing the complexity and power consumption of the processing chain.

2. RECEIVER MODEL AND PROCESSOR TEMPLATE

In this Thesis, several implementations are proposed for a limited set of baseband functions of digital receivers. With baseband it is meant that the processed signal has been demodulated and, therefore, the processing time requirements are determined by the data rate but not by the carrier frequency. Naturally, as there exists several inevitable baseband functions, only a couple of them are considered in detail in this Thesis. In this Chapter, a high-level MIMO–OFDM receiver model is used as an example to clarify where the targeted functions reside in the baseband processing chain. Naturally, the proposed implementations can be applied also with other transmission techniques when applicable. Most the addressed implementations are based on a specific processor template, namely transport triggered architecture (TTA) [32] processors in this Thesis. The principal structure of TTA processors and principles of programming them are presented in this Chapter. However, the methods and techniques presented in this Thesis could be applied also with other flexible ASP templates or with pure hardware implementations.

2.1 System Model

A high-level description of an example MIMO–OFDM receiver is presented in Fig. 2. The input ports of radio frequency functions are connected to the antennas of the receiver. The functional block diagram is only a high-level model as it does not suggest how the functions should be mapped to the processors nor it does not suggest how data is passed between the functions and whether the data vectors have serial or parallel presentations. In the following, the applied transmission techniques are presented briefly.

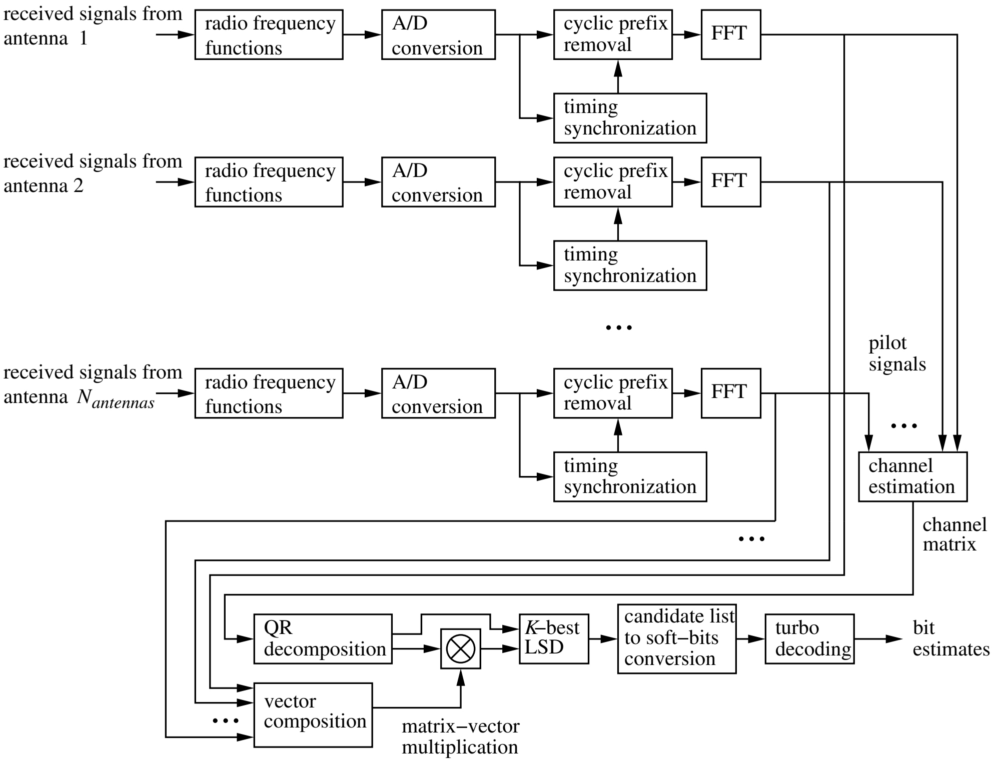


Fig. 2. *A simplified block diagram of baseband processing of a MIMO-OFDM receiver using K-best LSD for symbol detection.*

2.1.1 Orthogonal Frequency Division Multiplexing

OFDM uses the frequency spectrum efficiently as the used frequency band is divided into several orthogonal subcarriers [27]. The OFDM uses the discrete Fourier transform (DFT) and inverse DFT (IDFT) for conversions between the time and frequency domains. Typically, the transforms are computed with FFT and inverse FFT (IFFT) [36] in practice. The time domain signal is generated in the transmitter side with inverse transform,

$$X_{\mathrm{T}} = IDFT(X_{\mathrm{F}}) \,, \qquad (1)$$

i.e., data belonging to several parallel subcarriers is fed to the IDFT. In the receiver side, parallel subcarriers X_{F} are extracted from the time domain signal X_{T} with

$$X_{\mathrm{F}} = DFT(X_{\mathrm{T}}) \,. \qquad (2)$$

To alleviate timing synchronization, additional cyclic prefix is inserted to the signal. The channel estimation can be alleviated with pilot symbols. In the receiver side, distortion of the channel can be equalized conveniently in frequency domain. The equalization takes place by multiplication with equalizing factors which are determined with the aid of pilot symbols. Another advantage of the OFDM transmission is robustness against intersymbol interference with guard interval and low symbol rate. Before the DFT, the cyclic prefix must be removed from the signal, and timing synchronization is responsible for feeding the time domain signal, whose length equals the DFT length, with correct timing offset to the DFT block.

2.1.2 Symbol Detection

In a spatial multiplexing MIMO system, multiple antennas are used to transmit independent data streams. Spatial multiplexing gain, i.e., increase in capacity, is proportional to the number of antennas and it does not require extra power nor bandwidth [83]. Two transmit and receive antennas is a highly probable configuration for the first 3G LTE systems, since a higher number of antennas increases the computational requirements of symbol detection significantly. Therefore, a 2×2 MIMO configuration is assumed in this Thesis.

The computational complexity of maximum likelihood (ML) detection of transmitted symbols depends exponentially on the number of spatial channels. Therefore, even with a modest number of antennas, simpler approximative methods must be used. The usage of list sphere decoding algorithms is tempting as they can achieve higher performance than linear minimum mean square error (LMMSE) algorithm [78], even though they are computationally demanding. The sphere detector restricts the search space by evaluating only the symbols inside the sphere centered in the received symbol [50]. In the example system model in Fig. 2, K-best LSD is assumed. The K-best LSD operates by gradually increasing the dimension of the symbol vector [135]. At each level, a list of the K best partial solutions is selected for continued processing. The list processing of the LSD is a likely bottleneck of the list sphere decoding and, therefore, there is a strong demand for efficient implementations.

The QR decomposition is required by the K-best LSD as is shown in Fig. 2. The decomposition transforms a channel matrix $\boldsymbol{H}$ to a decomposition of an orthogonal $\boldsymbol{Q}$ and an upper triangular $\boldsymbol{R}$ matrices. The matrices are required by the LSD algorithm, which detects the received symbols. The LSD is used to estimate the transmitted symbol vector, $\boldsymbol{s}$, by approximating ML detection

$$\boldsymbol{s}' = \arg\min_{\boldsymbol{s}} \|\boldsymbol{y} - \boldsymbol{H}\boldsymbol{s}\|^2 \tag{3}$$

where the $\boldsymbol{y}$ is the received symbol vector and $\boldsymbol{H}$ is the channel matrix whose dimensions equal to the number of transmit and receive antennas of the MIMO system. The approximation is based on substitution with QR decomposition $\boldsymbol{QR}$=$\boldsymbol{H}$, i.e.,

$$\boldsymbol{s}' = \arg\min_{\boldsymbol{s}} \|\boldsymbol{y}' - \boldsymbol{R}\boldsymbol{s}\|^2 \quad \text{where } \boldsymbol{y}' = \boldsymbol{Q}^H\boldsymbol{y} \ . \tag{4}$$

The LSD approximates (4) by gradually increasing the dimensions of symbol vector and computing partial euclidean distances (PED). With this practice, the search space can be limited efficiently.

The coherence time, t_{coh}, indicates how long the channel impulse response is essentially invariant. Thus, the minimum update rate of $\boldsymbol{Q}$ and $\boldsymbol{R}$ is inversely proportional to the t_{coh}. The coherence time can be expressed as

$$t_{coh} = c/(v_r f_{carrier}) \tag{5}$$

where c is the speed of light, v_r is the speed of the receiver, and $f_{carrier}$ is the carrier frequency.

2.1.3 Forward Error Correction

The forward error correction (FEC) [12] operates by adding redundant data to the transmitted signal. With the aid of redundancy, errors caused by interfered channel can be corrected up to a certain limit. Naturally, with severely interfered channel all the errors cannot be corrected. The degree of added redundancy is defined by the code rate of the error correction code. Thus, the FEC is a vital part of modern telecommunications systems. In 3G and 3G LTE, both the convolution code and turbo code are used [2, 3] and Viterbi and turbo decoding are efficient algorithms for decoding such codes, respectively.

Principles of Viterbi Decoding

In principle, the Viterbi algorithm [121] traverses through a trellis consisting of alternative paths via states of the convolutional encoder. The target of the algorithm

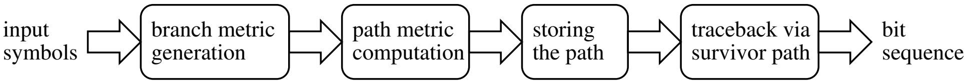


Fig. 3. *The Viterbi decoding consists of four loosely distinct computation phases.*

is to find the most likely sequence of the states. Basically, the Viterbi decoding can be divided into four tasks. The first task includes generating branch metrics of the received bits. The branch metrics present the distances between received symbols and the symbol alphabet. The second and the most computing intensive task is the computation of path metrics. In practice, the computation takes place by repetitions of ACS operations. As a third task, the path selection information of the previous phase must be saved. In the last phase, the trellis is traversed backward via survivor path which describes the estimated bit sequence. These four phases are illustrated on high level in Fig. 3.

Principles of Turbo Decoding

In principle, the turbo decoder decodes parallel concatenated convolutional codes (PCCC) in an iterative manner [21]. In addition to the variations in the actual decoding algorithm, the implementations can be characterized also with the level of parallelism, scheduling, or the required memory throughput.

The functional description of the PCCC encoding and turbo decoding is shown in Fig. 4. The encoding process in Fig. 4 passes the original information bit, i.e., systematic bit, unchanged. Two parity bits are created by two component encoders. One of the component encoders takes systematic bits in sequential order but the input sequence of the second component encoder is interleaved. The interleaving is denoted by π in Fig. 4.

The turbo decoding is described with the aid of soft-input soft-output (SISO) component decoders [24]. The soft information is presented as logarithm of likelihood ratios. The component decoder processes systematic bit vector, $\boldsymbol{y}^s$, parity bit vector, $\boldsymbol{y}^p$, and a vector of extrinsic information $\boldsymbol{\lambda}^{in}$. As a result new extrinsic information, $\boldsymbol{\lambda}^{out}$, and soft-bit estimates, $\boldsymbol{L}$, of the transmitted systematic bits are generated, i.e.,

$$(\boldsymbol{\lambda}^{out}, \boldsymbol{L}) = \boldsymbol{f}_{SISO}(\boldsymbol{\lambda}^{in}, \boldsymbol{y}^s, \boldsymbol{y}^p). \tag{6}$$

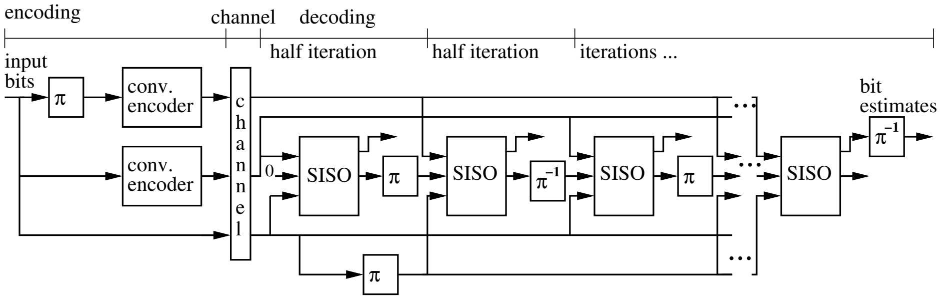


Fig. 4. *Turbo encoding and decoding. The decoding is an iterative process, which runs SISO component decoder several times. Interleaving and de-interleaving are denoted with π and π^{-1}, respectively.*

Passing the extrinsic information between the component decoders describes how *a priori* information of the bit vector estimates is used to generate new *a posteriori* information. The turbo decoding is an iterative process where generated soft information is passed to the next iteration. Every second half iteration corresponds with the interleaved systematic bits. Since the interleaving changes the order of the bits, the next component decoding cannot be started before the previous is finished. Therefore, the signals passed between the SISO component decoders in Fig. 4 are, in fact, vectors whose length is determined by the code block length.

Due to the long code block lengths a practical decoder implementation in Fig. 5 consists of the actual SISO decoder and memories. Since only one component decoding phase, i.e., half iteration, can be run at a time, it is economical to have only one SISO whose role is interchanged on every half iteration [123]. Although, if decoding is block-wise pipelined, then several component decoders can be used [24]. The extrinsic information is passed via a dedicated memory between the half iterations. If the component decoder is capable of processing one trellis stage on every clock cycle, dual access to the extrinsic information memory is required. The interleaving takes place by accessing the memory with interleaved addresses as shown in Fig. 5. In practice, the extrinsic information can reside in the memory in sequential order and no explicit de-interleaving is needed [123]. When the interleaving is required the extrinsic information is read from and written to according to interleaved addresses. Thus, the order remains unchanged and no explicit de-interleaving is required before accessing the memory in sequential order. In the end of decoding, the soft-bit estimates can overwrite the extrinsic information memory in Fig. 5.

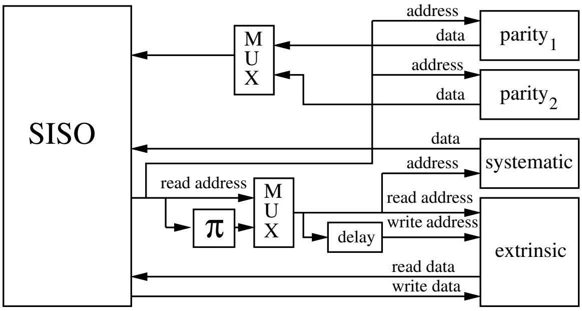


Fig. 5. Practical decoder requires SISO component decoder, interleaved address generation, and memories. Extrinsic information memory is both read and written.

The SISO component decoder can be implemented with, e.g., soft-output Viterbi algorithm or some variation of a maximum *a posteriori* (MAP) algorithm. The MAP algorithm is also referred as the Bahl, Cocke, Jelinek, and Raviv (BCJR) algorithm according to its inventors [17]. In this Thesis a max-log-MAP algorithm [41,60,88] is assumed. The basic MAP algorithm consists of forward and backward processes and both forward and backward path metrics are required to compute the final outcome. Due to the long block lengths, some type of sliding window algorithm, like the one presented in [120], is usually applied to reduce the memory requirements. In the sliding window algorithm, the backward computation is not started in the end of code block but two window length blocks before the beginning of current forward process. The backward path metrics are initialized with an acquisition process to appropriate values during the first window length trellis stages. After the acquisition, the backward process generates valid path metrics for the next window length stages.

Two alternative schedules for forward and backward computations are shown in Figs. 6(a) and (b). The stage-parallel schedule in Fig. 6(a) processes one trellis stage with three parallel processes. The partial-stage schedule in Fig. 6(b) processes one trellis stage sequentially, i.e., only one process running at a time.

2.2 Transport Triggered Architecture Processor Implementations

In this Thesis, the TTA has been used as the architecture template for ASPs. The TTA processors can be created and programmed with up-to-date TTA-based Codesign Environment (TCE) tools [54] or with the original MOVE toolset [33]. Processors with

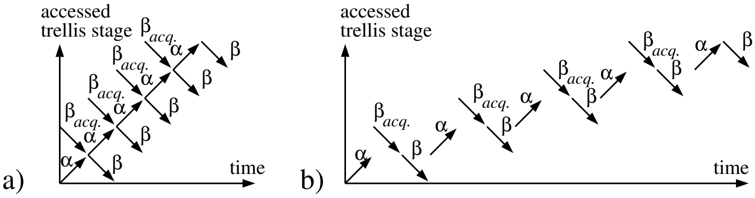


Fig. 6. *Schedules for sliding window algorithm: a) stage-parallel schedule applies three parallel processes, b) one process runs at a time in partial-stage schedule. Initialization of backward metrics with acquisition process is denoted with $\beta_{acq.}$, backward computation with β, forward computation with α.*

similar efficiency and performance could be implemented also with some other ASP templates if sufficient parallelism and customizability were supported. In TTA, the computations are triggered by data transported to the computing unit, which is contrary behavior to conventional operation triggered architectures. The processor is programmed with data transports, which reflects the architecture to the programmer. The maximum number of parallel data transports is determined by the number of buses of the interconnection network. As the interconnection network connecting the computing resources is visible to the programmer, there is accurate control of all the operations.

The modularity of TTA processors allows to tailor them by including only the necessary function units (FU). Application-specific functions are implemented as user defined special FUs (SFU) which are utilized in a similar way as conventional FUs, i.e., by transporting data on assembly level or by using function-like macros in C language. Due to frequent direct data transports between the FUs or SFUs, the register pressure is very low. However, the modularity of the processor allows a variable number of register files (RF) with variable numbers of input and output ports. In Fig. 7, a high-level example of a TTA processor is given. The figure highlights the modular and customizable structure of the processor by denoting the variable numbers of the respective resources. The control unit (CU) in Fig. 7 allows data transports to access the program counter and the return address register, which is required for jump or call operations. Basically, the data transports transport data via buses and the computing resources are connected to the buses with sockets. The instruction word controls the sockets so that the data is passed to the correct bus and the operands are read from the correct bus.

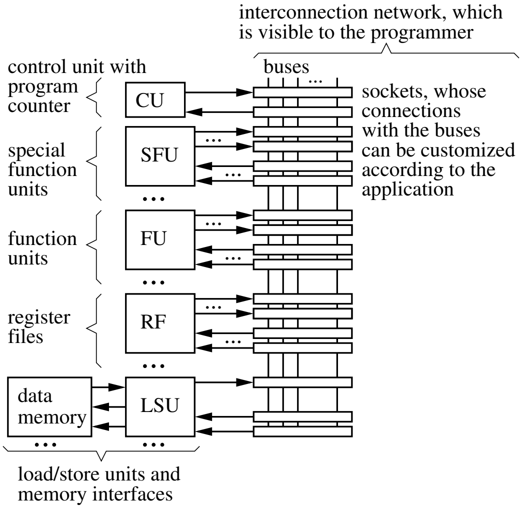

interconnection network, which is visible to the programmer
buses
control unit with program counter
CU
sockets, whose connections with the buses can be customized according to the application
special function units
SFU
function units
FU
register files
RF
data memory
LSU
load/store units and memory interfaces

Fig. 7. TTA processors consist of control unit (CU), function units (FU), special FUs (SFU), load/store units (LSU), register files (RF), and an interconnection network between the resources.

The load on the buses of the interconnection network can be lowered by excluding the unnecessary connections if the work load of the processor is known beforehand. In this case, the targeted application program determines which connections are used. Typically, one application requires only a fraction of all the possible connections between the computing resources. If any other application is run on the same processor, it must be able to use the same connections. As a consequence of the limited connectivity and lowered load on the buses, the maximum clock frequency of the interconnection network is raised.

In addition to customization by computing resources and their connections, also the word length of each bus, FU, SFU, and RF can be set according to the requirements of the application. Furthermore, even if the interconnection network had longer word length, the SFUs can use shorter words internally when appropriate. The possibility to vary word length is beneficial since the minimum word length requirements are known if the application is transformed from floating-point version to a fixed-point version and the word length requirements are analyzed during the transformation.

Since the TTA processors are fully generic, there does not exist any particular conventional or regular TTA processor as a counterpart to the more optimized and customized TTA processors. However, some classifications can be based on the applied toolset. For example, if the processor development tools have a full support for at maximum two load/store units (LSU) but using wider memory bandwidth requires more user interaction, the processors could be classified according to their memory interfaces. In a similar way, they could be classified to the processors using only the FUs provided by the toolset and to the processors accompanied with user defined SFUs. In practice, such a classification would be quite arbitrary as it would depend more on the features of the design tools than on the fundamental structure of the processor. Furthermore, TTA processors are typically used with applications which lend themselves to acceleration with SFUs. Thus, it could be concluded that conventional TTA processors include SFUs but less conventional TTA processor exclude SFUs.

2.2.1 Programming TTA Processors

The computation kernels of all the proposed TTA processor implementations are programmed in parallel assembly. The syntax of the TTA processor assembly language consists of only one operation. The move operation, $src \rightarrow dst$, moves data from left-hand side source, src, to the right-hand side destination, dst. The maximum number of moves in the instruction is determined by the number of buses in the interconnection network. In practice, the assembly instruction word consists of a series of move operations which are mapped to the buses in the same order as they appear in the word. Conditional execution takes place by guarding the move operations. The value of the guard can be read from a RF or from a FU in some cases. Guarded move operations with the program counter as their destination are used for conditional branching.

To achieve high efficiency, the principles of software pipelining have been applied in the computation kernels of the proposed TTA processor implementations. The software pipelining is a technique which allows executing loops in such a way that several loop iterations are issued in parallel. An example case is presented in Figs. 8(a) and (b). It shows how a total of four instances of loop iterations are run in parallel. Depending on the total number of required iterations, four parallel instances are iterated by conditionally branching from instructions at 07 back to instructions at 04. The example in Fig. 8(b) shows that during any of the instructions 03. . . 07 all the four operations A, B, C, and D are executed in parallel and, therefore, they must be mapped to four separate FUs.

```
for i:=1:N step 1 do
00:  A
01:  B
02:  C
03:  D
end
```

a)

	FU$_0$	FU$_1$	FU$_2$	FU$_3$
00 :	A			
01 :	B	A		
02 :	C	B	A	
03 :	D	C	B	A
04 :	A	D	C	B
05 :	B	A	D	C
06 :	C	B	A	D
07 :	D	C	B	A
08 :		D	C	B
09 :			D	C
10 :				D

b)

Fig. 8. *Principles of software pipelining: a) an example loop consisting of operations A, B, C, and D, b) pipelined iterations are mapped to function units* $FU_{0...3}$.

2.2.2 Developing TTA Processor Applications

Typically, developing a TTA processor application begins with a target algorithm given in fully sequential form, e.g., as a C program or a pseudo code which can be transformed to a C program. If fixed-point implementation is targeted, similar transformations from floating-point to fixed-point presentations as with any fixed-point processor application must be carried out. Since the compilation and scheduling of move operations map execution to a particular processor architecture, the initial architecture must be created before compilation. Typically, a minimum architecture capable of executing any programs generated by the compiler can be used as an initial TTA processor architecture. Naturally, instead of C compilation, the algorithm can be written also in parallel assembly.

Next, the processor design is iterated by including accelerating units or resources and excluding resources which have minor effect on the performance. The bottleneck operations, i.e., candidates for acceleration with SFUs, can be identified with the aid of profiling a compiled program or if the designer is familiar with the application, the likely bottlenecks are known beforehand. A candidate operation for a SFU can be determined, e.g., by chaining consecutive operations. Naturally, operation chaining leads to trade-off between cycle time of the SFU and the number of clock cycles taken by the application. In principle, the minimum latency of a SFU is one clock cycle due to the input registers. As the SFUs may contain arbitrary number of pipeline stages, the cycle time can be decreased efficiently with increased latency. The tradeoff between TTA processors accompanied with SFUs and processors without SFUs depends heavily on the application. For example, if the compiler can efficiently map the desired functionality to conventional FUs and use them in a pipelined fashion, it can be possible that simple SFUs do not accelerate the application significantly but they still take some extra area.

When compared to pure hardware implementations the main benefit of processor based implementations is their programmability. If the data path of a TTA processor is optimized only for a single application it may resemble the data path of a pure hardware implementation. However, at least some programmability is still present and designing and programming a TTA processor with up-to-date tools takes typically less time than designing a pure hardware implementation with the same functionality.

2.2.3 Multiprocessor Systems with TTA Processors

The system example in Fig. 2 suggests that a fully functional receiver could be built as a heterogeneous multiprocessor system consisting of several TTA processors or other ASPs. Implementation techniques of multiprocessor systems are beyond the scope of this Thesis. In other words, the targeted functions are considered independently. Some requirements for an ideal inter-processor communication (IPC) of baseband processing are derived in Chapter 9 but also the implementation of IPC is left beyond the scope of this Thesis and an abstract multiprocessor system using shared memory banks or shared RFs as communication links is assumed.

However, as a related work, many multiprocessor systems applying TTA processors can be referred. In [58], a simple asynchronous communication link between TTA processors is enabled with units containing a first in, first out (FIFO) buffer. TTA and LEON3 processors are connected with an AMBA bus in [43]. On the contrary to a shared bus, a network-on-chip approach has been applied in [6] where two Coffee RISC processors, a TTA processor, and a shared memory are connected with a network. A bio-inspired multiprocessor system is presented in [109] and [90] where TTA processors are abstracted as cells of a biological system. Multiple processors could be also abstracted as a hierarchical structure where the SFUs would be comprised of TTA sub-processors. Another way would be to combine all the TTA processors to a set of loosely connected clusters inside a single TTA processor. However, assembly programming such a processor would be error prone due to the extremely long instruction word and the scheme would limit the control flow of the clusters very strictly to a single combined flow.

3. SPECIAL FUNCTION UNITS FOR VITERBI DECODING

In this Chapter, an ASP implementation of Viterbi decoder is addressed and SFUs suitable for continuous path metric computation are proposed. Further parallelization of the ACS computation with the proposed SFUs is mainly limited by the memory throughput of the applied processor. The SFUs are experimented with an implementation of a 256-state, rate 1/2 decoder. As a result, high utilization is achieved as the path metrics can be computed with a software-pipelined schedule.

3.1 Viterbi Decoding Principles

Viterbi decoding can be illustrated with the aid of trellis diagrams. In Fig. 9(a), a trellis diagram with N_{states}-states is shown. The diagram expresses, which state transitions are possible. The convolutional encoder has an internal state and the state transitions follow some path via trellis diagram as the input bits are encoded. The Viterbi algorithm [121] is used to restore the traversed path. In Fig. 9(b) a path via 4-state trellis is shown as an example.

Each state transition corresponds to a transmitted symbol. The number of bits in symbol depends on the code rate. Based on the received symbol estimates the path metrics are computed for each state. The path metric is the minimum of previous state path metrics, to which a branch metric describing the distance between symbol estimate and a symbol corresponding to respective state transition is added. For each state, information describing the selected state transition which ends in the current state is saved. After the path metrics have been computed for a code block, a trace-back procedure traverses the states backwards and restores the state transition path and information bits corresponding to the path.

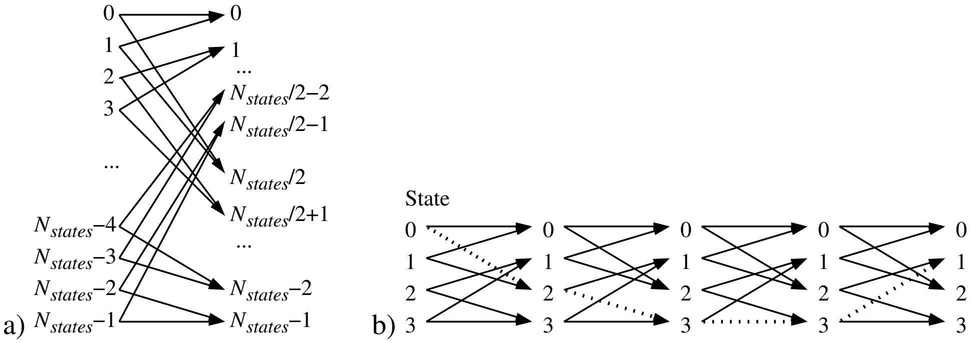


Fig. 9. *Trellis diagrams: a) an example of N_{states}-state trellis shows valid state transitions. b) An example path via states 0, 2, 3, 3, and 1 of a 4-state trellis is shown.*

3.2 Previous Work

Several Viterbi decoder implementations on processor platforms have been reported. Roughly, the processor based implementations can be divided into three categories. Firstly, the Viterbi algorithm can be implemented with conventional DSPs, which are often augmented with units alleviating fast decoding. Secondly, the processor can be accompanied by a co-processor, which performs most of the decoding task. Thirdly, a very long instruction word (VLIW) processor can utilize its inherent parallelism and dedicated units to speed up the decoding. Naturally, the aforementioned DSP and VLIW categories can overlap.

In [49], a Viterbi decoder is implemented on C54x DSP. The implementation utilizes compare, select, and store unit (CSSU) which is incorporated into the datapath of the processor. Also with the C55x DSP the same principles can be used but it has greater parallelism [111]. The drawback of this approach is that the processor has rather limited parallelism for the Viterbi algorithm, thus the inherent parallelism of the algorithm can be exploited only partially. The performance can be improved by including Viterbi decoder co-processor (VCP) into DSP like in C6416 DSP [113]. The drawback of this approach is that the implementation is strictly separated from the host processor and it is monolithic, i.e., it has a memory of its own and the user cannot have a full control of the internal resources of the VCP. A VLIW DSP core SPXK5 is presented in [61]. The processor has seven FUs, of which four units can operate in parallel, and it has highly orthogonal instruction set, i.e., for each appropriate instruction there are no hard restrictions on the choice of registers and addressing mode. Due to its parallelism, it lends itself also to Viterbi decoding. The IA32 architecture is also capable of efficient Viterbi decoding with streaming SIMD (Single Instruction Multiple Data) extension (SSE) instruction set [51].

Finally, there exist variety of intellectual property (IP) cores for Viterbi decoding [8, 94, 100]. Naturally, the dedicated core cannot be used for other tasks like processors. As a benefit, the core can be optimized only for Viterbi algorithm. Therefore, it may have shorter word length than the processor, which decreases the complexity of the datapath efficiently.

In this Chapter, the main computations, i.e., computation of path metrics, are addressed and SFUs are designed for maintaining a continuous software-pipelined ACS computation. In addition to actual ACS operation, the SFUs feed operands and save the results continuously, so that the memory throughput is the main limiting factor of further parallelization. The proposed SFUs are experimented with 256-state code and an ASP with dual-port memory. With the proposed SFUs, the inherent parallelism of the algorithm can be exploited. As a second benefit, the SFUs are designed to be tightly integrated within the datapath. On the contrary to the monolithic co-processor, the SFUs can be fully controlled and used also for other tasks than the intended Viterbi decoding. Furthermore, the implementation does not require additional dedicated memory for Viterbi decoding. As a result of high resource utilization, computation of one radix-2 ACS operation takes only one clock cycle with dual memory accesses.

3.3 Special Function Units

The SFUs are designed to support rapid radix-2 ACS computations. In addition to the actual ACS, pre- and post-processing of operands and results is required for a continuous computation.

3.3.1 Add Compare Select Unit

The structure of the ACSU is straightforward as there is a direct mapping between the computations and computing resources. The structure of the ACSU is illustrated in Fig. 10 where the path metrics of kth trellis stage and ith state are denoted with

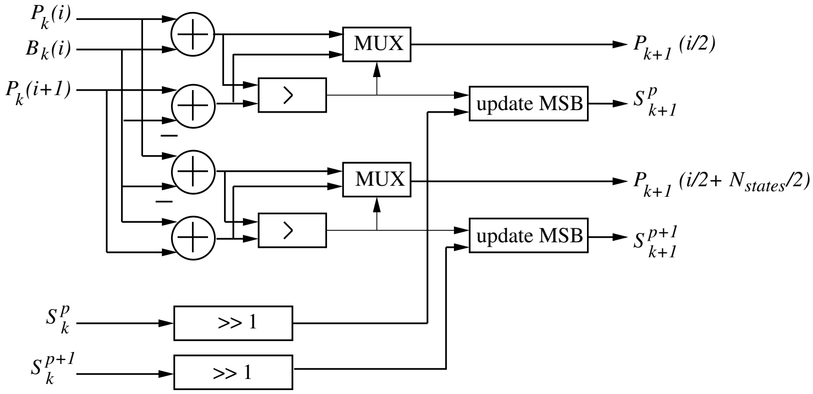


Fig. 10. *Radix-2 ACSU with single branch metric,* $B_k(i)$*, input and survivor path information,* S^p_k*, update circuit. The path metrics are denoted with* $P_k(i)$ *and right shift with* $>> 1$.

$P_k(i)$ and the corresponding branch metrics with $B_k(i)$. Due to the structure of the experimented trellis, one branch metric input port can be avoided if two additions are substituted with subtractions as shown in Fig. 10. This practice alleviates SFU instantiation in ASPs as there are less operands to pass and, therefore, lower internal data transfer overhead when the SFU is used. As the SFU is targeted to 32-bit ASPs, two 16-bit path metrics and two 16-bit branch metrics are packed into 32-bit words. The correct branch metric is selected with the operation code of the SFU.

The SFU must also save the information about the selected branch. The SFU in Fig. 10 packs the bits indicating the selection in 32-bit words by shifting the survivor path information word one bit right and, thereafter, updating the most significant bit (MSB). On the contrary to simple hardwired logic, such bit-wise operations would take at least two clock cycles with conventional instruction set. Since the selections are required for each state for each processed trellis stage before the traceback routine, it is advantageous to pack them to avoid extensive memory consumption. The pth 32-bit word for packing survivor path information is denoted with S^p_k in Fig. 10.

3.3.2 Branch Metric Generation Unit

In principle, the symbol alphabet depends on the code rate. For example, with code rate 1/2, there are two bits per transmitted symbol, i.e., the alphabet is $\{00, 01, 10, 11\}$. The branch metrics describe distances between the received symbols and the ele-

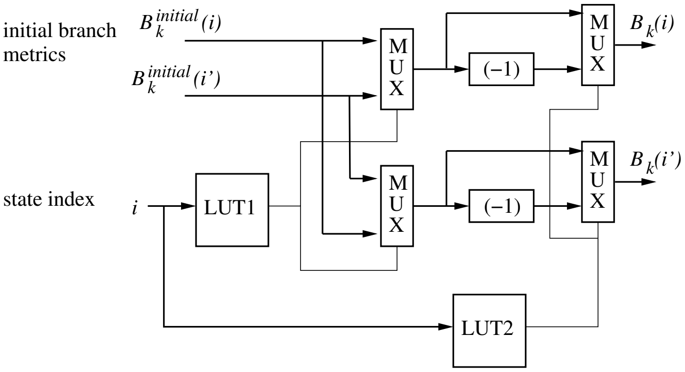


Fig. 11. *Branch metrics are generated by selecting the metric corresponding to the transmitted symbol according to the index of the ACS operation. Negating the value is denoted with* (-1).

ments of symbol alphabet. However, the branch metrics corresponding to symbols with complemented bits have negated values with respect to each other. Therefore, they can be computed with the aid of only two initial branch metrics, $B_k^{initial}(i)$ and $B_k^{initial}(i')$. This practice is applied in the branch metric generation unit in Fig. 11. The figure shows how the state index, i, of the radix-2 ACS operations is used to select the correct branch metric, i.e., the trellis states, for which branch metrics are generated, control the multiplexers via look-up-tables (LUT).

3.3.3 Path Metrics Packing Unit

The function of the path metrics packing unit in Fig. 12 is to organize data in such a way that it can be read and written in one access cycle. With 32-bit wide data bus, two adjacent 16-bit path metrics should be read or written at once. Therefore, the accessed path metrics should always be adjacent even if the results of the ACSU are not mapped to adjacent memory locations.

The diagram in Fig. 9(a) indicates that with the exemplified N_{states}-state trellis, the radix-2 ACSU, and even state index i, processing the path metrics of the states i and $i+1$ generates new path metrics for the states $i/2$ and $(N_{states}+i)/2$ for the next trellis stage. Thus, the generated path metrics are not mapped to adjacent memory locations. However, if the computation is followed with another radix-2 ACS operation for states $i+2$ and $i+3$, the new path metrics for states $(i+2)/2 = i/2+1$

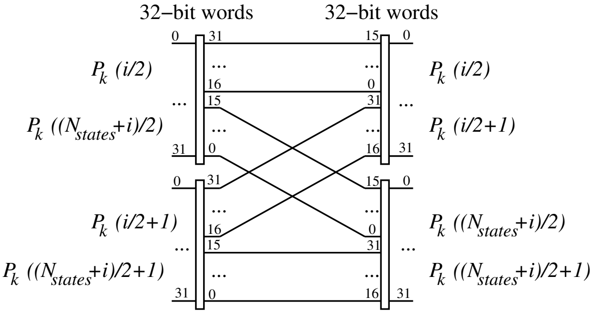

32–bit words
32–bit words
P_k (i/2)
P_k ((N_{states}+i)/2)
P_k (i/2+1)
P_k ((N_{states}+i)/2+1)
P_k (i/2)
P_k (i/2+1)
P_k ((N_{states}+i)/2)
P_k ((N_{states}+i)/2+1)
0
31
15
16
...

Fig. 12. *Path metrics packing SFU alleviates efficient usage of 32-bit load and store operations.*

and $(N_{states} + i + 2)/2 = (N_{states} + i)/2 + 1$ are adjacent with previously generated path metrics. Thus, storing the path metrics can be delayed for one clock cycle and thereafter 16-bit path metrics for adjacent states can be packed to a single 32-bit word which can be stored in one clock cycle. The SFU in Fig. 12 is used for the required interchanging of half words. Again, with conventional instruction set such interchanging would require shifting and masking operations taking several clock cycles.

3.3.4 Address Generation Unit

The address generation unit generates addresses for loading and storing the path metrics. The principal computation flow of the unit is shown in Fig. 13. The SFU is targeted for byte-wise addressing, i.e., the minimum distance between 32-bit words

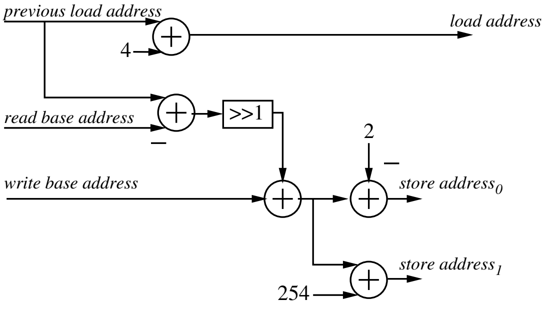


Fig. 13. *Address generation unit generates one load address and two store addresses. Right shift is denoted with* $>> 1$.

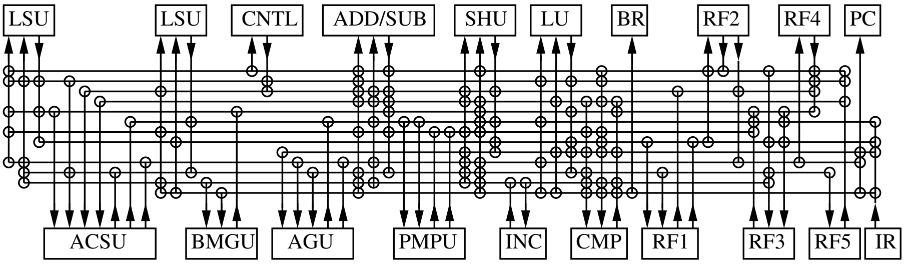


Fig. 14. *Viterbi decoder TTA processor. Controllable connections between resources and buses are denoted with circles.* `AGU`*: address generation unit,* `BMGU`*: branch metric generation unit,* `PMPU`*: path metric packing unit,* `LSU`*: load store unit,* `CNTL`*: status word,* `ADD/SUB`*: addition and subtraction,* `SHU`*: left and right shifting,* `LU`*: logical operations,* `CMP`*: comparison,* `INC`*: increment by one,* `BR`*: boolean register,* `PC`*: program counter,* `IR`*: long immediate register,* `RF1`*...* `RF5`*: integer register files.*

in memory is four address units. The load address operand is fed back to the unit via internal buses of the processor. On the contrary to load address, the base address operands of the SFU are locally constants and they can be saved in the registers of the input ports of the SFU. Since the path metrics are stored in two phases there are two different address generation functions for store addresses. The first one generates addresses for the states $0, 1, \ldots, N_{states}/2 - 1$ and the next one for the states $N_{states}/2, N_{states}/2 + 1, \ldots, N_{states} - 1$.

3.4 Viterbi Decoder Implementation

The SFUs are applied on a TTA [32] processor developed and programmed with the original MOVE toolset [33]. A high-level block diagram of the Viterbi decoder TTA processor is shown in Fig. 14. The processor has two LSUs which determine the memory throughput which limits maximum parallelism of path metric computations. In addition to the proposed SFUs, the processor has conventional FUs for addition/subtraction, comparison, shifting, and logic operations.

The SFUs and the processor are synthesized with 130 nm technology and area estimates are given in terms of logic gate equivalents (GE). The resources and performance of the TTA processor applying the proposed SFUs are summarized in Table 1. The area of the SFUs is given for independent units, i.e., they are not connected to any other logic which would increase the critical path and result in higher area. De-

Table 1. *Characteristics of the SFUs and Viterbi decoder TTA processor with 100 MHz clock frequency.*

Area of the ACSU	1.4 kGEs
Area of the address generation unit	2.2 kGEs
Area of the path metric packing unit	0.7 kGEs
Area of the branch metric generation unit	0.7 kGEs
Clock cycles per bit ($N_{states} = 256$)	175 cycles / bit
Decoding rate	0.57 Mbps
Number of internal buses	13
Area of the TTA processor	46 kGEs

coding a code with 256-state trellis require 128 radix-2 ACS operations and with the proposed SFUs the radix-2 ACS operations can be computed continuously at rate one operation per clock cycle. Thus, the traceback routine and any extra overhead take $175 - 128 = 47$ clock cycles per trellis stage. The number of buses in Table 1 gives the minimum parallelism in terms of internal data transports for applying the proposed SFUs efficiently.

3.4.1 Decoder Program

The core of the decoder program consists of two nested loops which are followed by a traceback routine as shown in Fig. 15. The path metric memory is divided into two parallel accessible regions, whose roles are interchanged as the decoding proceeds to the next trellis stage. The decoder is programmed with C but the innermost loop is optimized manually and programmed with parallel assembly language.

```
loop (TRACEBACK_DEPTH)
        call Swap_path_metric_memory_pointers
        loop (32)
                call Generate_branch_metrics
                call ACS(1)
                call ACS(2)
                call ACS(3)
                call ACS(4)
        end
        call Update_survivor_path_memory
end
loop (TRACEBACK_DEPTH)
        call Traceback
end
```

Fig. 15. *Pseudo code of the core decoder program.*

0: AGU$_l$→LD LD→ACSU r$_{s01}$→ACSU R$_{s11}$→ACSU BMGU→ACSU ACSU$_p$→PMPU
PMPU$_l$→ST AGU$_s$→ST AGU$_l$→AGU ACSU$_{s0}$→R$_{s00}$ ACSU$_{s1}$→R$_{s10}$ INC→INC BR→PC

1: AGU$_l$→LD LD→ACSU R$_{s02}$→ACSU R$_{s12}$→ACSU BMGU→ACSU ACSU$_p$→PMPU
PMPU$_h$→ST AGU$_s$→ST AGU$_l$→AGU ACSU$_{s0}$→R$_{s01}$ ACSU$_{s1}$→R$_{s11}$ INC→CMP$_0$
30→CMP$_1$

2: AGU$_l$→LD LD→ACSU R$_{s03}$→ACSU R$_{s13}$→ACSU BMGU→ACSU ACSU$_p$→PMPU
PMPU$_l$→ST AGU$_s$→ST AGU$_l$→AGU ACSU$_{s0}$→R$_{s02}$ ACSU$_{s1}$→R$_{s12}$ INC→BMGU$_i$
R$_{bm}$→BMGU$_b$

3: AGU$_l$→LD LD→ACSU R$_{s00}$→ACSU R$_{s10}$→ACSU BMGU→ACSU ACSU$_p$→PMPU
PMPU$_h$→ST AGU$_s$→ST AGU$_l$→AGU ACSU$_{s0}$→R$_{s03}$ ACSU$_{s1}$→R$_{s13}$ CMP→BR

Fig. 16. *Pseudo assembly code of the main loop kernel.*

A pseudo assembly listing of the loop kernel is given in Fig. 16. Due to the jump latency the loop contains four instruction words. The parallel data transports are indicated by → in Fig. 16. The left and right hand side of the move operation are the source and destination resources. The subscripts indicate the output port of the units or identifies the register. The move BR→PC is a conditional branch, i.e., guarded modification of the program counter. The first move provides load address to the load unit. Second move passes previously loaded path metric to the ACSU. Third and fourth moves pass survivor path operands to the ACSU. Fifth move passes branch metrics to the ACSU. Sixth move passes previously computed path metrics to the path metric packing unit and the next move passes packed path metrics to the store unit which gets the store address by eighth move. Previous load address is fed back to the address generation unit in ninth move. The updated survivor path information is stored into registers by the next two moves.

3.4.2 Discussion

In principle, multiple sets of the proposed SFUs could be applied in parallel for achieving higher throughput. In this case, each set of the SFUs would decode the same trellis stage with different offset of the state index. Naturally, such a practice would require higher memory throughput as each set would load and store path metrics in parallel.

Finally, different processor architectures can be characterized by analyzing how they lend themselves to ACS computations. In Table 2, the ACS computation with C54x

Table 2. *Radix-2 ACS computations with different processor architectures.*

Architecture	Cycles	Resources	Instructions of the ACS computation
TTA with SFUs	1	ACSU	7 parallel moves
C54x [110]	4	ALU and CSSU	2 dual addsub, 2 compare select store
C55x [111]	3	ALU	2 dual addsub, 1 dual comparison
C64 VCP [4]	0.25	4 ACSUs	N/A
SPXK5 [61]	2	2 ALUs	1 quadruple addsub, 1 dual maximum

DSP [110], C55x DSP [111], VCP of C64x DSP [4], and a VLIW processor [61] are exemplified. The table shows that typical DSPs have sufficient instructions for efficient path metric computation. However, their computation requires more clock cycles than computation with proposed SFUs. The VCP in Table 2 contains dedicated ACSUs but a monolithic structure requires dedicated memory. The proposed SFUs can be integrated in the data path of the ASP and, therefore, conventional memory accesses can be used.

The throughput of the ACS computation with processors in Table 2 is directly proportional to the clock frequency and inversely proportional to the cycles in the second column. For example, even if the C55x requires three clock cycles per radix-2 ACS, its throughput is comparable with TTA with the SFUs as the C55x can achieve 300 MHz clock frequency [115]. The level of parallelism used for the ACS computation is indicated by the Resources and Instructions of the ACS computation columns in Table 2. For example, the SPXK5 VLIW processor uses two arithmetic logic units (ALU) each capable of dual operations. In the first clock cycle, two additions and two subtractions are executed in parallel in the two ALUs. In the next clock cycle, two maximum operations are executed in parallel.

The ACS computation with a TTA processor without any SFUs depends heavily on the available FUs, RFs, interconnection network, and on the program. Therefore, computation with such a processors is exemplified with the number of basic operations carried out by the SFUs in Table 3. The rightmost column in Table 3 shows the number of clock cycles if there are sufficient FUs for each basic operation of the second column, i.e., no resource multiplexing is required and only the data dependencies and the assumed one clock cycle latency of the FUs limits the total delay. Naturally, even if several clock cycles are taken in total, the operations can be computed in a pipelined manner when there are no restricting data dependencies.

Table 3. *Computation of accelerated functions with conventional operations.*

SFU operation	Basic operations	Cycles with max parallelism
Radix-2 ACS and survivor path update	2 add, 2 sub, 2 cmp 2 shift, 2 or	3
Address generation	3 add, 2 sub, 1 shift	4
Path metric packing	2 shift, 2 and, 2 or	2
Branch metric generation	2 shift, 2 and, 2 sub	3

4. MULTIPLEXED ACSUS FOR MAX-LOG-MAP COMPUTATION

Max-log-MAP algorithm, a simplification of log-MAP algorithm, can be used in SISO component decoding of turbo decoders. In this Chapter, it is shown how all the computations of the max-log-MAP algorithm can be computed by time-multiplexing a slightly modified ACSU. As a result of mapping all the computations to a single type of resource, the dimensions, in terms of parallel computing resources, of the design space are reduced as different decoder structures can be derived by varying only a single parameter, i.e., the number of ACSUs. Obviously, such a reduction of dimensions simplifies the design of max-log-MAP decoders. The proposed method is applied in practice as SFUs for a decoder compatible with the eight state trellis and 2.0 Mbps data rate of Universal Mobile Telecommunication System (UMTS) [1] are developed and applied in an ASP.

4.1 Parallelism and Throughput

Basically, the throughput of the max-log-MAP SISO component decoder of the turbo decoder is proportional to the level of parallelism. With radix-2 algorithms the level of parallelism can be increased up to a level where one trellis stage is processed in a one clock cycle, i.e., a stage-parallel decoding. Thereafter, the recursive data dependencies prevent efficient utilization of additional resources. Fig. 17 illustrates this relation. In one extreme, a simple processor with a limited instruction set can sequentially carry out all the computations. In the other extreme, a stage-parallel decoder carries out all the computations per trellis stage in parallel.

Many implementations reside close to the two extremes. A fully sequential program describing turbo decoding is easy to develop as there are no parallel operations with conflicting data dependencies. However, depending on the applied technology, it is possible that the desired data rate can be achieved between the two extremes. The

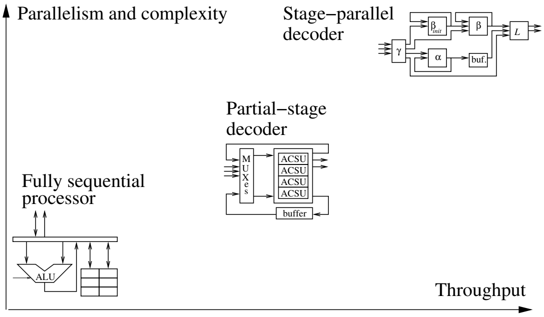


Fig. 17. *The throughput increases as parallel computing resources are increased until a component decoder processing one trellis stage in one clock cycle is achieved.*

derivation in this Chapter shows how all the computations of max-log-MAP algorithm can be mapped to slightly modified ACSUs. The main benefit of this notion is that it simplifies the design of decoders residing between the two extremes in Fig. 17. As all the computations can be mapped to a single type of resource, part of the designing of a decoder is reduced to scheduling and exploring of a one type of a resource. Naturally, designing with a set of homogeneous resources is simpler than with heterogeneous resources.

The increasing level of parallelism and throughput as presented in Fig. 17 corresponds also with increasing memory bandwidth requirements. The partial-stage turbo decoder processor presented in this Chapter has such a high parallelism that it requires parallel memory accesses which cannot be met with simple dual-port memory. However, the memory accesses can be mapped to separate memory banks straightforwardly. On the contrary, the stage-parallel decoder in Chapter 5 has even higher memory bandwidth requirements and it requires more sophisticated parallel memory access methods which are presented in Chapter 6.

4.2 Max-Log-MAP Algorithm

The max-log-MAP algorithm is a simplified, approximative, derivation of the MAP algorithm [17]. Basically, max-log-MAP algorithm [41, 60, 88] can be divided into four computation tasks, which are branch metrics generation, forward metrics generation, backward metrics generation, and generation of soft or hard bit estimates together with new extrinsic information. The forward path metric of state u at trellis stage k, $\alpha_k(u)$, is defined recursively as

$$\alpha_k(u) = \max_{u' \in U_{pred}(u)} (\alpha_{k-1}(u') + d_k(u', u)) \tag{7}$$

where $d_k(u', u)$ is the branch metrics, u' is the previous state, and the set $U_{pred}(u)$ contains all the predecessor states of u, i.e., the states from which there is a state transition to the state u [122]. Respectively, the backward path metrics are defined as

$$\beta_{k-1}(u') = \max_{u \in U_{succ}(u')} (\beta_k(u) + d_k(u', u)) \tag{8}$$

where the set $U_{succ}(u')$ contains all the successor states of state u' [122].

The soft output, L_k, is computed with the aid of the forward, backward, and branch metrics as a difference of two maximums [122]. In the following, the minuend maximum corresponds to the state transitions with transmitted systematic bit $x^s = 0$ and the subtrahend maximum corresponds to the state transitions with $x^s = 1$,

$$\begin{aligned} L_k = &\max_{(u',u):x^s=0} (\alpha_{k-1}(u') + \beta_k(u) + d_k(u', u)) - \\ &\max_{(u',u):x^s=1} (\alpha_{k-1}(u') + \beta_k(u) + d_k(u', u)). \end{aligned} \tag{9}$$

The hard bit estimate is obtained simply by the signum function, $\mathrm{sgn}(\cdot)$, of the L_k. The new extrinsic information λ_k^{out} is computed with the aid of L_k, i.e., *a posteriori* information λ_k^{out} is obtained as

$$\lambda_k^{out} = \frac{1}{2} L_k - y_k^s - \lambda_k^{in} \tag{10}$$

where λ_k^{in} is the *a priori* information, and y_k^s is the received soft systematic bit, i.e., y_k^s can have positive or negative non-integer values [122].

4.3 Re-Organized Forms for Efficient Resource Mapping

Mapping of computations of forward and backward metrics is trivial, since the recursive computation in (7) and (8) are already in a form suitable for ACS operations. The only re-organization is required if the maximums in (7) and (8) contain more arguments than the targeted ACS operation takes, e.g., there are more than two state transitions (u', u) leading to a state and radix-2 ACS operations are targeted. In such a case, the computations must be split and carried out in steps consisting of maximum operations with two arguments.

The computation of soft output, L_k, must be re-organized to several steps. First, the computations are grouped by applying the fact that

$$\begin{aligned} &\max_{(u',u)\in U_{sp}} (\alpha_{k-1}(u') + \beta_k(u) + d_k(u',u)) \\ &= \max_{(u',u)\in U_{sp}} (\alpha_{k-1}(u') + \beta_k(u)) + d_k(v',v) \text{ where } (v',v) \in U_{sp} \end{aligned} \tag{11}$$

where the set U_{sp} consists of all the state transitions (u',u) which correspond with the same combination of systematic and parity bits. As the last term is moved outside of the maximum in (11), the states of the last term, $d_k(v',v)$, can be defined by any of such a state pair in the set U_{sp}. This re-organization is applied for each U_{sp}, i.e., for all the systematic and parity bit combinations. Thus,

$$\begin{aligned} L_k = &\max_{(u',u):x^s=0} (\alpha_{k-1}(u') + \beta_k(u) + d_k(u',u)) - \\ &\max_{(u',u):x^s=1} (\alpha_{k-1}(u') + \beta_k(u) + d_k(u',u)) \\ = &\max_{U_{sp}:x^s=0} \left(\max_{(u',u)\in U_{sp}} (\alpha_{k-1}(u') + \beta_k(u)) + d_k(U_{sp}(0)) \right) - \qquad (12) \\ &\max_{U_{sp}:x^s=1} \left(\max_{(u',u)\in U_{sp}} (\alpha_{k-1}(u') + \beta_k(u)) + d_k(U_{sp}(0)) \right) \qquad (13) \\ = &L_k^0 - L_k^1 \qquad (14) \end{aligned}$$

where the state pair sets, U_{sp}, in (12) correspond with state transitions in which transmitted systematic bit $x^s = 0$ and in (13) the corresponding systematic bit $x^s = 1$. As any branch metric of the respective set U_{sp} can be used, the first state pair, which is indexed with zero, is used in the term $d_k(U_{sp}(0))$ in (12) and (13). The maximums in (12) and (13) are denoted as L_k^0 and L_k^1, respectively, in (14) to clarify further derivations. Next, the maximums with multiple arguments can be split to maximums of two arguments, which can be mapped straightforwardly to ACS operations.

Computation of new extrinsic information λ_k^{out} is defined with the aid of L_k in (10). However, it can be computed in parallel with the last step of the computation of the L_k. In (10) a sum $y_k^s + \lambda_k^{in}$ is required. It can be obtained from branch metrics with

$$d_k(u',u) + d_k(v',v) = 2y_k^s + 2\lambda_k^{in} \tag{15}$$

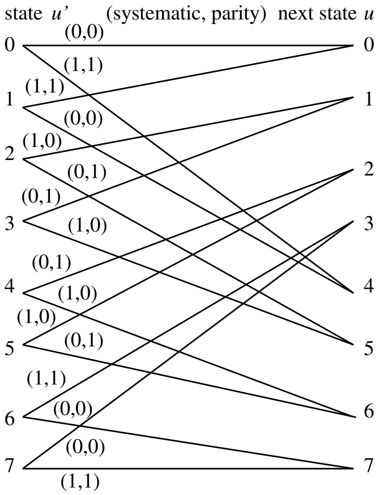


Fig. 18. *Trellis of eight state 3GPP constituent code. Transmitted systematic and parity bit pairs (x^s, x^p) correspond with state transitions of the component encoder.*

where the branch metrics corresponding with state transitions (u', u) and (v', v) are chosen so that they correspond with transmitted systematic bit $x^s = 0$ and parity bits with respectively opposite signs. This step is further illustrated with an example case in (18)–(21) and (30). With this practice the parity bits cancel each other in the sum. With this substitution and with previously chosen (u', u) and (v', v) the (10) is expressed as

$$\begin{aligned}\lambda_k^{out} &= \frac{1}{2}(L_k - (d_k(u', u) + d_k(v', v))) \\ &= \frac{1}{2}(L_k^0 - L_k^1 - d_k(u', u) - d_k(v', v)).\end{aligned} \tag{16}$$

However, it is also possible to use a state transition (w', w) corresponding to the branch metric with negated value, i.e., $d_k(w', w) = -d_k(u', u)$. Thus,

$$\lambda_k^{out} = \frac{1}{2}(L_k^0 + d_k(w', w) - (L_k^1 + d_k(v', v))), \tag{17}$$

which is a more suitable form for resource mapping.

4.3.1 Example with 3GPP Constituent Code

With the presented method, the structure of the max-log-MAP decoder is simplified and the computing resources can be shared economically. The eight state trellis of

3GPP constituent code in Fig. 18 is used as an example case to clarify the derivation. Since the trellis is fixed, a more suitable presentation for the branch metrics can be used. There are only eight states and four possible systematic and parity bit combinations in Fig. 18. However, there are 16 possible branches expressed as $d_k(u', u)$. All the branch metrics $d_k(u', u)$ correspond to the transmitted systematic and parity bit pairs (x_k^s, x_k^p) and, therefore, the branch metrics notation can be defined also with the following four symbols

$$\gamma_k^{00} = \underset{x_k^s=0, x_k^p=0}{d_k(u', u)} = y_k^s + \lambda_k^{in} + y_k^p \tag{18}$$

$$\gamma_k^{01} = \underset{x_k^s=0, x_k^p=1}{d_k(u', u)} = y_k^s + \lambda_k^{in} - y_k^p \tag{19}$$

$$\gamma_k^{10} = \underset{x_k^s=1, x_k^p=0}{d_k(u', u)} = -y_k^s - \lambda_k^{in} + y_k^p \tag{20}$$

$$\gamma_k^{11} = \underset{x_k^s=1, x_k^p=1}{d_k(u', u)} = -y_k^s - \lambda_k^{in} - y_k^p \tag{21}$$

where the received soft parity bit is y_k^p. Previous notation shows how the branch metrics are computed. Since the branch metrics with complemented indices are negations of each other, computing only two branch metrics is sufficient if respective additions in (7)–(9) were substituted with subtractions.

The computation of forward and backward metrics is straightforward by following the state transitions in Fig. 18. For example,

$$\alpha_k(0) = \max(\alpha_{k-1}(0) + \gamma_k^{00}, \alpha_{k-1}(1) + \gamma_k^{11}) \tag{22}$$

$$\alpha_k(4) = \max(\alpha_{k-1}(0) + \gamma_k^{11}, \alpha_{k-1}(1) + \gamma_k^{00}) \tag{23}$$

$$\beta_{k-1}(4) = \max(\beta_k(2) + \gamma_k^{01}, \beta_k(6) + \gamma_k^{10}) \tag{24}$$

$$\beta_{k-1}(5) = \max(\beta_k(2) + \gamma_k^{10}, \beta_k(6) + \gamma_k^{01}). \tag{25}$$

The soft output, L_k, in (9) can be written with the aid of branch metrics, $\gamma_k^{00\ldots11}$, as

$$\begin{aligned} L_k = \max(\, & \alpha_{k-1}(0) + \beta_k(0) + \gamma_k^{00}, \alpha_{k-1}(1) + \beta_k(4) + \gamma_k^{00}, \\ & \alpha_{k-1}(2) + \beta_k(5) + \gamma_k^{01}, \alpha_{k-1}(3) + \beta_k(1) + \gamma_k^{01}, \\ & \alpha_{k-1}(4) + \beta_k(2) + \gamma_k^{01}, \alpha_{k-1}(5) + \beta_k(6) + \gamma_k^{01}, \\ & \alpha_{k-1}(6) + \beta_k(7) + \gamma_k^{00}, \alpha_{k-1}(7) + \beta_k(3) + \gamma_k^{00}) - \\ \max(\, & \alpha_{k-1}(0) + \beta_k(4) + \gamma_k^{11}, \alpha_{k-1}(1) + \beta_k(0) + \gamma_k^{11}, \end{aligned}$$

$$\begin{aligned}&\alpha_{k-1}(2)+\beta_k(1)+\gamma_k^{10},\alpha_{k-1}(3)+\beta_k(5)+\gamma_k^{10},\\&\alpha_{k-1}(4)+\beta_k(6)+\gamma_k^{10},\alpha_{k-1}(5)+\beta_k(2)+\gamma_k^{10},\\&\alpha_{k-1}(6)+\beta_k(3)+\gamma_k^{11},\alpha_{k-1}(7)+\beta_k(7)+\gamma_k^{11}).\end{aligned}\tag{26}$$

Common $\gamma_k^{x^s x^p}$ addends can be extracted from maximum operations as follows,

$$\begin{aligned}L_k=\max(&\max(\alpha_{k-1}(0)+\beta_k(0),\alpha_{k-1}(1)+\beta_k(4),\\&\qquad\alpha_{k-1}(6)+\beta_k(7),\alpha_{k-1}(7)+\beta_k(3))+\gamma_k^{00},\\&\max(\alpha_{k-1}(2)+\beta_k(5),\alpha_{k-1}(3)+\beta_k(1),\\&\qquad\alpha_{k-1}(4)+\beta_k(2),\alpha_{k-1}(5)+\beta_k(6))+\gamma_k^{01})-\\\max(&\max(\alpha_{k-1}(0)+\beta_k(4),\alpha_{k-1}(1)+\beta_k(0),\\&\qquad\alpha_{k-1}(6)+\beta_k(3),\alpha_{k-1}(7)+\beta_k(7))+\gamma_k^{11},\\&\max(\alpha_{k-1}(2)+\beta_k(1),\alpha_{k-1}(3)+\beta_k(5),\\&\qquad\alpha_{k-1}(4)+\beta_k(6),\alpha_{k-1}(5)+\beta_k(2))+\gamma_k^{10}).\end{aligned}\tag{27}$$

Next, for brevity, maximums are denoted with variables $s_k^{0\ldots3}$ and $t_k^{0\ldots3}$ as follows,

$$\begin{aligned}s_k^0&=\max(\alpha_{k-1}(0)+\beta_k(0),\alpha_{k-1}(1)+\beta_k(4))\\t_k^0&=\max(\alpha_{k-1}(0)+\beta_k(4),\alpha_{k-1}(1)+\beta_k(0))\\s_k^1&=\max(\alpha_{k-1}(6)+\beta_k(7),\alpha_{k-1}(7)+\beta_k(3))\\t_k^1&=\max(\alpha_{k-1}(6)+\beta_k(3),\alpha_{k-1}(7)+\beta_k(7))\\s_k^2&=\max(\alpha_{k-1}(2)+\beta_k(5),\alpha_{k-1}(3)+\beta_k(1))\\t_k^2&=\max(\alpha_{k-1}(2)+\beta_k(1),\alpha_{k-1}(3)+\beta_k(5))\\s_k^3&=\max(\alpha_{k-1}(4)+\beta_k(2),\alpha_{k-1}(5)+\beta_k(6))\\t_k^3&=\max(\alpha_{k-1}(4)+\beta_k(6),\alpha_{k-1}(5)+\beta_k(2)).\end{aligned}$$

Next, the soft output, L_k, can be further simplified to

$$\begin{aligned}L_k=&\max(\max(s_k^0,s_k^1)+\gamma_k^{00},\max(s_k^2,s_k^3)+\gamma_k^{01})-\\&\max(\max(t_k^0,t_k^1)+\gamma_k^{11},\max(t_k^2,t_k^3)+\gamma_k^{10})\end{aligned}\tag{28}$$

and with new variables s_k^4, s_k^5, t_k^4, and t_k^5

$$\begin{aligned}s_k^4&=\max(s_k^0,s_k^1)\ ,\ s_k^5=\max(s_k^2,s_k^3)\\t_k^4&=\max(t_k^0,t_k^1)\ ,\ t_k^5=\max(t_k^2,t_k^3)\end{aligned}$$

it can be written in the following form

$$\begin{aligned} L_k = &\max(s_k^4 + \gamma_k^{00}, s_k^5 + \gamma_k^{01}) - \\ &\max(t_k^4 + \gamma_k^{11}, t_k^5 + \gamma_k^{10}), \end{aligned} \tag{29}$$

whose intermediate terms are defined as s_k^6 and t_k^6,

$$\begin{aligned} s_k^6 &= \max(s_k^4 + \gamma_k^{00}, s_k^5 + \gamma_k^{01}) \\ t_k^6 &= \max(t_k^4 + \gamma_k^{11}, t_k^5 + \gamma_k^{10}). \end{aligned}$$

Computation of new extrinsic information, λ_k^{out}, can be defined with the aid of s_k^6 and t_k^6 instead of using L_k. First, the sum $y_k^s + \lambda_k^{in}$ is obtained from branch metrics, since

$$\begin{aligned} \gamma_k^{00} + \gamma_k^{01} &= y_k^s + \lambda_k^{in} + y_k^p + y_k^s + \lambda_k^{in} - y_k^p \\ &= 2y_k^s + 2\lambda_k^{in}. \end{aligned} \tag{30}$$

With this substitution to (10)

$$\begin{aligned} \lambda_k^{out} &= \frac{1}{2}(L_k - (\gamma_k^{00} + \gamma_k^{01})) \\ &= \frac{1}{2}(s_k^6 - t_k^6 - \gamma_k^{00} - \gamma_k^{01}). \end{aligned} \tag{31}$$

The values of branch metrics with complement indices are negated with respect to each other, i.e., $\gamma_k^{11} = -\gamma_k^{00}$. Thus,

$$\lambda_k^{out} = \frac{1}{2}(s_k^6 + \gamma_k^{11} - (t_k^6 + \gamma_k^{01})). \tag{32}$$

To compute the branch metrics, they are grouped as follows,

$$\gamma_k^{00} = (y_k^s + \lambda_k^{in}) + y_k^p \tag{33}$$

$$\gamma_k^{01} = (y_k^s + \lambda_k^{in}) - y_k^p \tag{34}$$

$$\gamma_k^{10} = y_k^p - (y_k^s + \lambda_k^{in}) \tag{35}$$

$$\gamma_k^{00} = -y_k^p - (y_k^s + \lambda_k^{in}) \tag{36}$$

where intermediate terms can be defined with the aid of s_k^7 and t_k^7 as,

$$s_k^7 = y_k^s + \lambda_k^{in} \tag{37}$$

$$t_k^7 = -y_k^p. \tag{38}$$

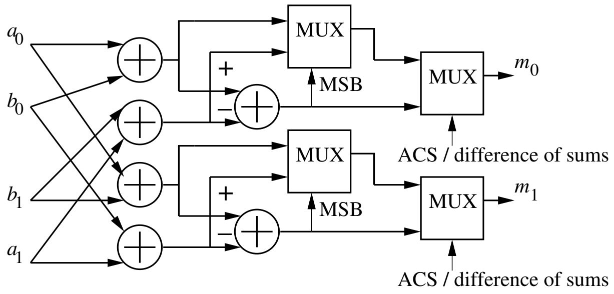


Fig. 19. *Proposed ACSU can compute difference of sums.*

4.4 Decoding with Modified ACSUs

In this Section, a slightly modified ACSU and decoding with the proposed ACSUs are presented.

4.4.1 Modified Add Compare Select Unit

The structure of the proposed ACSU is shown in Fig. 19. When compared to conventional ACSU, it is enhanced with an additional multiplexer which allows subtraction operations. With ACS mode selected, the ACSU computes

$$m_0 = \max(a_0 + b_0, a_1 + b_1) \tag{39}$$

$$m_1 = \max(a_0 + b_1, a_1 + b_0) \tag{40}$$

and with a difference of sums mode, it computes

$$m_0 = a_0 + b_0 - (a_1 + b_1) \tag{41}$$

$$m_1 = a_0 + b_1 - (a_1 + b_0). \tag{42}$$

Thus, the unit is capable of computing: 1) radix-1 ACS, 2) radix-2 ACS, 3) maximum, 4) sum, 5) difference, or 6) difference of sums. For example, when computing the maximum of a_0 and a_1, the operands b_0 and b_1 are set to zero.

4.4.2 Decoding Example

The 3GPP constituent code is used as an example for decoding with the presented principles and the computations are mapped to four modified ACSUs. The mapping

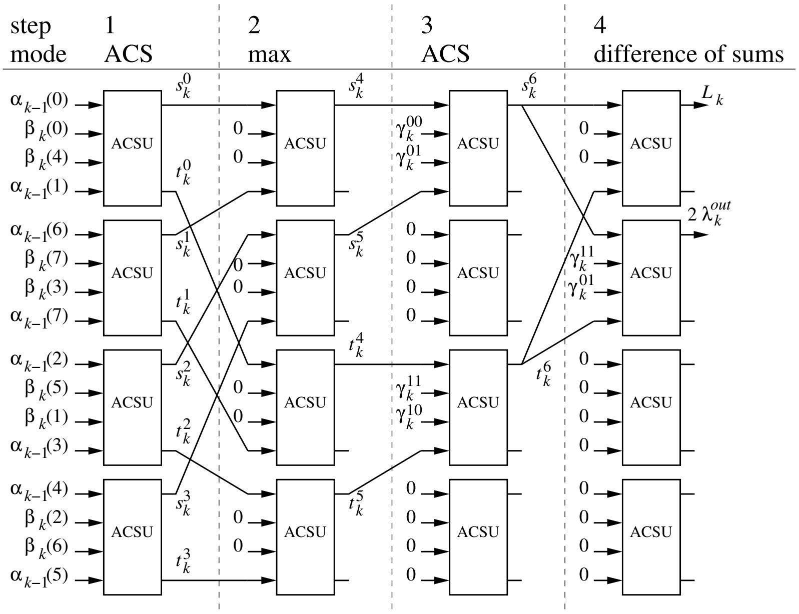


Fig. 20. Soft output, L_k, and extrinsic information, λ_k^{out}, can be computed within four steps with the proposed ACSUs.

of forward and backward metrics computations in (22)–(25) is trivial. Four radix-2 ACS operations are required for the computations in both directions and they can be executed in parallel.

In (28), all the $(s_k^i,\, t_k^i)$, $i = 0 \ldots 3$, pairs can be computed with radix-2 ACS operations. This practice is indicated in Fig. 20, in step 1. In (29), the maximums s_k^4, s_k^5, t_k^4, and t_k^5 are computed with ACSUs as shown in Fig. 20, step 2. Single maximums are computed and, therefore, the inputs b_0 and b_1 of the ACSUs are set to zero. In a similar way, s_k^6 and t_k^6 are computed with ACSUs as shown in Fig. 20, step 3. In this case, two radix-1 ACS operations are computed, since the computations do not share the same operands as required for radix-2 ACS. The final subtraction in (29) takes place in the last step in Fig. 20.

In (32), a difference whose subtrahend and minuend are sums of two operands is computed. Therefore, it can be computed with the proposed ACSU as shown in Fig. 20, in step 4. The final scaling by $\frac{1}{2}$ is trivial by hardwiring the output bits.

Branch metrics can be computed in two steps. First, intermediate values, s_k^7 and t_k^7, which require only the addition and subtraction capabilities of the ACSUs, are

Table 4. *Internal utilization of the computing resources of four ACSUs in max-log-MAP decoding example.*

Computation process	Step	# of steps	Utilization (%)
Forward metrics $\alpha_k(0 \ldots 7)$	1	1	100
Backward metrics $\beta_k(0 \ldots 7)$	1	1	100
Soft output L_k and λ_k^{out}	1		100
	2		17
	3		25
	4	4	17
Branch metrics $\gamma_k^{00 \ldots 11}$	1		8
	2	2	17
Total	–	8	48
Total with sliding window	–	9	54

computed. During the second step (33)–(36) are computed. With four ACSUs, they can be computed in parallel.

The internal utilization of the four ACSUs is shown in Table 4. As the internal utilization is computed, the number of required additions, subtractions, or comparisons is compared to the total resources inside the four ACSUs. With sliding window scheduling there is an extra backward metrics process and the total utilization is higher. With the additional backward process of sliding window algorithm one trellis stage can be processed in a total of 9 steps as shown in Table 4. If a decoder structure with more dedicated units were designed, the Table 4 shows that the computation of branch metrics $\gamma_k^{00 \ldots 11}$ is the most inefficient taking two steps and having modest utilization. Therefore, if more heterogeneous resources than a set of homogeneous ACSUs were applied it would be efficient to first map the computation of branch metrics to a dedicated unit. If the branch metrics were computed in parallel with other decoding tasks, processing one trellis stage would take seven clock cycles. In other words, the speedup would be $9/7 = 1.29$.

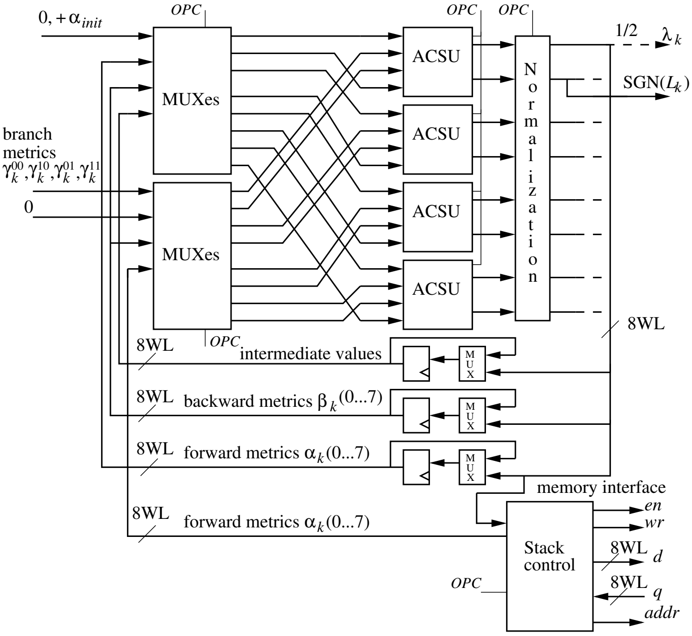


Fig. 21. *Multiplexed ACSUs and interface to the forward metric stack. Division by two is denoted with 1/2 and word length with WL.*

4.5 Special Function Units for an ASP Implementation of Max-Log-MAP Turbo Decoder

In an ASP implementation, the kernel computations are mapped to SFUs. In addition to the plain computations in Sections 4.2–4.4, also memory interfaces, storing of temporary values, feedback circuitry, and normalization must be considered in practical implementation.

4.5.1 Multiplexed ACSUs and Forward Metric Stack SFU

The unit computes the forward, $\alpha_k(0\dots7)$, and backward, $\beta_k(0\dots7)$, path metrics, extrinsic information, λ_k^{out}, and the hard output, $\mathrm{sgn}(L_k)$, with multiplexed ACSUs as described in Section 4.4. The structure of the SFU is shown in Fig. 21. Basically, the unit consists of four ACSUs, multiplexers which select the input operands for the ACSUs, sets of registers for storing the state of the process, and a stack for the forward path metrics.

The operation codes of the SFU are tabulated in Table 5. Several operation codes in Table 5 do not have external operands nor results, since the internal state of the unit is used instead. Ten operation codes are required since seamless context switches between operation modes are established with initialization and finishing steps of respective operation modes. Except for the stack memory of forward path metrics, all the other intermediate data is stored in internal registers. The same registers act as the mandatory delay component in the feedback loop structure of the multiplexed ACSUs. The registers can be divided into the registers holding the values of forward path metrics, backward path metrics, and intermediate results when computing the extrinsic information and the soft bit estimates.

The forward path metrics, $\alpha_k(0\ldots7)$, are pushed into a stack, since they must be read in a reverse order during the extrinsic information computation. The stack control unit in Fig. 21 maintains the stack pointer and interfaces an external memory. The stack size equals to the window length of the sliding window algorithm. The window length can be varied easily, since the stack resides in the external memory. The word length of the forward path metrics is 11 bits. Applying longer word length does not give significant improvement on error correction performance and even shorter word lengths can be used for lowering the hardware costs [129]. So, the word length of the stack is 8×11 bits $= 88$ bits. The window length, i.e., the maximum depth of the stack, is 32 stages. With only the conventional LSUs and dual-port memory, pushing eight path metrics would take four clock cycles meanwhile all the other memory accesses were blocked. Thus, using a dedicated stack memory instead of conventional LSUs has a significant impact on the performance of forward path metric computation.

The computation according to (7) and (8) shows the recursive dependency of path metrics. Due to the recursiveness, the latency of pipelined execution of (7) or (8) would limit the throughput. As single cycle computation results in a long critical path, it is also beneficial to feed back the intermediate results internally as shown in Fig. 21 instead of using the interconnection network of the processor. Naturally, the SFU includes also input and output registers to limit the critical path to the internal computations of the unit.

Table 5. *Operation codes of the multiplexed ACSUs SFU.*

Operation code	Description	Operands	Results
AFB	Feedback $\alpha_k(0\ldots7)$	$\gamma_k^{00\ldots11}$	–
BCSI	Initialize $\beta_k(0\ldots7)$	$\gamma_k^{00\ldots11}$	–
BFB	Feedback $\beta_k(0\ldots7)$	$\gamma_k^{00\ldots11}$	–
LLR1	Compute λ_k^{out} step 1	–	–
LLR2	Compute λ_k^{out} step 2	–	–
LLR3	Compute λ_k^{out} step 3	$\gamma_k^{00\ldots11}$	–
LLR4	Compute λ_k^{out} step 4	$\gamma_k^{00\ldots11}$	λ_k^{out}, $\mathrm{sgn}(L_k)$
BZERO	Init. the last $\beta_k(0\ldots7)$	–	–
AINIT1	Init. $\alpha_k(0\ldots7)$, step 1	–	–
AINIT2	Init. $\alpha_k(0\ldots7)$, step 2	$\gamma_k^{00\ldots11}$	–

Path Metric Normalization

The path metric normalization limits the word length of path metrics, which results in smaller area, shorter critical path for arithmetic units, and smaller storage space. The recursive update in (7) and (8) shows that without any limitations the path metrics would increase continuously.

The applied simple normalization requires non-negative path metrics. Therefore, non-negative initialization is used. It sets $\alpha_0(0)$ to a large positive value and $\alpha_0(1,\ldots,7)$ to the zero. This has the same effect as the more common initialization by setting $\alpha_0(0)$ to the zero and $\alpha_0(1,\ldots,7)$ to large negative values. After non-negative initialization, the recursive update in (7) and (8) maintains the path metrics non-negative. This can be verified with the trellis in Fig. 18, which shows that any state can be reached only with state transitions corresponding to pairwise complement systematic and parity bit pairs (x^s, x^p). Branch metrics with such complemented indices are negations, i.e.,

$$\gamma_k^{00} = -\gamma_k^{11} \tag{43}$$

$$\gamma_k^{01} = -\gamma_k^{10} , \tag{44}$$

as (18)-(21) show. Thus, the update of, e.g., $\alpha_k(0)$, can be written as

$$\alpha_k(0) = \max(\alpha_{k-1}(0) + \gamma_k^{00}, \alpha_{k-1}(1) - \gamma_k^{00}) . \tag{45}$$

With non-negative initialization, $\alpha_{k-1}(0)$ and $\alpha_{k-1}(1)$ are non-negative. If γ_k^{00} is

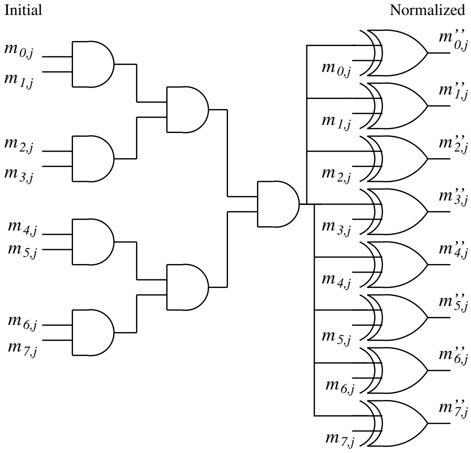


Fig. 22. *Applied normalization method. The jth bit of the ith path metric is denoted with $m_{i,j}$. The same structure is repeated for those jth path metrics bits, whose normalization is targeted.*

positive, then $\alpha_{k-1}(1) - \gamma_k^{00}$ can be negative. In this case $\alpha_{k-1}(0) + \gamma_k^{00}$ must be positive. Similarly, with negative γ_k^{00}, the term $\alpha_{k-1}(1) - \gamma_k^{00}$ must be positive. Similar derivation can be presented for all $\alpha_k(1), \ldots, \alpha_k(7)$ and $\beta_k(0), \ldots, \beta_k(7)$, respectively.

The most obvious normalization method searches for the minimum of path metrics and subtract it from all the path metrics. Lower complexity variations have been introduced, e.g., in [7, 127, 129]. The presented normalization methods compare the path metrics against a threshold value and when any of the path metrics exceeds the thresholds, a predefined value is subtracted from all the path metrics.

In this Thesis, the applied normalization method is based on fixed-point binary representation of path metrics. Subtracting 2^j from a binary number, whose jth bit is one, is simple by setting the jth bit to zero. The applied method detects when the same bit is one in all the path metrics of the current stage and it can be set to zero. The normalization circuitry of jth bit is shown in Fig. 22. Path metrics are denoted with m_i and the jth bit of m_i is denoted with $m_{i,j}$, since the same circuitry applies both for the forward, $\alpha_k(i)$, and backward, $\beta_k(i)$, metrics. The same circuitry is repeated for all the bit indices j, which will be normalized. Thus, there is no path going from the least significant bit (LSB) to the MSB and the extra length of the critical path is independent of the word length. If time-multiplexed structure of multipurpose ACSUs is targeted, the normalization can be enabled with one additional AND gate instead of multiplexers.

The operation of the normalization can be clarified by expressing the path metrics, m_i, as a sum of constant and variable terms,

$$m_i = c + \Delta_i \tag{46}$$

where $\min \Delta_i = 0$ and the maximum word length of Δ_i is defined as

$$W_\Delta = \lfloor \log_2(\max_i \Delta_i) \rfloor + 1. \tag{47}$$

The W_Δ is bounded, since the difference between maximum and minimum path metrics,

$$\max \Delta_i = \max m_i - \min m_i, \tag{48}$$

is bounded for all the trellis stages [48]. If the constant term, c, of path metrics has zero bit at jth position, i.e., $c_j = 0$, then binary representation of path metrics m_i is

$$m_i = (c_{MSB} \dots c_{j+1} 0 c_{j-1} \dots c_{LSB})_2 + \Delta_i. \tag{49}$$

Now, if $j > W_\Delta$ and the addition in (49) is carried out, then all the path metrics m_i share the same leftmost bits down to the $(j+1)$th bit, i.e.,

$$m_{0,MSB} = \ m_{1,MSB} \ = \dots = m_{7,MSB} \tag{50}$$

$$m_{0,MSB-1} = m_{1,MSB-1} = \dots = m_{7,MSB-1} \tag{51}$$

$$\dots$$

$$m_{0,j+1} = \ m_{1,j+1} \ = \dots = m_{7,j+1}. \tag{52}$$

The leftmost bits remain unaltered, since the c_j prevents carry bit propagation. Thus, the method can always normalize the bits from the MSB down to the $(j+1)$th bit for all the path metrics m_i, when $c_j = 0$ and $j > W_\Delta$.

4.5.2 Branch Metric Computation SFU

The branch metric computation unit can be used for further acceleration with heterogeneous resources instead of using only homogeneous ACSUs. The unit computes the branch metrics, $\gamma_k^{00\dots11}$, and interfaces the memories for soft systematic, y_k^s, and

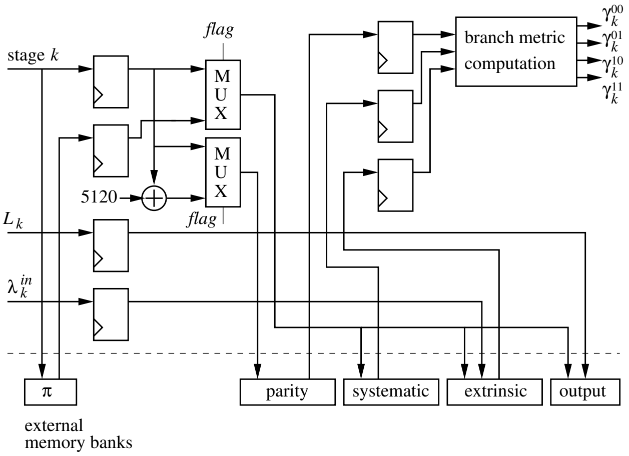


Fig. 23. *Branch metric computation and memory interfaces SFU. The interleaver memory is denoted with π.*

parity bits, y_k^p, extrinsic information, λ_k^{out},λ_k^{in}, and hard output, $\mathrm{sgn}(L_k)$. Even if the computation of branch metrics is possible with ACSUs as shown in Subsection 4.3, a dedicated unit can be used for obtaining higher utilization as indicated by Table 4, for avoiding branch metric buffer memory, and for grouping branch metric related memory interfaces into the same unit.

The structure of the SFU is shown in Fig. 23. The branch metrics are computed straightforwardly according to (33)–(36). The main advantage of the SFU is the integration of five different memory interfaces and branch metric computation. The integration hides memory accesses, interleaving, and their latencies which simplifies programming and requires fewer buses in the interconnection network of the processor. When computing branch metrics, $\gamma_k^{00\ldots11}$, the operands are trellis stage index k and a *flag* bit, which indicates whether the interleaved addressing mode is used. The interleaving sequence is read from the memory but, in principle, it could be replaced with a hardware unit capable of generating one interleaved address in one clock cycle.

As a second operation, the SFU is used to write the new extrinsic information, λ_k^{out}, and hard bit estimates, $\mathrm{sgn}(L_k)$, to the memory. Again, the stage index, k, and *flag* bit are given as operands in addition to the actual data. The extrinsic information and hard bit estimates are stored into separate memory banks. In the last iteration, the extrinsic information could be overwritten by the hard bit estimates. However, such an approach would prevent variable number of iterations. In that case, stopping

Table 6. *Sizes of external memory banks with 5120 length code block. The length of the sliding window is denoted with L_{win}.*

Memory	Addr. width	Data width	Size (bits)
Systematic bits, $\boldsymbol{y}^s$	13	7	35840
Parity bits, $\boldsymbol{y}^p$	14	7	71680
Extrinsic inf., $\boldsymbol{\lambda}^{out}$, $\boldsymbol{\lambda}^{in}$	13	10	51200
Interleaver	13	13	66560
Hard output, $\mathrm{sgn}(\boldsymbol{L})$	13	1	5120
Frwd. metrics, $\alpha_{k\ldots k-L_{win}+1}(0\ldots 7)$	5	88	2816

criterion like cyclic redundancy check [18] would determine the number of iterations and it is not known in advance, when the last iteration has been started. In principle, if the operation code encoding of the SFU were extended, it could operate also as a conventional LSU.

Table 6 summarizes the sizes of external memory banks. The required data widths depend on the initial scaling of the input data and the code block length. In this Thesis, the maximum code block size is 5120, which requires 13-bit address and data buses for e.g, interleaver memory. Since both parity bits are not accessed on the same half iteration, they can be stored in the same memory bank. So, the parity bit memory is double sized when compared to the systematic bits. Eight path metrics are read from or written to the stack memory and, therefore, it has wide data width.

4.5.3 Applying the SFUs in Turbo Decoder TTA Processor

The turbo decoder is implemented on a customizable ASP by applying the principles presented in previous sections. In this experiment, TTA [32] has been used as the architecture template. The TTA was chosen mainly for up-to-date tool support [54]. However, a somewhat similar implementation could be possible with any highly customizable processor with sufficient parallelism.

The block diagram of the proposed turbo decoder TTA processor is presented in Fig. 24. The addition/subtraction unit and comparison unit are the only conventional FUs. Since the processor is customized only for turbo decoding and the main computations are carried out with SFUs, no other FUs are required. Jumps, subroutine calls,

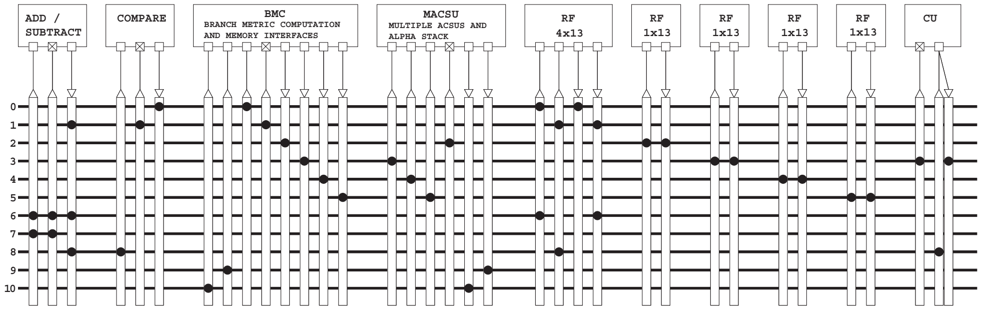


Fig. 24. *TTA turbo decoder processor. Connections between FUs and buses are denoted with filled circles. The external memory banks are interfaced with branch metric computation,* `BMC`*, and multiplexed ACSUs,* `MACSU`*, SFUs. The control unit is denoted with* `CU`*.*

and return operations access the program counter via control unit, CU, in Fig. 24. The processor contains one general purpose RF of four registers. In addition, there are four dedicated RFs of one register. They are used for temporarily storing branch metrics when the old extrinsic information is overwritten.

Turbo Decoder Program

A high-level pseudo code of the turbo decoder program is shown in Fig. 25. In practice, the program is written in parallel assembly and it follows the sliding window schedule with one process running at a time. Since the computation of the first and last windows are special cases, they are separated as shown in Fig. 25. In addition to the backward path metrics, $\beta_k(0\ldots7)$, also computations of extrinsic information, λ_k^{out}, and soft output, L_k, are included in `backward_process` procedure in Fig. 25, since they are always computed for the same trellis stage as backward metrics. The `backward_cold_start_process` procedure in Fig. 25 is the process of sliding window algorithm, which initializes the backward metrics beginning from an unknown initial state.

Complexity and Performance

The processor was synthesized on 130 nm standard cell technology and 1.35 V voltage. The area of the processor in terms of logic GEs is given in Table 7. Of the

```
procedure main begin
   call max_log_MAP || $\pi$:=False   # The first
   call max_log_MAP || $\pi$:=True    # iteration
   ...
   call max_log_MAP || $\pi$:=False   # The last
   call max_log_MAP || $\pi$:=True    # iteration
end
procedure max_log_MAP begin
   callfirst_forward_process
   loop (Block_length / Window_length)-1
      call backward_cold_start_process
      call backward_process
      call forward_process
   end
   call last_backward_process
end
```

Fig. 25. *High-level turbo decoder program flow. Parallelism is denoted with || and π presents the flag indicating interleaved access.*

four clock frequency alternatives in Table 7 the throughput of the 180 and 210 MHz processors is sufficient for UMTS.

The actual max-log-MAP algorithm is not altered, so the error correction performance is the same as with typical max-log-MAP turbo decoders with the same window and block lengths. Therefore, only the throughput is analyzed in detail. Decoding 5120 length code block takes 76408 clock cycles, so the throughput of one iteration with clock frequency, f_c, is

$$R = 5120\,\text{bits}/(76408/f_c)\ . \tag{53}$$

The number of clock cycles per trellis stage, C_{stage}, describes how efficiently the resources are used. It is inversely proportional to the throughput but independent of the clock frequency. The proposed processor achieves

$$C_{stage} = (76408/2)/5120 = 7.46\ . \tag{54}$$

Also a relative efficiency can be defined as the ratio between the theoretical minimum of required clock cycles and the used clock cycles for analyzing the proposed processor further. The proposed method in Section 4.4 requires seven clock cycles per trellis stage for computing forward and backward path metrics, extrinsic information, and hard bit estimates when the additional branch metric computation SFU is used. The last 32 stages long window takes six cycles per stage, as there is no additional backward path metric computation process of the sliding window algorithm. In one full iteration, all the trellis stages are processed twice. Thus, the relative efficiency of

Table 7. *Area as GEs and throughput of the turbo decoder TTA processor.*

Clock freq. (MHz)	Area (kGE)	Throughput 1 iter. (Mbps)	Throughput 6 iters. (Mbps)
100	12.722	6.70	1.12
150	14.942	10.05	1.68
180	17.886	12.06	2.01
210	20.750	14.08	2.35

the proposed turbo decoder processor would be

$$E_{decoding} = \frac{2 \times ((5120 - 32) \times 7 + 32 \times 6)}{76408} = 0.94\ . \tag{55}$$

The efficiency of any processor can be degraded by, e.g., loop prologues and epilogues, jump latency, or poor software pipelining of loops. The achieved relative efficiency indicates that such unavoidable overhead has very minor effect on the developed processor. The relative efficiency, $E_{decoding}$, equals also to the utilization of the multiplexed ACSUs SFU, which indicates that the processor is used almost as efficiently as it is theoretically possible with the given resources.

To avoid redundancy, the proposed turbo decoder processor is compared with other turbo decoders in Section 5.4, Table 10 in the next Chapter where a more parallel turbo decoder processor implementation is presented. Naturally, when the results in Table 10 are interpreted, it has to be taken into account that different implementations can be targeted to different telecommunications systems applying different data rates.

5. SPECIAL FUNCTION UNITS FOR STAGE-PARALLEL TURBO DECODING

In this Chapter, SFUs for the kernel computations of a stage-parallel turbo decoder applying max-log-MAP algorithm are proposed. The eight state trellis used in Chapter 4 is used again as an example target for the units. The SFUs are applied in practice as highly-parallel ASP applying them is implemented. On the contrary to previous partial-stage turbo decoder in Chapter 4, far higher parallelism is applied and higher throughput is targeted. As a consequence, also higher memory throughput is required. Expensive dual-port memory is avoided with a novel memory interface of the extrinsic information memory. The SFUs are connected directly to the memory interfaces of the processor to enable fast memory access. The proposed ASP achieves 22.7 Mbps throughput for the eight-state code, $[1\ \frac{1+D+D^3}{1+D^2+D^3}]$, with six iterations at 277 MHz clock frequency with 130 nm technology.

5.1 Previous Work

Turbo decoders are implemented on high-performance DSPs in [45, 57, 81]. However, their throughput is not sufficient even for current 3G systems if interfered channel conditions require several turbo iterations. Obviously, common DSPs are mainly targeted for other algorithms like digital filtering, but not for turbo decoding. The complexity of typical computations of turbo decoding is high and in the lack of appropriate computing resources the throughput is modest.

Higher throughput can be obtained if the processor is designed especially for turbo decoding, i.e., it has dedicated computing units for typical tasks of decoding algorithms. Such an approach is applied in [68, 124] where SIMD processor turbo decoders are presented. In [68], three pipelines and a specific shuffle network is applied. In [124], the pipeline has specific stages for turbo decoding tasks. With this approach the computing resources are more tightly dedicated to specific tasks of the decoding algorithm.

Even higher throughput can be obtained with pure hardware designs like [22, 137]. However, the programmability and flexibility is lost. Naturally, the more parallelism is used the higher throughput can be obtained. For example, by applying radix-4 algorithms the decoders in [22, 30] can process more than one trellis stage in one clock cycle. A slightly more flexible solution is to use monolithic accelerator, which is accompanied with a fully programmable processor like in [4, 96]. However, a monolithic solution can be uneconomical if the memory banks are not shared. Turbo coding requires long code blocks, so the memories can take significant chip area.

When compared to DSPs in [45, 57, 81], the proposed SFUs and ASP are mainly optimized for turbo decoding. There are no typical signal processing resources like multipliers. The resources of the proposed processor can be used in a pipelined fashion but there is no similar pipeline as in SIMD processor in [124]. In addition, more computing resources are used in the proposed processor as the targeted throughput is one trellis stage per clock cycle. Instead of using a specific shuffle network as a separate operation [68], the permutations are integrated in the internal feedback circuits of the path metric computations in the proposed processor. On the contrary to [22,137], the proposed processor is programmable. When compared to [4,96], the application-specific computing resources are accessed via datapath in the proposed processor. Thus, the resources can be controlled in detail with software. A similar processor template was applied in the previous Chapter, but far higher parallelism and throughput are targeted in this Chapter.

5.2 Special Function Units for Kernel Computations

The kernel computations of max-log-MAP algorithm include computation of forward metrics defined in (7), backward metrics (8), soft output (9), extrinsic information (10), and branch metrics (18)–(21). Three SFUs are proposed for these computations. The structure and operation of the units are discussed in the following.

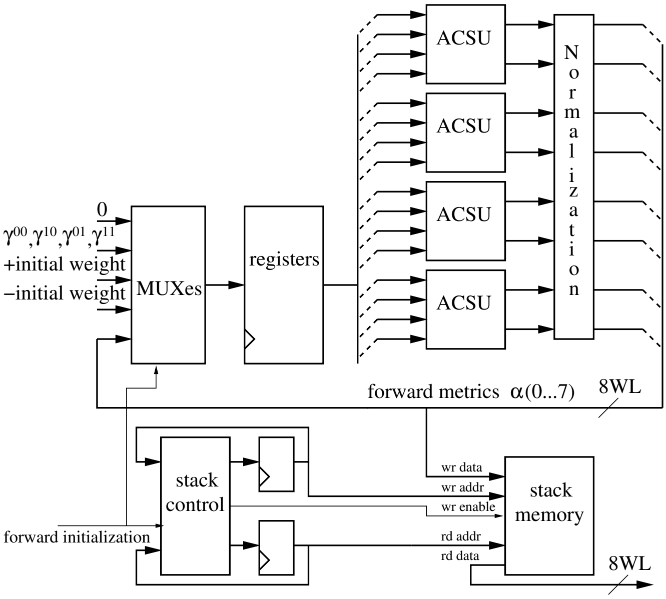


Fig. 26. *Forward path metric SFU with four ACSUs. The unit interfaces an external stack memory. Word length is denoted with WL.*

5.2.1 Forward Computation SFU

The forward computation SFU generates forward path metrics, $\alpha_k(i)$, normalizes them and continuously reverse orders one window of forward path metrics. The path metric computation in (7) requires ACS operations. All the path metrics for one trellis stage are computed in parallel, so the number of ACSUs depends on the size of the trellis. With the eight state example trellis in Fig. 18, four ACSUs are required. The structure of the unit with four ACSUs is shown in Fig. 26. Since the ACSUs followed by normalization reside in the critical path of the processor, the path metrics are fed back internally, instead of using internal buses of the processor where the unit will be instantiated.

Reverse ordering a window length block of path metrics is required since the extrinsic information, λ_k^{out}, and hard output, $\mathrm{sgn}(L_k)$ are computed together with the backward path metrics. The path metrics are reverse ordered with a stack which resides in external memory as in Section 4.5.1. The SFU updates read and write pointers of the stack memory. Instead of having two stacks, the same memory area can be used for continuous reverse ordering. The new samples are stored to the memory locations which were previously loaded. The direction of the stack is interchanged after a window length of push and pop operations. In other words, the pointers are first incremented, then they are decremented and so on. With this practice, the stack memory area remains full all the time after the first window length push operations. Since the push and pop operations access always consecutive memory locations, the parallel access can be implemented trivially by two memory banks.

5.2.2 Backward Computation SFU

The backward computation SFU is divided into several stages as indicated by Fig. 27. The first stage computes the backward path metrics of the acquisition mode and the second stage computes valid path metrics, $\beta_k(0 \ldots 7)$. The first two stages are structured similarly as the forward path metric computation SFU, i.e., both stages contain four ACSUs.

The next stages in Fig. 27 are responsible for computing the extrinsic information, λ_k^{out}, and hard output, $\mathrm{sgn}(L_k)$. Since there is no feedback loop, the computations can be pipelined freely to several stages. The structure takes an advantage of mapping the computations of (9) to radix-2 ACS operations, maximum operations, and radix-1 ACS operations as described in Section 4.3. The computation of λ_k^{out} in the last stage in Fig. 27 uses (30) to obtain $y_k^s + \lambda_k^{in}$ required by (10). With this practice the required term, $y_k^s + \lambda_k^{in}$, can be obtained without additional memory access. Otherwise, the memory of systematic bits, y_k^s, would need parallel access or there should be a long delay line preserving the values of y_k^s to provide y_k^s for (10). The backward path metric computation SFU includes a lot of arithmetic operations but, on the other hand, the design is straightforward since there is a one-to-one mapping between the computations and arithmetic units. Control signals are required for initialization and passing forward the path metrics from the acquisition mode process.

5.2.3 Branch Metric Computation SFU

On the contrary to the previous branch metric computation SFU in Section 4.5.2, the proposed SFU in Fig. 28 is targeted for higher throughput and, therefore, it includes a buffer of branch metrics. The SFU computes the branch metrics and interfaces the external memories for soft systematic, y_k^s, and parity bits, y_k^p, extrinsic information,

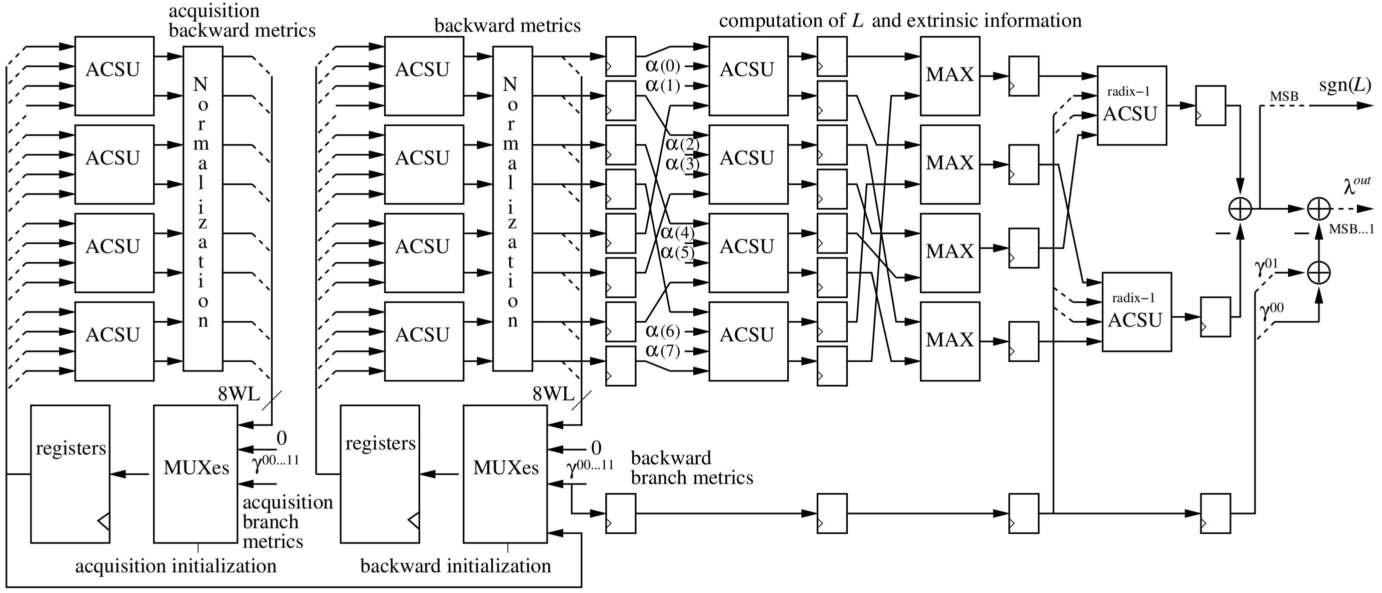


Fig. 27. Backward computation SFU contains several stages for acquisition of path metrics, computation of valid backward path metrics, and computation of extrinsic information, λ_k^{out}*, and hard output,* $\mathrm{sgn}(L_k)$*. Word length of path metrics is denoted with WL.*

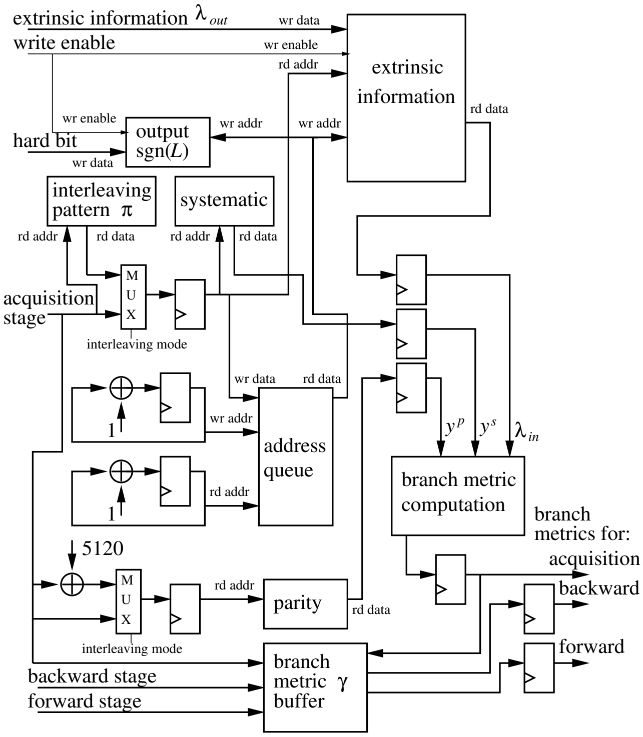


Fig. 28. Branch metric computation unit interfaces external output, extrinsic information, systematic bit, parity bit, interleaver, address queue, and branch metric buffer memories.

λ_k^{in}, λ_k^{out}, hard output, $\mathrm{sgn}(L_k)$, interleaving sequence, address queue, and branch metric buffer. Generated branch metrics are buffered in memory banks as proposed in Section 6.4. The SFU groups all the branch metric related memory interfaces into the same unit. The SFU hides memory accesses and, therefore, fewer internal buses are required in the interconnection network of the ASP. Since the branch metrics with complemented indices are negations of each other, only the branch metrics γ_k^{00} and γ_k^{01} are buffered for a four window length block. The branch metrics generated for the acquisition mode backward process are passed forward and stored in the buffer. The branch metrics for the forward and backward processes are read from the buffer.

The second operation of the SFU is writing the new extrinsic information, λ_k^{out}, and hard bit estimates, $\mathrm{sgn}(L_k)$, to the memory. The parallel memory accesses are implemented with the structure proposed in Section 6.5. In the proposed processor the interleaving sequence is read from a dedicated memory. If a hardware interleaver

Table 8. *External memory banks. Turbo code block length is 5120.*

Memory	Address width	Data width	Size (bits)
Systematic bits, $\boldsymbol{y}^s$	13	7	$5120 \times 7 = 35840$
Parity bits, $\boldsymbol{y}^p$	14	7	$10240 \times 7 = 71680$
Extrinsic inf., $\boldsymbol{\lambda}^{out}$, $\boldsymbol{\lambda}^{in}$	11	10	$4 \times 1280 \times 10 = 51200$
Interleaver,	13	13	$5120 \times 13 = 66560$
Hard output, $\mathrm{sgn}(\boldsymbol{L})$	13	1	$5120 \times 1 = 5120$
Address queue	7	13	$73 \times 13 = 949$
Branch metric buffer, $\gamma^{00}_{k \ldots k+4L_{win}-1}, \gamma^{01}_{k \ldots k+4L_{win}-1}$	5	20	$4 \times 32 \times 20 = 2560$
Fwrd. metric stack, $\alpha_{k \ldots k+L_{win}-1}(0 \ldots 7)$,	5	88	$32 \times 88 = 2816$

were used, generating two interleaved addresses in one clock cycle would increase the complexity of the interleaver. To enable an option for using hardware interleaver instead of memory, the addresses are buffered in a queue data structure, which is denoted as address queue in Fig. 28. As the two access sequences of the address queue are sequential with a constant offset between them, the memory can be divided into two banks according to the LSB trivially. In principle, with the window length, L_{win}, and an implementation dependent inherent delay, c_{rw}, the address queue buffer delays the addresses $2L_{win} + c_{rw} = 73$ clock cycles, which is the difference between read and write indices with the access sequence $B(i)$ defined in (59). The memories interfaced by the branch metric and forward computation SFUs are summarized in Table 8.

5.3 Turbo Decoder Processor

The proposed turbo decoder ASP applies the SFUs presented in previous Section. In the proposed implementation, the TTA [32] has been used as the architecture template. However, similar principles can be applied on any customizable processor, which possesses sufficient parallelism.

The principal block diagram of the developed TTA processor is shown in Fig. 29. Since the processor is targeted only to turbo decoding, it has only two conventional

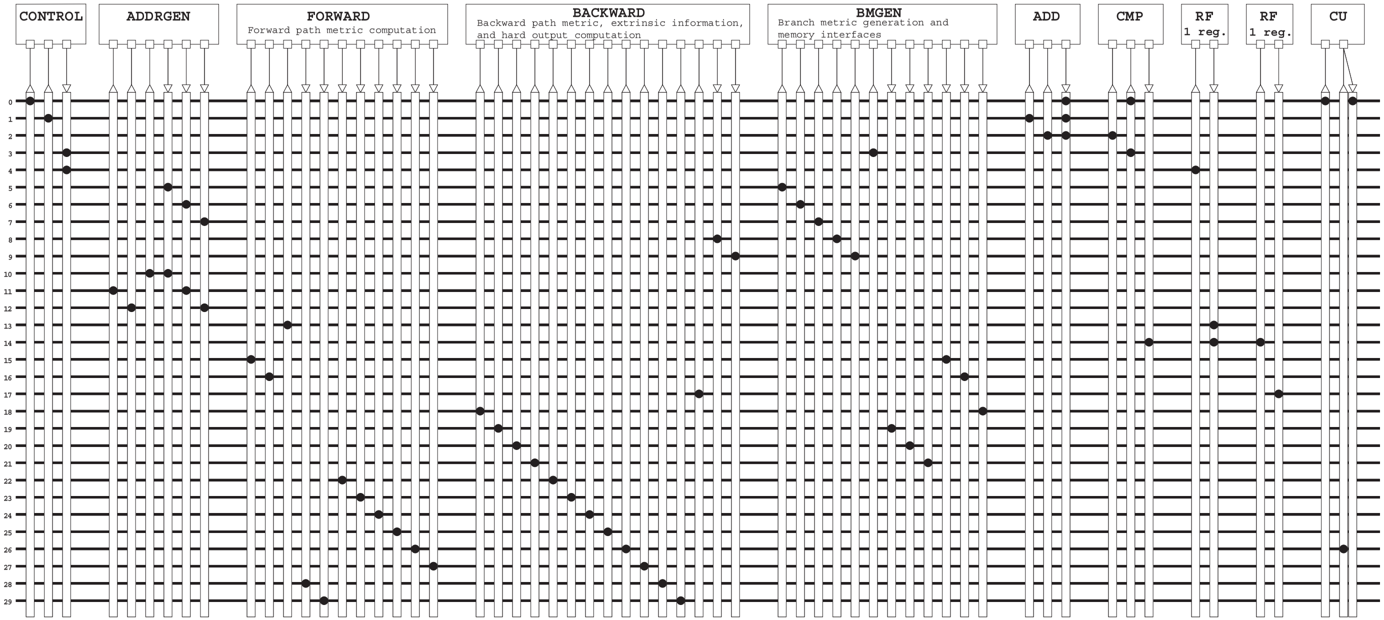

CONTROL
ADDRGEN
FORWARD
Forward path metric computation
BACKWARD
Backward path metric, extrinsic information,
and hard output computation
BMGEN
Branch metric generation and
memory interfaces
ADD
CMP
RF
1 reg.
RF
1 reg.
CU
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

Fig. 29. *TTA turbo decoder processor. Filled circles denote connections between resources and buses.* CONTROL*: generation of control word for SFUs,* ADDRGEN*: address generation,* FORWARD *forward path metric computation and stack memory interface,* BACKWARD*: acquisition, backward path metric, extrinsic information, and hard output computations,* BMGEN*: branch metric buffering and memory interfaces,* ADD*: addition unit,* CMP*: comparison unit,* RF*: register file,* CU*: program flow control unit.*

FUs, namely addition and comparison units. The control unit, CU, in Fig. 29 is used for `jump`, `call`, and `return`, i.e., the new value of program counter is written to the CU and the return address is read from the CU. Due to the frequent bypassing of data, the processor contains only two general purpose registers in two RFs. In the turbo decoding program, the registers are used in parallel to delay continuously generated values, which are needed also on the next clock cycle.

The SFUs presented in previous Section carry out the core computations of max-log-MAP algorithm. However, implementation of a decoder requires also assisting functions which control the core computations and provide the correct data at correct time to the respective SFU. The assisting SFUs for address generation and control of computations are presented in the following.

5.3.1 Address Generation SFU

The address generation SFU generates addresses for accessing branch metrics, γ_k^{00} and γ_k^{01}. As it is shown by the schedule in Fig. 6(a), there are three parallel processes and all of them require branch metrics. The branch metrics are generated and buffered in the branch metric computation SFU. Thus, the generated addresses are addresses of the buffer and they are not affected by the interleaving mode.

The access sequence of the addresses of the forward path metric, $\alpha_k(0\ldots7)$, computation is sequential but the backward processes require sawtooth access patterns as shown in Fig. 6(a) and Fig. 34. The previous addresses are operands of the SFU and they are fed back via the interconnection network. The internal operation of the SFU is depicted in Fig. 30. The window length parameter, L_{win}, in Fig. 30 determines the period of the sawtooth pattern.

5.3.2 Control SFU

The purpose of the controlling SFU is to generate a control word, which is used as an argument of other SFUs. Even if the highest level control takes place in software level, the lowest level control can be implemented more conveniently in hardware. With this practice unnecessary details are hidden from the application program. The word is used to control multiplexers, initialization of state registers, and signals in the memory interfaces.

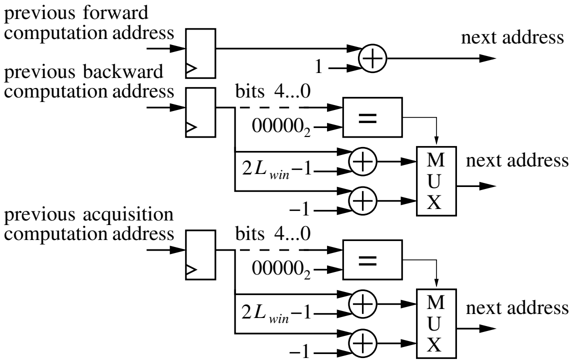

previous forward computation address
previous backward computation address
previous acquisition computation address
1
next address
bits 4...0
00000₂
=
2L_win−1
−1
M U X
next address
bits 4...0
00000₂
=
2L_win−1
−1
M U X
next address

Fig. 30. Address generation SFU generates one sequential and two sawtooth pattern address sequences. Length of the sliding window is denoted with L_{win}.

Generating the control word requires evaluating several conditionals in parallel as depicted in Fig. 31. The parameter, $2L_{win} + c_{rw}$ in Fig. 31 is the constant distance between read and write operations of the extrinsic information, λ_k^{in}, λ_k^{out}, memory. The operands of the SFU are current trellis stage, k, and interleaving mode. Even if the control could be distributed among the SFUs, the verification and any future changes, if required, are alleviated since the control signals are packed to the single control word generated with an independent unit.

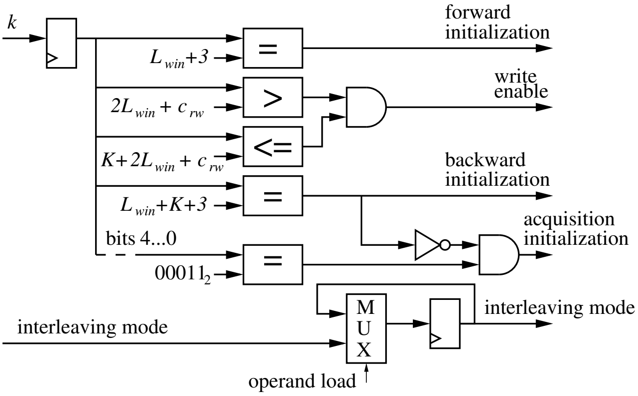


Fig. 31. Control SFU evaluates several conditions in parallel and generates control word for the other SFUs. Length of sliding window is denoted with L_{win}, distance between extrinsic information read and write operations with $2L_{win} + c_{rw}$, and code block length with K. Input k is the current trellis stage.

```
procedure turbo begin
   # First iteration
   call max-log-MAP || interleaving := false
   call max-log-MAP || interleaving := true
   ...
   # Last iteration
   call max-log-MAP || interleaving := false
   call max-log-MAP || interleaving := true
end

procedure max-log-MAP begin
   call initialization_of_SFUs
   loop $(K + 2 \times L_{win})$ begin
      call run_SFUs
   end
   call finish_computations
end
```

Fig. 32. *High-level program flow of the turbo decoder. Parallelism is denoted with | |. Code block length and window length are denoted with K and L_{win}, respectively*

5.3.3 Turbo Decoder Program

The turbo decoder program is programmed in parallel assembly and it follows the sliding window schedule in Fig. 6(a). The highest-level pseudo code is shown in Fig. 32. The subprograms of the `max-log-MAP` procedure are inlined to avoid jump latency. The first procedure, `initialization_of_SFUs`, feeds the initial constants to the control and address generation SFUs. The loop kernel repeats instruction words consisting of computation and loop control parts. The computation part of the instruction word feeds the control word to all the SFUs, addresses to branch metric computation SFU, branch metrics to forward and backward computation SFUs, and hard bit estimates and extrinsic information to the branch metric computation SFU. The loop control part includes addition, comparison, and conditional jump operations. In total, the instruction word consists of 30 parallel data transports.

The number of iterations of the main loop in Fig. 32 exceeds the block length K. Additional clock cycles are taken by the first window length L_{win} trellis stages as the branch metrics buffer is filled with the values of first window. The last window requires also additional L_{win} clock cycles, as the results can not be computed before the forward path metrics for the last window are ready. Due to the latencies of the SFUs, valid results are not generated immediately. Therefore, the total number of activations of SFUs exceeds the number of iterations in the loop. The last stages are not processed in the loop to match the total number of required activations.

Table 9. *Complexity and throughput of the turbo decoder TTA processor.*

Clock frequency	Area	Throughput 1 iteration	Throughput 6 iterations
100 MHz	27.9 kGEs	49 Mbps	8.2 Mbps
200 MHz	31.9 kGEs	98 Mbps	16.4 Mbps
250 MHz	35.7 kGEs	123 Mbps	20.5 Mbps
277 MHz	43.2 kGEs	136 Mbps	22.7 Mbps

5.3.4 Performance and Complexity

The throughput is determined by the number of clock cycles per code block. The developed TTA turbo decoder processor takes 10404 clock cycles with the length-5120 code block. So, the throughput of one iteration is

$$R = 5120\,\text{bits}/(10404\frac{1}{f_c}) \tag{56}$$

where f_c is the clock frequency. The processor was synthesized on 130 nm standard cell technology with 1.35 V voltage. The area in terms of logic GEs of the generated netlist and the corresponding throughput are given in Table 9. The design is area-efficient as the computing resources are used efficiently, there are no large multiplexers, and the high-level structure of the processor is very simple as shown in Fig. 29. The dynamic power consumption of the decoder is 50.8 mW with 250 MHz. So, decoding a length-5120 code block would take 0.0127 mJ energy.

In addition to absolute valued throughput in Table 9, the relative efficiency of the developed processor and decoder program can be analyzed. The number of clock cycles per trellis stage, C_{stage}, of max-log-MAP computation, i.e., half iteration is

$$C_{stage} = (10404/2)/5120 = 1.016\,. \tag{57}$$

The efficiency can be described as a measure how close to the theoretical cycle count the achieved number of clock cycles approaches. With the applied resources, the theoretical cycle count equals to the block length. Thus the efficiency can be defined as

$$E_{decoding} = 1/C_{stage} = 0.984\,. \tag{58}$$

Since the efficiency in (58) can be interpreted similarly as the efficiency in (55), it shows that any unavoidable overhead has only minor part in the total cycle count and the utilization of the SFUs is very high.

5.4 Comparison of Turbo Decoder Implementations

A comparison with other turbo decoder implementations is summarized in Table 10. The implementations are categorized into three classes. Pure hardware designs are not programmable. Monolithic accelerators are implementations where processor is accompanied by a dedicated hardware decoder. The third category contains processors, in which the computing resources are accessed via a datapath.

Naturally, turbo decoders applying more accurate algorithms like log-MAP instead of max-log-MAP require more area and the longer critical path lowers clock frequency. In log-MAP algorithms, an approximation, $\ln(e^a + e^b) = \max(a, b) + f(a, b)$, is used and comparisons are difficult since the accuracy of the correction term, $f(a, b)$, may vary. Typically the recursive update in (7) and (8) dominates the critical path and prevents high clock frequencies. However, the path metric computation can be accelerated also by expressing the recursion in such a way that the control signals of selection operations are computed in parallel with additions like in [66, 125].

The complexity is tabulated if it is given as logic GEs excluding the memories in the respective reference. Due to the differing underlying cell structures, comparing different FPGA architectures would be difficult and the size of the memories depends on the targeted block size and technology. For example, [117] takes 250 kGEs with memories but the computing units of the core decoder take only 24 kGEs. Even if the memories are excluded in Table 10, it is still possible that some implementations may use register based delay lines for queue data structures and the registers are naturally included in the gate count. Such a register based approach is simple to design as it does not require address generation nor memory bank selection logic. As a drawback, transferring electric charge through all the registers in a delay line consumes a lot of energy.

The throughput metrics are normalized to one iteration to alleviate comparisons. The throughput is directly proportional to the clock frequency, which results in a low throughput for some FPGA based implementations. Therefore, also the last column should be observed, as it gives the number of clock cycles per trellis stage, C_{stage}. It is calculated from the throughput and the clock frequency, unless it is given in respective reference. For [80], an achievable 300 MHz clock frequency has been assumed to calculate the throughput.

Table 10. *Comparison of turbo decoder implementations. The proposed ASPs are denoted with Chapter 4 and Chapter 5.*

Category	Ref.	Architecture / technology	Clock frequency	Area	Throughput 1 iteration	Algorithm	Cyc. / stg. (C_{stage})
Pure HW design	[22]	180 nm technology	145 MHz	410 kGEs	144 Mbps	log-MAP	0.50
	[137]	Virtex 5 FPGA	310 MHz	–	139 Mbps	MAX SCALE	1.12
	[19]	130 nm technology	246 MHz	44.1 kGEs	112 Mbps	max-log-MAP	1.10
	[30]	Virtex 2 FPGA	56 MHz	–	79.2 Mbps	MAP	0.35
	[25]	180 nm technology	100 MHz	115 kGEs	27.1 Mbps	max-log-MAP	1.84
	[23]	180 nm technology	111 MHz	85.0 kGEs	25 Mbps	log-MAP	2.21
	[117]	180 nm technology	133 MHz	24.0 kGEs	22.8 Mbps	log-MAP	2.92
Monolithic accelerator	[4], [134]	Turbo co-processor of C64x	300 MHz	86.6 kGEs	90.4 Mbps	log-MAP	1.66
	[96]	SISO dec. with SIMD	135 MHz	34.4 kGEs	32.9 Mbps	[max]-log-MAP	2.05
Programm. processor	Chapter 5	TTA proc. (130 nm)	277 MHz	43.2 kGEs	136 Mbps	max-log-MAP	1.02
	[124]	SIMD ASP (65 nm)	400 MHz	64.1 kGEs	100 Mbps	log-MAP	2.00
	Chapter 4	TTA proc. (130 nm)	210 MHz	20.8 kGEs	14.1 Mbps	max-log-MAP	7.46
	[68]	SIMD DSP	400 MHz	–	10.4 Mbps	max-log-MAP	19.2
	[57]	TigerSHARC DSP	250 MHz	–	9.6 Mbps	max-log-MAP	13.0
	[52, 53]	VLIW ASP (FPGA)	80 MHz	–	5.0 Mbps	max-log-MAP	8.00
	[45]	SP-5 SuperSIMD DSP	250 MHz	–	4.7 Mbps	max-log-MAP	26.9
	[81]	C62x VLIW DSP	300 MHz	–	4.4 Mbps	max-log-MAP	34.5
	[74]	ST120 VLIW DSP	200 MHz	–	2.7 Mbps	max-log-MAP	37.0
	[80]	C55x DSP	300 MHz	–	2.0 Mbps	max-log-MAP	74.8
	[119]	PC with Pentium III	933 MHz	–	366 kbps	max-log-MAP	1275
	[89]	XiRisc reconf. proc. (FPGA)	100 MHz	–	270 kbps	log-MAP	185
	[74]	DSP56603 DSP	80 MHz	–	243 kbps	max-log-MAP	165

The implementations [22] and [30] in Table 10 have $C_{stage} < 1$, as they apply the radix-4 algorithm. The architecture in [30] includes two component decoders. The decoders in [23] and [25] support also Viterbi decoding. The area of [25] includes a path metric memory of the Viterbi decoder and an embedded interleaver. The interleaver is included also in the area of [19]. The implementation in [117] is targeted for high-speed turbo architecture consisting of several parallel decoders. The performance and complexity are reported for one decoder in Table 10. Naturally, non-programmable decoders tend to have a lot of dedicated computing resources for the functions of the decoding algorithm and they have high throughput when compared to majority of programmable processors.

For [4, 134] the complexity in Table 10 includes only the turbo co-processor, but not the accompanying C64x VLIW DSP. The accompanying processor is included in the complexity of [96] since the proportion of only the decoder part was not available. The interleaving pattern is computed with the processor and the decoder supports both max-log-MAP and log-MAP algorithms in [96]. In both implementations, the decoder is not tightly connected to the datapath of the processor, so it is not flexibly controllable. Instead, the decoder must process independently which resembles pure hardware decoders. As a second drawback of monolithic accelerators, some memory is dedicated only for the turbo decoder component.

The ASP in [124] supports also Viterbi decoding. The ASP has an 11 stage pipeline. There are dedicated pipeline stages for address generation, branch metric generation, state metric computation, and four stages for computing the soft output. The processor in [68] has three pipelines and trellis butterflies are alleviated with a specific shuffle network. The decoding algorithm of the processors in [52, 53] is selectable. In the table, performance of max-log-MAP is given as it achieves higher clock frequency.

The processor presented in Chapter 4 applies more sequential schedule, which is presented in Fig. 6(b), as it contains less computing resources than the proposed processor in this Chapter. However, it achieves sufficient performance for the UMTS data rate 2 Mbps. Most of the implementations with higher throughput in Table 10 exceed the UMTS data rate drastically. If their architectures are designed only for decoding, they cannot be used for other tasks and they have to idle most of the time if they were used in UMTS receivers. Naturally, such an approach would not be economical. Finally, the Table 10 shows that conventional commercial DSPs have modest throughput and C_{stage}. This is understandable, since their architectures are optimized mainly for high throughput multiply and accumulate operations but not for turbo decoding.

The proposed processor has the highest throughput of all the programmable turbo decoder processors. The performance is comparable with pure hardware implementations and the number of clock cycles per trellis stage, C_{stage}, is best of all the implementations, which do not apply the radix-4 algorithm. For example, even if the clock frequency is lower when compared to [137], the proposed processor has only slightly worse performance, since it has better C_{stage}. The low C_{stage} shows that the programmability and flexibility of the processor does not degrade the efficiency. The utilization of the computing resources is even higher than with the pure hardware decoder.

5.4.1 Scaling and Processing without SFUs

In Chapter 4, it was shown that in principle all the computations of max-log-MAP algorithm can be mapped to slightly modified ACSUs. Therefore, one way to scale the decoder design in order to target it to different throughput requirements would be to vary the number of ACSUs. With this practice there would be only one major design parameter which is varied and the other resources were tailored according to the requirements of ACSUs. For comparison, with several design parameters, the number of their different combinations may increase too rapidly.

Processing solely without SFUs and without any tailoring of the processor can be exemplified by assuming that there were only two LSUs available, i.e., dual-port memory would be used. The memory bandwidth limitation gives a coarse estimate on the maximum performance of such a TTA processor. The branch metric generation requires accessing extrinsic information, systematic and parity bits for each trellis stage. On every second half iteration, interleaved accesses are used which requires accessing the interleaving sequence. Thus, the total number of load operations is 3.5 where the fraction 0.5 originates from the interleaving. Naturally, storing branch metrics requires two store operations when only one of the respectively complement branch metrics is stored. The forward metric computation requires two loads for obtaining the branch metrics and eight stores for saving the forward metrics. The acquisition mode of backward metric requires only loading branch metrics.

The computation of backward path metric, extrinsic information, and soft output requires again two loads for the branch metrics and eight loads for the forward path metrics. Storing the extrinsic information or soft output requires one store operation. On every second half iteration, the store operations use interleaved access sequence which requires an additional load of interleaving sequence. Thus, roughly 3.5 + 2 + 2 + 10.5 load operations and 2 + 8 + 1 store operations, in total 29 operations, are required per trellis stage. Even if only 15 dual memory accesses, i.e., 15 clock cycles per trellis stage, would be sufficient in theory, developing a program which achieves full utilization of LSUs can be challenging due to the data dependencies between the computation loops.

6. PARALLEL MEMORY ACCESS IN TURBO DECODERS

In this Chapter the parallel memory accesses of the turbo decoders are addressed. Parallel access methods are proposed for the extrinsic information memory and for the branch metric memory.

6.1 Principles of Extrinsic Information Memory Accesses

In general, parallel access can be implemented with separate read and write memories, dual-port memory, a memory running with higher clock frequency, or some type of memory banking structure. A general structure consisting of N_{banks} parallel memory banks and T parallel access interfaces is presented in Fig. 33. The structure abstracts the N_{banks} memory banks so that the T interfaces can use them as a continuous single array of words. The purpose of the address generation and bank selection functions is to map parallel accessed data to separate banks. The derivation of conflict-free bank selection and address generation functions is typically the main design problem of parallel memory bank structures.

Computing the branch metrics according to (18)–(21) requires reading the extrinsic information memory. If the decoder processes one trellis stage in one clock cycle, the new extrinsic information must be written to the extrinsic information memory at the same time. Thus, at least dual access is required. The sliding window schedule in Fig. 6(a) shows that, in fact, there are three parallel processes requiring branch metrics. Instead of quadruple access of the extrinsic information and triple access of systematic and parity bit memories, the locality of branch metric accesses can be utilized as shown in Section 6.4.

For clarity, when accessing the extrinsic information memory the term index is used instead of the term trellis stage, since both component decoders can access the same data at different trellis stages. The index can be interpreted as an array index pointing

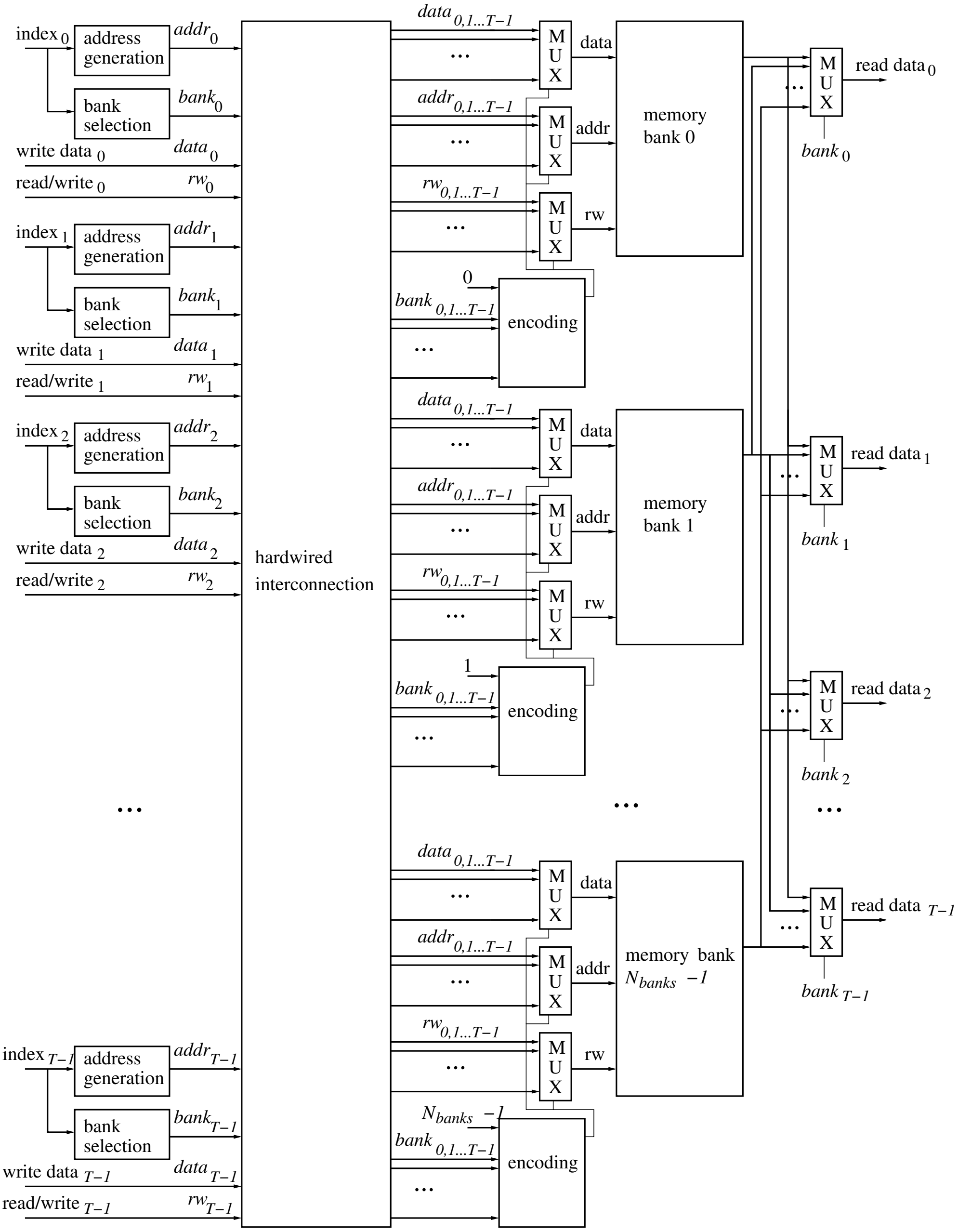


Fig. 33. *General parallel memory bank structure with T parallel access interfaces and N_{banks} memory banks. The encoding unit selects $address$, $data$, and rw signals of that interface which has selected the respective bank.*

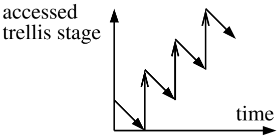


Fig. 34. Access sequence of the extrinsic information is based on the sawtooth pattern.

to elements of the extrinsic information vector, $\boldsymbol{\lambda}^{in}$, $\boldsymbol{\lambda}^{out}$. The half iterations require different access sequences. In this Thesis, the term linear access sequence is used for the locally linear access of the first half iteration. The term interleaved access sequence is used for the second half iteration. With sliding window scheduling both access sequences are based on the sawtooth sequence shown in Fig. 34. With a window length L_{win} and a code block length K the sawtooth access sequence can be defined as,

$$B(i) = L_{win}\lfloor i/L_{win}\rfloor + L_{win} - 1 - (i \bmod L_{win}) \tag{59}$$

where $i \in \{0, 1, \ldots, K - 1 + 2L_{win} + c_{rw}\}$ and constant term c_{rw} originates from from the delay of computations in the decoder. The accessed indexes of the linear access sequence are generated with $B(i)$ by updating the i with $i+1$. The interleaved access sequence is generated in a similar way with $\pi(B(i), K)$ where $\pi(\cdot, \cdot)$ is the interleaving function. The write operations of the extrinsic information follow the read operations with the same but delayed access sequence. The delay equals to $2L_{win} + c_{rw}$. The generation of the read and write indexes of both access sequences is clarified with the structure shown in Fig. 35.

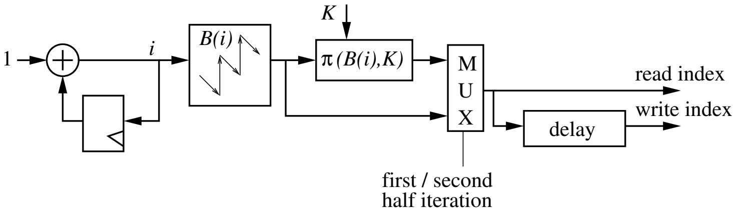


Fig. 35. Generation of read and write indexes of linear access sequence and interleaved access sequence.

6.1.1 3GPP Turbo Coding Interleaver

Since the parallel memory access sequences depend on the interleaving function, $\pi(j, K)$, the 3GPP turbo code internal interleaver [2] is used as an example and presented in the following. Basically, the 3GPP interleaver is defined with the aid of a matrix. The number of rows, R, and columns, C, of the matrix depend on the code block length, K, which can vary in the range $40 \leq K \leq 5114$. The derivation of the matrix dimensions, $R \times C$, is defined in [2]. In the following the dimension functions are denoted with $R = M_{rows}(K)$ and $C = M_{cols}(K)$. The matrix can contain more elements than the code block, i.e., there are unused elements in the matrix.

First, the matrix is filled with elements, e_j, $j \in \{0, \ldots, K-1\}$ row-wise beginning on the first row and the first column. Each matrix element, e_j, corresponds to a unique row, $row(j, K)$, and column, $col(j, K)$, i.e.,

$$row(j, K) = \lfloor j/M_{cols}(K) \rfloor \ ; \ col(j, K) = j \bmod M_{cols}(K). \tag{60}$$

Thus, the $row(j, K)$ and $col(j, K)$ denote the original row and column of the e_j before any subsequent permutations. Next, intra-row permutations, denoted with $P_{intra}(col(j), row(j), K)$ in this Thesis, are applied as defined in [2]. The intra-row permutation function gives the new column of the respective element. The intra-row permutations depend on the row and they are based on the exponentiation function in a finite field. As the base of the exponentiation is primitive root of the field, the powers of the primitive root give all the elements of the field except zero [73]. The first two steps of the interleaving are illustrated in Fig. 36(a) and (b).

In the third step, inter-row permutations, $P_{inter}(row(j, K), K)$, are applied. The $R \in \{5, 10, 20\}$ and there are only four different inter-row permutations [2]. The applied inter-row permutation is selected according to the code block length, K. In the last step, the matrix is read column-wise beginning on the first column and first row. If the matrix contains more than K elements, the additional elements are pruned, i.e., only the original elements e_j are output when reading the matrix. These two steps are shown in Fig. 36(c) and (d).

Thus, if the matrix is originally filled with elements $e_j = j$, $\forall j \in \{0, \ldots, K-1\}$, the jth element read from the matrix column-wise gives the value of interleaving function, $\pi(j, K)$, which denotes the interleaved index of the original index j. If the

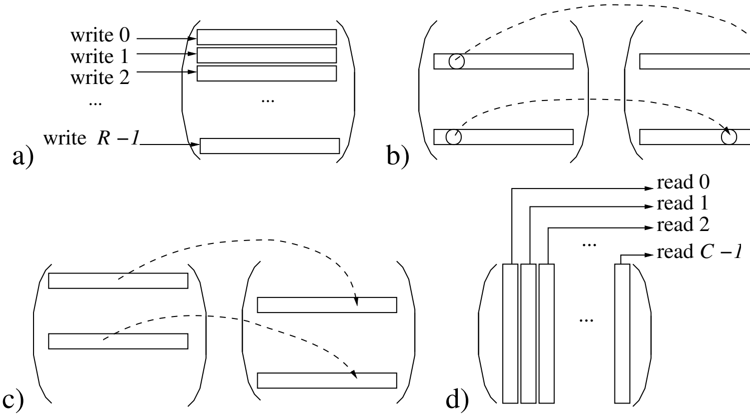


Fig. 36. *3GPP interleaving: a) the interleaving matrix is filled row-wise, b) intra-row permutations are applied, c) inter-row permutation is applied, and d) the matrix is read column-wise.*

bits were used as elements, i.e., $e_j = x_j^s$, then reading the matrix column-wise would result in the interleaved bit sequence.

6.2 Previous Work

In [108], a conflict free mapping is derived with an iterative annealing procedure. The native block length of the algorithm is a product of the number of parallel component decoders and the number of memory banks. Even if the reconfiguration is mandatory for varying interleaving patterns, no hardware implementation is presented for the annealing procedure. The turbo decoder implementation studies [4, 22, 30, 137] do not provide details of the applied access methods.

In [47], graph coloring is used to find mappings. It uses more memory banks than [108], but a hardware architecture for the reconfiguration is presented. The reconfiguration takes about $10K$ clock cycles for K length code block [47]. For comparison, one conflict would take one additional clock cycle. Therefore, it can be more advantageous to suffer all the conflicts instead of reconfiguration in some cases. In addition, the address computations in [47] require division and modulus, which are difficult to implement on hardware when the block length is not a power of two.

In [97], a conflict free access scheme for extrinsic information memory is developed. The bank mapping is based on the interleaving matrix and derived with the aid of exclusion sets. Since the parity of columns is used in bank selection, it is unclear how parallel accesses are mapped when there is odd number of columns. A different approach is applied in [20, 116, 117, 126, 128] where buffers are applied instead of deriving conflict free address generation and bank selection functions. In [20, 116, 117], high-speed decoding with several write accesses is assumed. For each writer there is one memory bank and for each bank there is a dedicated buffer. In [117], the buffered approach is developed further and the memories are organized in ring or chordal ring structures. The work is continued in [79] where a packet switched network-on-chip is used and several network topologies are presented. To reduce the sizes of queue buffers and to prevent overflows, the network flow control is applied. In [126], multiple parallel decoders decoding the same code block are targeted.

For 3GPP interleaving, a simple structure with four or six memory banks is proposed in this Thesis. Six memory banks are required for conflict free access and only 3.2% of the accesses conflict with four banks. Instead of focusing only on a predefined interleaving function, also a more general memory structure is proposed. The structure applies simple address generation and bank selection functions and buffering of conflicting accesses. Instead of solving all the conflicts with memory bank selection as in [47,97,108], the proposed approach is to use a very simple memory bank selection function and to maintain a constant throughput with buffering in spite of conflicting accesses.

6.3 Parallel Memory Structure for 3GPP Turbo Decoding

The general structure in Fig. 33 indicates that the address generation and bank selection are the key functions of parallel memory structures. In this Section, address generation and bank selection functions are designed for the access sequences of the 3GPP turbo decoding.

Naturally, if the code block length is less than half of the maximum, i.e., $K < 2557$, the memory banks can be always organized as separate read and write memories, which limits efficiently the number of different interleaved access sequences that must be considered. Since parallel read and write operations can be substituted with parallel read operations followed by parallel write operations, the analysis of parallel accesses can be focused on adjacent accesses, i.e., accessing extrinsic information in parallel with indexes $j, j+1$ or with indexes $\pi(j, K), \pi(j+1, K)$ where

$j \in \{0, 2, 4, \ldots K-2\}$. Naturally, parallel access with linear access sequence can be implemented with low-order interleaving. In that case the memory bank is selected of a total of N_{banks} banks with $j \bmod N_{banks}$ when accessing the jth element. However, applying such an approach straightforwardly with interleaved access sequence would require de-interleaving function, $\pi^{-1}(j, K)$ to calculate bank selection function $\pi^{-1}(j, K) \bmod N_{banks}$ which would map accesses of indexes $\pi(j, K), \pi(j+1, K)$ to separate banks. Since the 3GPP interleaving function is based on exponentiation functions in finite fields, the $\pi^{-1}(j, K)$ would require discrete logarithm. Computing discrete logarithm would be almost impractical as indicated by cryptosystems utilizing the fact, that power function can be computed relatively easily, whereas the discrete logarithm requires extensive computational effort [38].

6.3.1 Proposed Structure

Instead of deriving $\pi^{-1}(j, K)$, the proposed method organizes the data as an abstract data structure of row-wise filled $R \times C$ matrix and takes an advantage of the fact that parallel operations at indexes $\pi(j, K), \pi(j+1, K)$ always map to separate rows. The accesses do not necessarily map to the same column since the matrix is not permuted like the interleaving matrix in Section 6.1.1. Thus, the access conflicts with interleaved access sequence can be avoided by mapping data, which corresponds with parallel accessed rows, to separate memory banks.

The principal structure of the proposed bank selection and address generation functions is shown in Fig. 37. The generation of the sawtooth sequence, $B(i)$, is included in the diagram for clarity. The structure in Fig. 37 can be instantiated for address generation and bank selection in the general parallel memory structure in Fig. 33. Since the data is abstracted as a matrix, the row and column of the accessed element is required. With linear access sequence $row(j, K)$, $col(j, K)$ defined in (60) give the row and column of the jth element and with interleaved access sequence the permutation functions $P_{inter}(row(j, K), K)$ and $P_{intra}(col(j), row(j), K)$ give the required information. As they are intermediate functions of the computation of the interleaving function, it can be assumed that they are available without extra costs. The structure shows that the memory banks are selected both according to the low-order interleaving, i.e., the LSB, and according to the row. The selection with the LSB allows parallel accesses with linear access sequence.

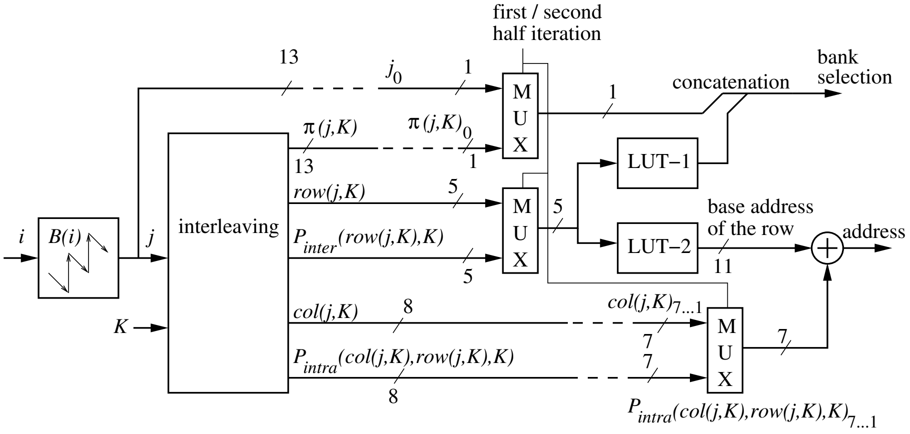


Fig. 37. *Proposed structure for interleaved and linear access sequences. The word length of the output of the LUT-1 depends on the number of banks.*

Since there are only four different inter-row permutation sequences and $R \leq 20$ [2] the contents of the LUTs are efficiently limited. The LUT-2 in Fig. 37 is used for the address generation and it provides the base address of each row. The proposed structure can be used with four or six memory banks. With six memory banks all the conflicts can be avoided. With four banks, 3.2% of the accesses conflict. The conflicts occur when $R \times C > K$ and the column-wise reading of the interleaving matrix in Section 6.1.1 has to skip the pruned entry and continue to the next row. The content of the LUT-1 depends on the inter-row permutation. For example, with permutation mapping row i to the new position according to the ith element of the set $\{19,9,14,4,0,2,5,7,12,18,16,13,17,15,3,1,6,11,8,10\}$ [2], it is sufficient to map row sets $\{3,5,6,8,14,16,17,19\}$, $\{2,4,7,10,11,13,15,18\}$, and $\{0,1,9,12\}$ to separate banks.

6.3.2 Area of Memory Configurations

Estimates of the required chip area for different memory configurations are presented in Fig. 38. The areas required by dual-port memory and single bank single-port memory are included as reference. The technology of the memories is Mitel Semiconductor SCA200 0.35 micron Embedded Arrays and the measures are obtained by Paracell Model Generator program. The word length is eight bits. The memory chip area estimates show that, as the memory is divided into banks, additional chip area is required. Thus, the memory should not be divided into too many banks. The single

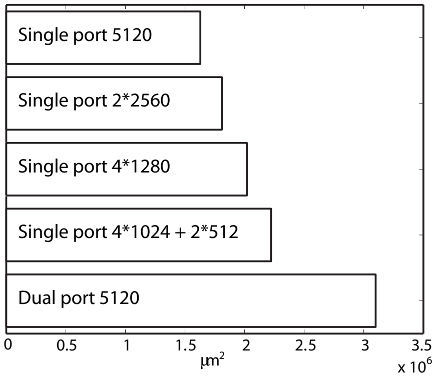


Fig. 38. *Required area for different memory configurations.*

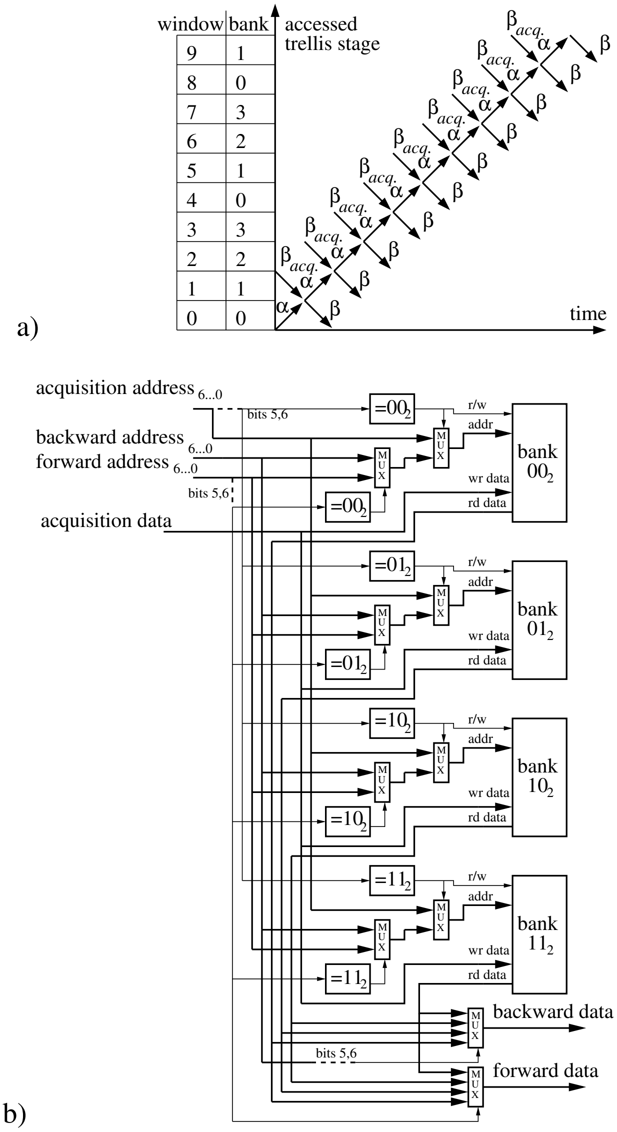
bank memory takes 73% of the area required by six memory banks, even if both of them have a total of 5120 words. However, the area for six single-port memory banks is 72% of the area required by the dual-port memory.

Typically, faster memories have higher costs in terms of area than slower memories, i.e., there is a trade-off between costs and access time. The access times of memories have not been used as design parameters in this Section nor in Section 6.4. However, if memory banking structure results in smaller total memory requirements than a straightforward solution, the obtained savings may give some freedom to replace memories with memory banks having faster access time, in theory.

6.4 Branch Metric Buffering

The sliding window schedule in Fig. 39(a) indicates that, in fact, three trellis stages are accessed in parallel. Instead of boosting the proposed parallel memory system for even higher throughput, buffering of branch metrics can be applied. There are certain advantages of such buffering. Firstly, the buffer requires only a small amount of memory when compared to the systematic bit, parity bit, and extrinsic information memories, whose sizes are determined by the maximum code block size. Secondly, single-port memory can be used for all the memories. Thirdly, the access sequence of the buffer is independent of the interleaving. The proposed method in this Section is applied in the stage-parallel turbo decoder processor in Chapter 5.

The accesses of the forward, backward, and initialization of backward path metric



window	bank
9	1
8	0
7	3
6	2
5	1
4	0
3	3
2	2
1	1
0	0

Fig. 39. Buffering of branch metrics: a) mapping the accessed windows to four memory banks, b) structure of memory interface with four banks.

computations can be mapped efficiently to separate windows. This practice is illustrated in Fig. 39(a) where accessed memory banks and windows are denoted. In theory, three memory banks are required. However, with four memory banks delays of memory banks or processes do not cause short-term conflicts when transition from previous window to the next one takes place. In addition, implementation complexity of division and modulo operations is avoided with simple hardwired logic.

The forward and backward path metric computation processes read branch metrics from the buffer. The data is written to the buffer by the backward path metric acquisition process. The bank mapping in Fig. 39(a) shows that the acquisition process of backwards metrics is always ahead of other processes. With window length L_{win}, the bank selection, B_{sel}, of the process accessing the kth trellis stage can be defined as high-order interleaving, i.e.,

$$B_{sel} = \lfloor k/L_{win} \rfloor \bmod N_{banks}. \tag{61}$$

The address of the accessed bank, B_{addr}, is defined as

$$B_{addr} = k \bmod L_{win}. \tag{62}$$

Naturally, division and modulo operations are avoided if the window length, L_{win}, and the number of banks, N_{banks}, are powers of two. With this practice, the B_{sel} is formed by the bits $\log_2 L_{win} \ldots \log_2 L_{win} + \log_2 N_{banks} - 1$ of the binary presentation of the k. In a similar way, the B_{addr} is given by the bits $0 \ldots \log_2 L_{win} - 1$ of the k. The structure of the memory interface applying the bit-wise bank selection and address generation with $L_{win} = 32$ and $N_{banks} = 4$ is shown in Fig. 39(b). Due to the hardwired logic, the overhead and delay of the interface is kept at minimum. Furthermore, using the buffer does not require any changes to the accessing processes.

6.5 Extrinsic Information Memory Access with Buffered Write Operations

The next proposed method is used in the stage-parallel turbo decoder processor in Chapter 5. Briefly, on the contrary to resolving the conflicts by analyzing $\pi(j, K)$, the method delays write accesses to prevent simultaneous read and write access in the same memory bank.

6.5.1 Parallel Memory Access Method

The proposed parallel access method combines simple memory bank mapping and buffering of conflicting accesses. With simple bank selection the number of conflicts

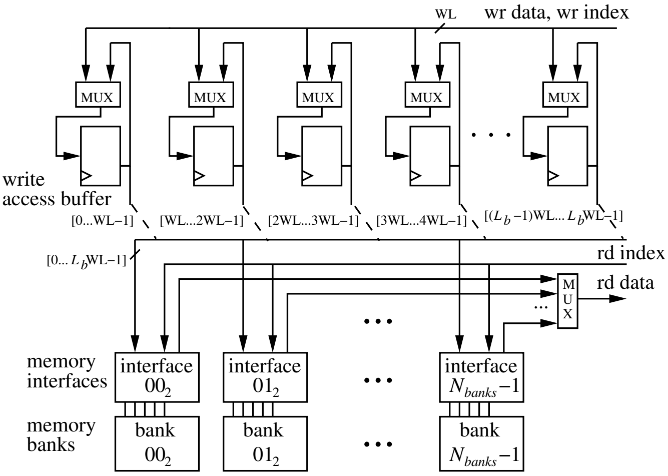


Fig. 40. *Proposed memory structure with buffered write operations. Word length of data and address pair is denoted with WL. Indices in brackets index the bus connected to the* L_b *length buffer.*

is decreased to a tolerable level and the performance penalty of memory access conflicts can be overcome with a short buffer. This practice of combining memory banks and a buffer is illustrated in Fig. 40.

The bank selection function is a simple modulo operation of the address and the number of banks, N_{banks}. Thus, when accessing the kth trellis stage the bank selection, B_{sel}, and address, B_{addr}, are generated according to

$$B_{sel} = k \bmod N_{banks} \ ; \ B_{addr} = \lfloor k/N_{banks} \rfloor \ . \tag{63}$$

So, if the number of banks is a power of two, the bank selection and address generation can be generated by low-order interleaving. In other words, bank selection is implemented by simply hardwiring the lowest bits to the selection signal and the highest bits form the address.

In Fig. 40, each memory bank is accompanied with an interface. The functionality of the memory interface is shown in Fig. 41. In principle, the memory interface gives the highest priority for memory read operations. The read operations must be always served to allow continuous decoding. On the contrary, write operations are inserted to the buffer, which consists of registers in Fig. 40. All the memory banks that do not serve the read operation are free to serve write operations waiting in the buffer.

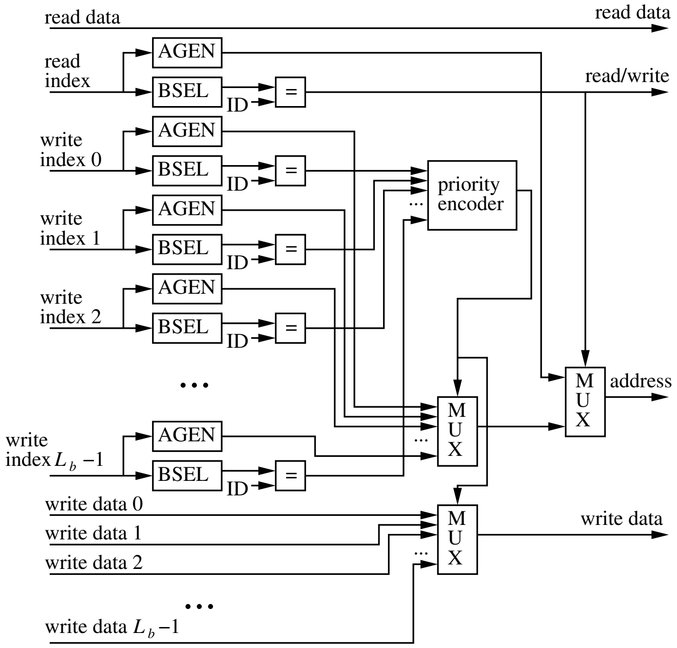


Fig. 41. *Memory bank interface. Length of the buffer is denoted with L_b and address generation and bank selection with AGEN and BSEL, respectively. The ID refers to the number of the interfaced bank. Write index and data signals are connected to the respective elements of the buffer.*

The proposed buffer must be able to be read and written in a random access manner and in parallel by all the memory bank interfaces. Thus, it must be implemented with registers. However, the length of the buffer for practical systems will be modest as will be shown later on.

Basically, the buffer balances memory accesses. Balancing is targeted also with a single shared buffer instead of dedicated buffers for each memory bank. If there were dedicated buffers for memory banks, their length should match the maximum requirements. However, the length of combined buffer is less than sum of dedicated buffers. This is natural, since only one buffer could be filled at a time if dedicated buffers were used.

The decoder produces memory accesses at a constant rate, two accesses per clock cycle, i.e., one read and one write operation. On the contrary, the memory system is capable of maximum throughput directly proportional to the number of banks. In other words, the ability of the proposed method to perform without performance degradation is based on the asymmetric throughput rates and throughput capability between the decoder side and memory bank side.

6.5.2 Operation with 3GPP Interleaving Pattern

For the 3GPP interleaving function [2], four memory banks are required. With $L_{win} = 32$, $c_{rw} = 9$, and $N_{banks} = 4$ a buffer of 16 data and address pairs is sufficient to avoid buffer overflows with all the 3GPP interleaving patterns with block length, $K = 2557, \ldots, 5114$. The buffer length requirement is verified with simulation. If the code block is shorter than 2557, memory banks can always be organized as dedicated read and write memories. In addition to the number of memory banks, the required overflow free buffer length depends also on the distance between read and write operations, $2L_{win} + c_{rw}$, but it is not directly proportional to the distance. In other words, the required buffer length can be shorter or longer with some other values of c_{rw} and the required minimum length must be searched with simulation.

In the end of a half iteration, there are no parallel read accesses but only write accesses for the last samples and the utilization of the buffer cannot increase. If the buffer is not emptied during this phase, extra clock cycles are spent to empty the buffer. The experimented cases with 3GPP interleaving pattern and $K = 2557, \ldots, 5114$ do not require such extra cycles, i.e., the buffer is empty when the decoder issues the last write operation. Since extra clock cycles are not required, there is no performance degradation due to the buffering of conflicting accesses.

The area costs in terms of logic GEs is only 3.3 kGEs for the buffer and 0.5 kGEs for one memory interface with f_c=100 MHz clock frequency and 130 nm technology. With four memory banks, four interfaces are required. The complexities of memory interface and buffer are relatively low, since they do not require complex arithmetic and the buffer length is short.

6.6 Discussion

The main benefit of the proposed structure with buffer is that it can be applied with different interleaving functions by adjusting the buffer length and the number of banks. The only requirement is that there is sufficient variation in the low-order bits to avoid too large buffer. The main drawback of the structure is that the buffer must be implemented with registers which increases the power consumption.

The structures with six or four memory banks in Section 6.3 require only input or output registers but they are targeted only for 3GPP interleaver. The structure with four memory banks in Section 6.3 results in few access conflicts. The conflicts originate from the pruned entries in the interleaving matrix. However, to provide new value of $\pi(j, K)$ on every clock cycle, the data path of the interleaving function generator must be doubled due to the pruned entries [23]. Thus, if the data path is not doubled and $\pi(j, K)$ generation stalls in the presence of pruned entries, there will be single access instead of dual access and conflicts could be avoided.

Methods in [47,108] solve conflicts with complex memory bank mapping and address generation mechanism. However, their complexity limits their practical applicability. Buffered accesses are presented in [20, 116, 117], but the ratio of memory banks to the number of parallel accesses differs and the methods are targeted for systems consisting of multiple parallel decoders. The proposed method relies on the asymmetric throughput rates of turbo decoder side and memory subsystem side. As a second difference there are dedicated buffers for each memory bank, which increases the total length of the buffers. The proposed structure applies single shared buffer which decreases the number of registers.

6.7 3G LTE Interleaving Sequence Generation

In this Section, it is shown that an interleaving sequence generator for 3G LTE turbo codes can be derived as a special case of polynomials modulo integer with linearly incremented variable. As the main result, a systematic method for deriving hardware structures for such computations is proposed. The method is derived by recursively applying principles of simplifying modulo operations in a limited domain. With the aid of the proposed method, efficient hardware structures can be derived for any polynomials and significant savings can be obtained.

6.7.1 Previous Work

Pseudo random number generators are typically defined with the aid of modulo operation. If the value of the modulus is a power of two, hardware implementation of the modulo operation is trivial by excluding the higher bits. On the contrary, if modulus is chosen more freely, a straightforward implementation of the modulo operation results in increased hardware complexity. Also software implementations slow down, since the modulo and division operations are excluded from the instruction sets of typical DSPs and software emulation [29] is required.

An implementation of 3G standard compliant interleaver is presented in [5]. Even if the interleaver is more complex than in 3G LTE, the modulo operation dominates the critical path. In [11], the division operation affects heavily on the area of the 3G interleaver. A software implementation of 3G interleaver is presented in [95] and the presented method avoids straightforward modulo operations. An implementation applying similar principles, i.e., avoiding straightforward modulo operation, as proposed in this Thesis is also presented in [104]. Several studies present implementations of elementary operations modulo integer. For example, the multiplication is considered in [9] and [39], square in [85], and addition/subtraction in [64]. Serial computation of whole polynomials is considered in [71]. In this Thesis, the targeted application has useful limitations, which are shown in Section 6.7.2, and due to the limitations, too general implementations in [9, 39, 64, 71, 85] would be inefficient.

In this Thesis, the targeted number generators are polynomials modulo integer, which are evaluated at consequent points with constant distances. On the contrary to software implementation of a first degree polynomial in [95], a systematically derived hardware is presented to arbitrary high degree polynomials. It is also shown that incrementing variable by one is a special case of the more general constant step incrementation. To achieve low complexity or area costs of hardware implementation, the proposed method does not apply multiplications nor general purpose modulo operations. Instead, only addition/subtraction units, multiplexers, registers, and basic logic gates are required. It is shown that the hardware structures can be derived systematically from a given polynomial. The number of elementary blocks in the derived structure is directly proportional to the degree of the polynomial. The results show that the reduction in the implementation complexity is significant when compared to a straightforward implementation. The interleaving function of the 3G LTE telecommunication systems is used as an example case of the targeted number generators.

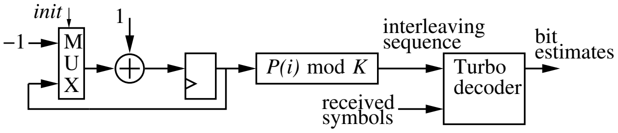


Fig. 42. *Targeted number generator applied for interleaving sequence generation.*

6.7.2 Problem Definition

The targeted function has the form

$$y(j) = W(j) \bmod K, \quad j \in \{0, c, 2c, 3c \dots (N-1)c\} \tag{64}$$

where $W(j)$ is an nth degree polynomial

$$W(j) = w_n j^n + w_{n-1} j^{n-1} + w_{n-2} j^{n-2} + \dots + w_1 j + w_0 \tag{65}$$

and K and c are some positive integers. The application of the polynomial of the targeted form as interleaving sequence generator is illustrated in Fig. 42.

Further derivation is simplified by considering a polynomial

$$P(i) = a_n i^n + a_{n-1} i^{n-1} + a_{n-2} i^{n-2} + \dots + a_1 i + a_0 \tag{66}$$

where $a_k = w_k c^k$, $k \in \{0, 1, 2, \dots n\}$. With this substitution $W(j) = P(i)$ with $i = j/c$. Thus, with the aid of substitution of the coefficients, further derivation can be limited to polynomials where the variable i is incremented with the step size of one. In other words, i gets values $0, 1, 2, \dots N-1$ in sequential order.

6.7.3 Modulo in Limited Domain

The computation of modulo operation

$$E(i) = ai \bmod K \text{ where } i \in \{0, 1, 2, \dots, N-1\} \tag{67}$$

can be transformed to a form

$$E(i) = (E(i-1) + a) \bmod K \text{ where } i \in \{1, 2, \dots, N-1\}\,. \tag{68}$$

The constant a can be substituted with $a' = a \bmod K$ to guarantee that $0 \leq a' < K$. With this substitution,

$$0 \leq E(i-1) + a' < 2K - 1 \quad \text{and} \tag{69}$$

$$E(i) = (E(i-1) + a') \bmod K \tag{70}$$

can be expressed as a selection between two alternatives,

$$E(i) = \begin{cases} E(i-1) + a' & \text{if } E(i-1) + a' < K \\ E(i-1) + a' - K & \text{otherwise.} \end{cases} \tag{71}$$

Computing (71) is far simpler than general modulo operation. This kind of practice has been applied in [95] for software implementation of 3G standard conforming interleaver, which requires only first degree polynomial.

In the following derivations, a limited domain modulo operation, $x \operatorname{mod}^* K$, is applied. It can be defined as

$$x \operatorname{mod}^* K = \begin{cases} x - K & \text{if } x \geq K \\ x + K & \text{if } x < 0 \\ x & \text{otherwise.} \end{cases} \tag{72}$$

The domain of $x \operatorname{mod}^* K$ covers the range $-K \leq x < 2K - 1$, which includes also a short range of negative values. In other words,

$$x \operatorname{mod}^* K = x \bmod K \text{ when } -K \leq x < 2K - 1 \,. \tag{73}$$

Such a limited domain $x \operatorname{mod}^* K$ operation can be implemented with a far simpler hardware than the general $x \bmod K$.

6.7.4 Modulo of High-Degree Polynomials

A more general case of computing a modulo of a polynomial is more demanding due to the higher order terms. For computing the modulo of a general polynomial, $P(i)$, i.e.,

$$P(i) \bmod K = \left(\sum_{k=0}^{n} a_k i^k \right) \bmod K \,, \tag{74}$$

the polynomial is presented as a sum of the value of polynomial with $i - 1$ and a difference polynomial $P_{(0)}(i)$. More formally,

$$P(i) \bmod K = (P(i-1) + P_{(0)}(i)) \bmod K \,, \tag{75}$$

which can be presented as

$$P(i) \bmod K = (P(i-1) \bmod K + P_{(0)}(i) \bmod K) \operatorname{mod}^* K \tag{76}$$

with $x \operatorname{mod}^* K$ operation. However, (75) nor (76) cannot be computed with straightforward application of (70) and (71), since the difference polynomial

$$P_{(0)}(i) \bmod K = (P(i) - P(i-1)) \bmod K \tag{77}$$

is not a constant, but a function of i. The differences, $P_{(k)}(i)$, are denoted with subscripts in parentheses as they resemble slightly the derivatives of $P(i)$ as will be shown later on.

In principle, a simplified modulo of a polynomial is derived by recursively applying the (80). The procedure is repeated for the differences as long as the degree of difference $\deg P_{(k)}(i) > 0$. The next two differences are

$$P_{(1)}(i) \bmod K = (P_{(0)}(i) - P_{(0)}(i-1)) \bmod K \tag{78}$$

$$P_{(2)}(i) \bmod K = (P_{(1)}(i) - P_{(1)}(i-1)) \bmod K \tag{79}$$

and in general

$$P_{(k-1)}(i) \bmod K = (P_{(k-2)}(i) - P_{(k-2)}(i-1)) \bmod K\ . \tag{80}$$

In the end,

$$P_{(n)}(i) = d \bmod K \tag{81}$$

and the last polynomial has a constant value for all i.

The procedure terminates, since

$$\deg P_{(k)}(i) < \deg P_{(k-1)}(i)\ . \tag{82}$$

This can be verified by considering the highest order term of $P(i) - P(i-1)$. The highest order terms of the $P(i)$ and $P(i-1)$ are $a_n i^n$ and $a_n(i-1)^n$, respectively. Again, the highest order term of $(i-1)^n$ is i^n, so the nth order term of the difference is $a_n i^n - a_n i^n = 0$. A similar proof can be applied for each $P_{(k)}(i)$, $k \in \{1, \ldots, n\}$. Since the degree decreases on every step of the procedure, the difference resembles slightly the derivative of a polynomial.

As an example the procedure is shown for a third degree polynomial $P(i)$ with constant coefficients and $K = 13$. First, (80) is applied as follows,

$$\begin{aligned} P(i) \bmod 13 &= (5i^3 + 4i^2 + 3i + 2) \bmod 13 && (83)\\ P_{(0)}(i) \bmod 13 &= (P(i) - P(i-1)) \bmod 13 && (84)\\ &= (15i^2 - 7i + 4) \bmod 13 && (85)\\ P_{(1)}(i) \bmod 13 &= (P_{(0)}(i) - P_{(0)}(i-1)) \bmod 13 && (86)\\ &= (30i - 22) \bmod 13 && (87)\\ P_{(2)}(i) \bmod 13 &= (P_{(1)}(i) - P_{(1)}(i-1)) \bmod 13 && (88)\\ &= 30 \bmod 13\,. && (89) \end{aligned}$$

It should be noted that the intermediate forms (83)–(89) are defined with the aid of general modulo $x \bmod K$, instead of $x \operatorname{mod}^* K$.

The initial values for $i = 0$ are,

$$\begin{aligned} P(0) \bmod 13 &= 2 \bmod 13 = 2 && (90)\\ P_{(0)}(0) \bmod 13 &= 4 \bmod 13 = 4 && (91)\\ P_{(1)}(0) \bmod 13 &= -22 \bmod 13 = 4 && (92)\\ P_{(2)}(0) \bmod 13 &= 30 \bmod 13 = 4 && (93) \end{aligned}$$

If the coefficients of the polynomial are constants, the modulos in (90)–(93) can be computed in advance and stored instead of polynomial coefficients. Otherwise, their modulos can be computed iteratively during an initialization phase, which is explained with a hardware example in Section 6.7.5.

Next, the values are derived for $i = 1$ by substitutions according to (76) except the first constant terms in (94) and (101),

$$\begin{aligned} P_{(2)}(1) \bmod 13 &= 4 && (94)\\ P_{(1)}(1) \bmod 13 &= (P_{(1)}(0) \bmod 13 + \\ &\qquad P_{(2)}(1) \bmod 13) \operatorname{mod}^* 13 && (95)\\ &= (4 + 4) \operatorname{mod}^* 13 = 8 && (96)\\ P_{(0)}(1) \bmod 13 &= (P_{(0)}(0) \bmod 13 + \\ &\qquad P_{(1)}(1) \bmod 13) \operatorname{mod}^* 13 && (97)\\ &= (4 + 8) \operatorname{mod}^* 13 = 12 && (98) \end{aligned}$$

$$\begin{aligned} P(1) \bmod 13 &= (P(0) \bmod 13 + \\ &\qquad P_{(0)}(1) \bmod 13) \operatorname{mod}^* 13 && (99) \\ &= (2 + 12) \operatorname{mod}^* 13 = 1 && (100) \end{aligned}$$

and for $i = 2$ with

$$\begin{aligned} P_{(2)}(2) \bmod 13 &= 4 && (101) \\ P_{(1)}(2) \bmod 13 &= (P_{(1)}(1) \bmod 13 + \\ &\qquad P_{(2)}(2) \bmod 13) \operatorname{mod}^* 13 && (102) \\ &= (8 + 4) \operatorname{mod}^* 13 = 12 && (103) \\ P_{(0)}(2) \bmod 13 &= (P_{(0)}(1) \bmod 13 + \\ &\qquad P_{(1)}(2) \bmod 13) \operatorname{mod}^* 13 && (104) \\ &= (12 + 12) \operatorname{mod}^* 13 = 11 && (105) \\ P(2) \bmod 13 &= (P(1) \bmod 13 + \\ &\qquad P_{(0)}(2) \bmod 13) \operatorname{mod}^* 13 && (106) \\ &= (1 + 11) \operatorname{mod}^* 13 = 12 && (107) \end{aligned}$$

and so on for $i = 3, 4, 5, \ldots$ It can be noticed that in (94)–(107) computations of only the simplified modulo operations, $x \operatorname{mod}^* K$, are required. The general modulos, $x \bmod K$, in (94)–(107) have been computed in previous steps.

6.7.5 Hardware Implementations

A simple hardware structure for computing the modulo of a polynomial according to the proposed principles can be derived systematically. In Fig. 43, such a structure is shown for a general case. The basic building block of the hardware structure in Fig. 43 is a computing element consisting of register, multiplexer, adder, and modulo operator in the limited domain, i.e., $x \operatorname{mod}^* K$. The computing element is shown in Fig. 44(a). According to the derived equations, the elements are repeated and their inputs and outputs are connected to pass forward $P_{(k)}(i)$ for computing the $P_{(k-1)}(i)$ with the next element as shown in Fig. 43.

Implementation of the proposed $x \operatorname{mod}^* K$ operation has very low complexity. A unit capable of computing $x \operatorname{mod}^* K$ operation of (72) is shown in Fig. 44(b). The unit consists of an AND gate, one addition or subtraction unit, and 2-input multiplexer.

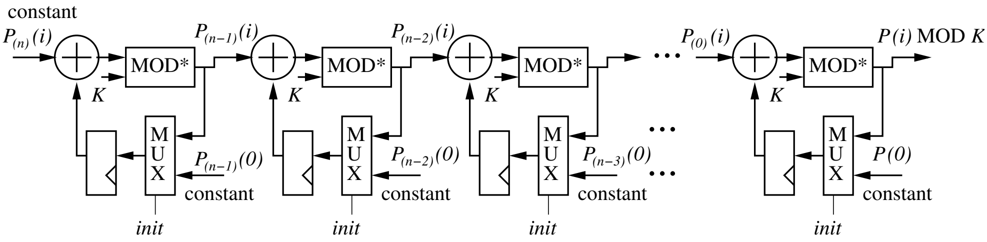


Fig. 43. A general structure for computing modulo of a polynomial with linearly incremented variable. The structure consists of systematically repeated computing elements.

The comparison, $x \geq K$, of (72) is carried out with the aid of subtraction operation. The subtraction operation, $x - K$, is computed only if x is greater or equal to zero, which is indicated by the MSB of the x. Otherwise, x is negative and the addition operation, $x + K$, is computed.

The four different conditions, which affect selection of addition or subtraction and selection of the multiplexer in Fig. 44(b), are tabulated in Table 11. If $x < 0$, the output is always $x + K$. If $x - K < 0$, the multiplexer selects x, since $x = x \bmod K$ in that case. An optional output signal can be used to indicate this condition. The

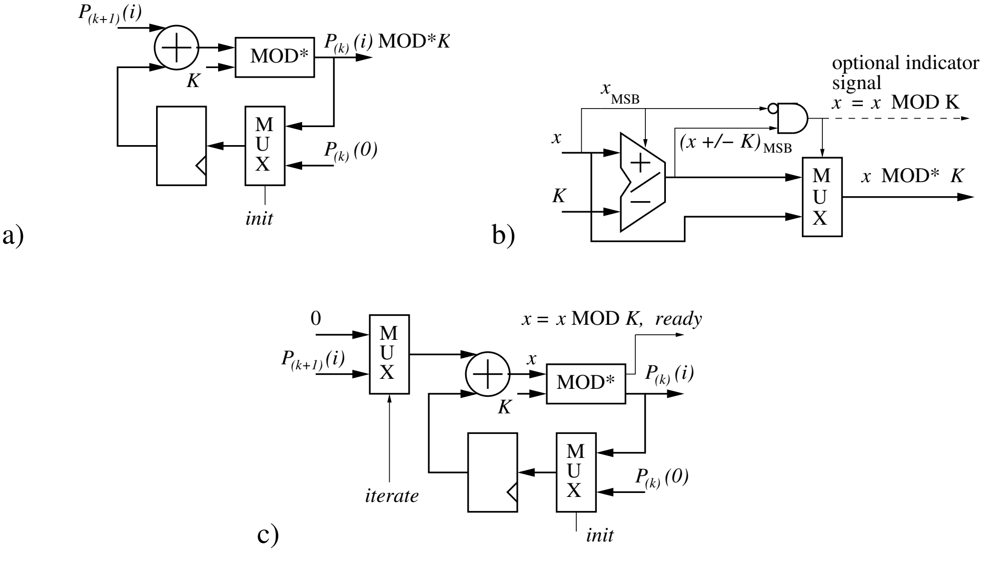


Fig. 44. Structures of: a) elementary block for structures computing the modulo of polynomials, b) computing element for $x \operatorname{mod}^* K$, and c) elementary block capable of computing $x \bmod K$ iteratively. The dashed line denotes an optional output signal.

Table 11. *Modes of the addition/subtraction unit and the multiplexer of $x \bmod^* K$ computing element.*

Sign of x	Add / sub	Sign of add / sub	MUX selects	$x = x \bmod K$ indicator
$x \geq 0$	sub	$x - K \geq 0$	$x - K$	False
$x \geq 0$	sub	$x - K < 0$	x	True
$x < 0$	add	$x + K \geq 0$	$x + K$	False
$x < 0$	add	$x + K < 0$	$x + K$	False

signal is shown as a dashed line in Fig. 44(b). Basically, the signal tells that the input, x, was already in the range of $x \bmod K$ and the unit did not change the value.

The optional output signal can be used if the x exceeds the valid domain, i.e., $x < -K$ or $x > 2K$, and the unit is used iteratively by feeding back the last result until the signal indicates that $x = x \bmod K$. In this case, the second last value would be already correct but the last iteration is required to raise the indicating signal. If the input, x, is always in the valid domain, the output is always correct and no iterations are required. A structure capable of iterative computation of $x \bmod K$ without limitations on the domain of x is shown in Fig. 44(c). The extra input signal *iterate* prevents addition operations to allow the current value to evolve to $x \bmod K$. The *ready* signal indicates when the iterative computation can be stopped. The option for iterative computation of $x \bmod K$ is the main reason why the $x \bmod^* K$ operation must be able to process also negative values of x. The intermediate values of the structure in Fig. 43 remain non-negative with non-negative initialization.

6.7.6 Case Study on 3G LTE Interleaver

As a case study, the method is applied on the 3G LTE interleaver. In principle, quadratic permutation polynomial interleaver [103] is used in 3G LTE. The interleaver permutes bits according to the interleaving function, $\Pi(i, K)$, which gives the permuted index of the ith bit. The interleaving function is defined as

$$\Pi(i, K) = (f_1(K)i + f_2(K)i^2) \bmod K \tag{108}$$

where K can take 188 different values [3]. The minimum and maximum of the K are 40 and 6114, respectively. The values of f_1 and f_2 are tabulated for each K

in [3], i.e., they are read from the LUTs. The (108) is far simpler than the older 3GPP interleaving function in Section 6.1.1.

In principle, the turbo decoder can use $\Pi(i, K)$ with linear increments, $i = 0, 1, 2 \ldots, K - 1$, as an address generator. However, internally the decoder applies also addressing in reverse order as indicated by the sawtooth sequence $B(i)$ in (59) and Fig. 34. This practice can be solved by caching the accessed data with an additional buffer memory, i.e., only the main memories are accessed with the aid of $\Pi(i, K)$ and the buffer memory is used for accessing the data in reverse order.

According to the proposed principles, the interleaving function (108) can be expressed with polynomials,

$$\begin{aligned}\Pi_{(0)}(i, K) \bmod K &= (\Pi(i, K) - \Pi(i-1, K)) \bmod K \\ &= (2f_2(K)i - f_2(K) + f_1(K)) \bmod K \\ \Pi_{(1)}(i, K) \bmod K &= (\Pi_{(0)}(i, K) - \Pi_{(0)}(i-1, K)) \bmod K \\ &= 2f_2(K) \bmod K\end{aligned} \tag{109}$$

and the initial values with $i = 0$ are

$$\Pi(0, K) = 0 \tag{110}$$

$$\Pi_{(0)}(0, K) = (-f_2(K) + f_1(K)) \bmod K \tag{111}$$

$$\Pi_{(1)}(0, K) = 2f_2(K) \bmod K\ . \tag{112}$$

Instead of tabulating values of f_1 and f_2 in a LUT, pre-computed constant values $(-f_2 + f_1) \bmod K$ and $2f_2 \bmod K$ can be stored without extra costs.

The $\Pi(i, K)$ interleaver function is implemented with the computing elements presented in Section 6.7.5. The implementation is synthesized with 130 nm technology, 1.35 V voltage and the area is given in terms of logic GEs in Table 12. Only the area of computing unit without any processor or other decoder implementation is presented in Table 12. For comparison, a straightforward implementation can be implemented with multiplier(s), adder, and $x \bmod K$ unit(s). The number of units depends on the parallelism of the implementation, i.e, whether the same unit is time-multiplexed. Since the (108) is quite simple and there are several ways to organize the computations, areas of elementary operations are tabulated in Table 12. As it is possible to limit the word length of intermediate terms by applying more than one

Table 12. *Complexity in terms of logic kGEs of the proposed 3G LTE conforming interleaving function unit and complexity of elementary operations for a straightforward implementation.*

Clock freq. (MHz)	100	150	200
Proposed structure (kGE)	0.963	1.402	2.326
13-bit adder (kGE)	0.064	0.064	0.064
13-bit multiplier (kGE)	0.848	0.911	1.083
13-bit div/mod unit (kGE)	3.191	N/A	N/A

modulo operation, the word length of the elementary operations is 13 bits in Table 12. On the last row, complexity of an off-the-shelf division and/or modulo component is given. The component contains six pipeline stages but higher clock frequencies were not achieved with prevailing operating conditions. The area estimates show clearly that the complexity of any straightforward implementation would be manifold when compared to the proposed structure.

7. INVERSE SQUARE ROOT APPROXIMATION AND QR DECOMPOSITION

Ever higher data rates require sophisticated transmission techniques. Such algorithms apply matrix operations which require highly non-linear division by square root operation. For example, for the current 3G UMTS the LMMSE estimation, in which Cholesky decomposition can be applied, has been proposed [34]. In the upcoming 3G LTE systems, symbol detection methods like LSD can apply QR decomposition of a matrix as shown in Fig. 2. In this Chapter, a scalable low-complexity inverse square root approximation method is proposed for baseband matrix operations and the method is applied in practice with a QR decomposition ASP.

7.1 Low-Complexity Inverse Square Root Approximation

In this Section, a method for approximating inverse square root is proposed. The method relies on binary presentation of the fixed-point number system and the principal idea of the method is to substitute the highly non-linear inverse square root function with a more implementation appropriate function with appropriate pre- and post-processing. The accuracy and complexity of the method can be adjusted with one design parameter. As a result, the method can accelerate any fixed-point system where cost efficiency and low power consumption are of high importance, and coarse approximation of inverse square root operation is required.

7.1.1 Previous Work

There are several methods to compute the inverse square root function. One of the basic approaches is to use LUTs for obtaining an initial value for iterations, which refine the value to higher accuracy [63, 82]. The main differences among these kind of methods are in the size and content of the LUT and the used iteration algorithm. In [82], a large multiplier is used since it is available in the targeted general purpose processor. In [136], savings are obtained by using a $m \times n$ multiplier, $m \leq n$, and utilizing the fact that less significant bits of intermediate result do not contribute to the accuracy of the final result. A software implementation using a LUT initialization followed by iterations is presented in [118]. Another software approximation in [70] relies heavily on the binary representation of floating-point numbers.

LUTs using low order polynomial approximation are applied in [55]. In [86], a second degree minimax polynomial approximation is followed by modified Goldschmidt iteration. Digit recurrence methods are proposed in [65,106]. The main disadvantage of using digit recurrence when compared to iterative algorithms is their linear convergence. Approximation based on a LUT followed by multiplication with operand modification is proposed in [105, 107], and used also in [93]. Argument reduction followed by series expansion is applied in [40]. Another approach is to work in logarithmic domain [31,46] where the computation of the inverse square root is straightforward [28,44].

For shorter word lengths and for using fixed-point numbers, table addition methods have been proposed. These methods consists of parallel LUTs and multi-operand additions. As a benefit, no multipliers are required. In [102], a symmetric table addition method (STAM) is developed as an extension to a simpler bipartite method. Selecting appropriate multipartite method, i.e., design space exploration, is considered in [35]. The STAM enhanced with an error correction term and internal presentation in exponent and mantissa form is used in [91].

When compared to the previously mentioned methods, the proposed method is not a derivative of any of the existing methods. The area costs are kept low as large LUTs and large multipliers are avoided. The method can be adjusted to work only in subunitary domain, which is sufficient for e.g., Cholesky decomposition with appropriate pre-scaling, and the accuracy of the method can be adjusted along with the complexity up to a certain level while maintaining high area-efficiency.

7.1.2 Low-Complexity Approximation Method

As the baseband functions are applied in receivers of, e.g., hand-held telecommunications devices, low complexity is important for decreasing the area costs and power consumption. Therefore, fixed-point number system is preferred, i.e., limited accuracy is applied. Naturally, the quantization creates some noise. The noise can be modeled as a sum of signal and a random variable [101]. In this Section, the targeted fixed-point number system has a fractional word length (FWL) of 11 bits and integer word length (IWL) of 5 bits, i.e., 16-bit words are assumed.

The main principle of the proposed method is to avoid straightforward approximation of $1/\sqrt{x}$ function which is highly non-linear in subunitary domain $0 < x \leq 1$. Instead, the more softly non-linear function $1/\sqrt{c+u}$ with $c \geq 1$ and $0 < u \leq 1$ is approximated. The usage of $1/\sqrt{c+u}$ is justified by the following fixed-point representations in two's complement format of x, c, and u. If the positive subunitary x has α leading zeros, c and u can be defined so that

$$x = \underbrace{0.00\ldots0}_{\alpha}c_{N-1}c_{N-2}\ldots c_0 u_{M-1}u_{M-2}\ldots u_0\ . \tag{113}$$

In other words, the bits of c and u do not overlap and the word lengths of c and u are denoted with N and M, respectively. Positive non-subunitary domain, $x > 1$, is presented similarly, except that the number of leading zeros, α, can have negative values. Since $c_{N-1} = 1$ for all valid values of x, the x can be presented with the aid of shift by α, i.e.,

$$x \times 2^{\alpha} = 1.c_{N-2}\ldots c_0 u_{M-1}u_{M-2}\ldots u_0 \tag{114}$$

$$\Leftrightarrow \quad x = 2^{-\alpha} \times 1.c_{N-2}\ldots c_0 u_{M-1}u_{M-2}\ldots u_0\ . \tag{115}$$

Thus, the desired form, $c + u$, is obtained and it can be noticed that u is a positive subunitary number. The targeted function can be written as

$$\frac{1}{\sqrt{x}} = \frac{1}{\sqrt{2^{-\alpha}(c+u)}} = 2^{\frac{\alpha}{2}}\frac{1}{\sqrt{c+u}}\ . \tag{116}$$

Two cases can be distinguished depending on the value of α, which represents the number of leading zeros of fixed point binary representation of x (113). This distinct behavior is obtained because the remainder of $\alpha/2$ in (116) can be either zero or one. For even values $\alpha = 2k$

$$\frac{1}{\sqrt{x}} = 2^{k}\frac{1}{\sqrt{c+u}}\ , \tag{117}$$

and for odd values $\alpha = 2k + 1$,

$$\frac{1}{\sqrt{x}} = 2^{k}\sqrt{2}\frac{1}{\sqrt{c+u}}\ . \tag{118}$$

In order to approximate (117) or (118), the expressions $1/\sqrt{c+u}$ must be considered. A tempting solution is to approximate $1/\sqrt{c+u}$ with binomial series. In principle, the $1/\sqrt{c+u}$ could be approached with arbitrarily high precision, as the binomial series converges. Multipliers are required if polynomial approximation [55, 86] or series expansion [40] are applied. Although the approximation with binomial series has a solid basis, it does not lend itself to low-complexity implementations due to the high order terms.

Linear Approximation

The characteristic of $1/\sqrt{c+u}$ are identified to determine a first degree polynomial for a low-complexity hardware implementation. The expression $1/\sqrt{c+u}$ is approximated with straight lines, i.e.,

$$1/\sqrt{c+u} \simeq a_t - b_t(c+u) \tag{119}$$

where $a_t, b_t > 0$ and subscript t is an integer interpretation of the concatenation $c_{N-2} \ldots c_0\alpha_0$. The number of approximating lines, i.e., the accuracy of the approximation, depends on the word length of c. Since, the MSB of c has always constant value, $c_{N-1} = 1$, the number of approximating lines is 2^N.

The range of the targeted expression is $1 \leq 1/\sqrt{c+u} \leq 1/\sqrt{2}$, since $1 \leq c+u < 2$. The range of c is defined by the word length, N, i.e., $1 \leq c \leq 2(1-2^{-N})$. Naturally, the range of u depends on N and M, i.e., $0 \leq u \leq 2^{-(N-1)} - 2^{-(M+N-1)}$. In practice, the approximating lines are formed by dividing the range of $c+u$ into evenly spaced regions, which are determined by the N MSBs of c. The values in the start and end points are given by $1/\sqrt{c}$ and the value of the last end point is $1/\sqrt{2}$. The linear approximation is illustrated in Fig. 45(a) where $1/\sqrt{c+u}$, with even α is approximated with $N = 1, 2, 3$. The error of approximation is shown in Fig. 45(b). The figures indicate that by increasing the word length N, the accuracy of the approximation can be adjusted conveniently.

For odd values $\alpha = 2k+1$, the $\sqrt{2}/\sqrt{c+u}$ is approximated in a similar fashion. The lines used for even values, $\alpha = 2k$, cannot be used without multiplication with $\sqrt{2}$ and, therefore, different approximating lines are preferred. To obtain the final result, i.e., the approximation of $1/\sqrt{x}$, the approximating straight lines in must be scaled

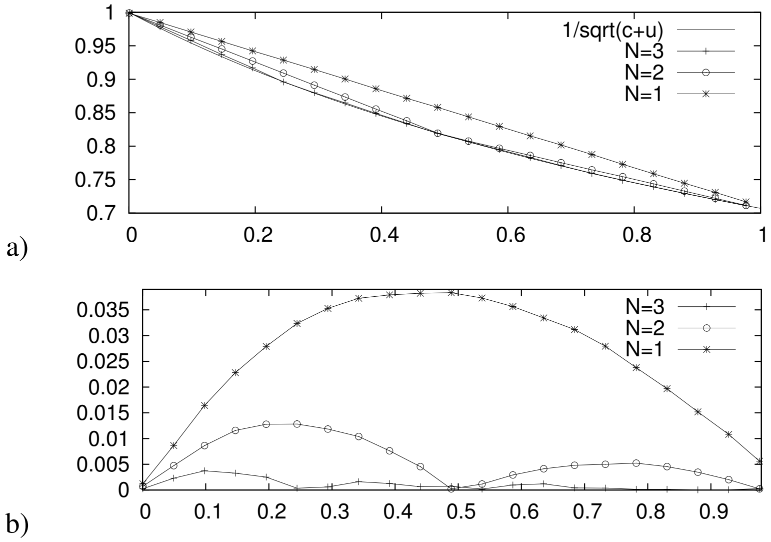


Fig. 45. *Linear approximation of* $1/\sqrt{c+u}$*: a) approximating lines, b) approximation error decreases as* N *is increased.*

with 2^k as shown in (117) and (118). The scaling can be carried out easily with shift operation, whose direction depends on the sign of α.

Coefficients for Hardware Implementation

The linear approximation has the form $a_t - b_t(c+u)$, which includes multiplication. However, for obtaining low complexity, the multiplications should be avoided. Braun multiplier adds shifted values of the multiplicand multiplied with one bit of the multiplier. The principle of adding shifted values can be used to approximate the product $b_t(c+u)$. Since $b_t \leq 1/2$, the product can be presented as

$$b_t(c+u) = d_{1,t}\frac{c+u}{2^1} + d_{2,t}\frac{c+u}{2^2} + \ldots + d_{M+N-1,t}\frac{c+u}{2^{M+N-1}} \tag{120}$$

where $d_{i,t} \in \{-1, 0, 1\}$. As division with powers of two can be implemented with hardwired shifting in hardware, an approximation of the previous form is suitable for low-complexity implementation. Naturally, the accuracy depends on the number of terms included in the sum. In the proposed method, at maximum three terms are included, i.e., an approximation,

$$b_t(c+u) \simeq d_{1,t}\frac{c+u}{2^{e_{1,t}}} + d_{2,t}\frac{c+u}{2^{e_{2,t}}} + d_{3,t}\frac{c+u}{2^{e_{3,t}}}\ , \tag{121}$$

in which $d_{i,t} \in \{-1, 0, 1\}$ and $e_{i,t} \in \{1, \ldots, 8\}$, are used. The coefficients $d_{i,t}$ and $e_{i,t}$ are searched for each approximating line, i.e., for each c and α_0, separately. Instead of three shifters with freely variable shift count, three multiplexers can be used.

7.1.3 Inverse Square Root Unit Implementations

The block diagrams of the hardware implementations of the inverse square root units are shown in Fig. 46. The Fig. 46(a) shows only the linear approximation $a_t - b_t(c + u)$. The top three multiplexers correspond with term $b_t(c + u)$ and the fourth multiplexer outputs a_t. The selections of multiplexer are controlled by parity of α and bits of c excluding the c_{N-1} which has constant value.

In the next block diagram in Fig. 46(b), the previous unit is instantiated in the inverse square root unit. The domain of the unit in Fig. 46(b) is positive subunitary, i.e., $0 < x \leq 1$, which is sufficient for e.g., the Cholesky decomposition. The structure is further extended in Fig. 46(c) to allow free domain, i.e., $x > 0$. Basically, non-subunitary domain of x results also in negative values of α and, therefore, both left and right shifting is required as indicated in Fig. 46(c). As the input signal x has wider word length in Fig. 46(c), the negative α is detected by comparing the number of leading zeros and IWL.

Only basic arithmetic and logic units are being used. The key components are priority encoder, adders, multiplexers, and shifters. Part of the functionality, e.g., constant scaling, is implemented by hardwiring bits to the new positions. Due to the scaling, word lengths of intermediate signals are relatively short. As the targeted accuracy depends on N, different implementations can be generated according to targeted application. Fig. 46(a) shows a general case, i.e., the number of inputs of multiplexers and N are free variables. In Figs. 46(b) and (c) $N = 1$ and, therefore, multiplexers are controlled solely by α_0. If $N > 1$, the c is obtained from the output of the first shifter(s) and the control signal is generated by concatenation of $c_{N-2} \ldots c_0$ and α_0. Only the structure of linear approximation depends on N, the other components in Figs. 46(b) and (c) remain unaltered if N is increased.

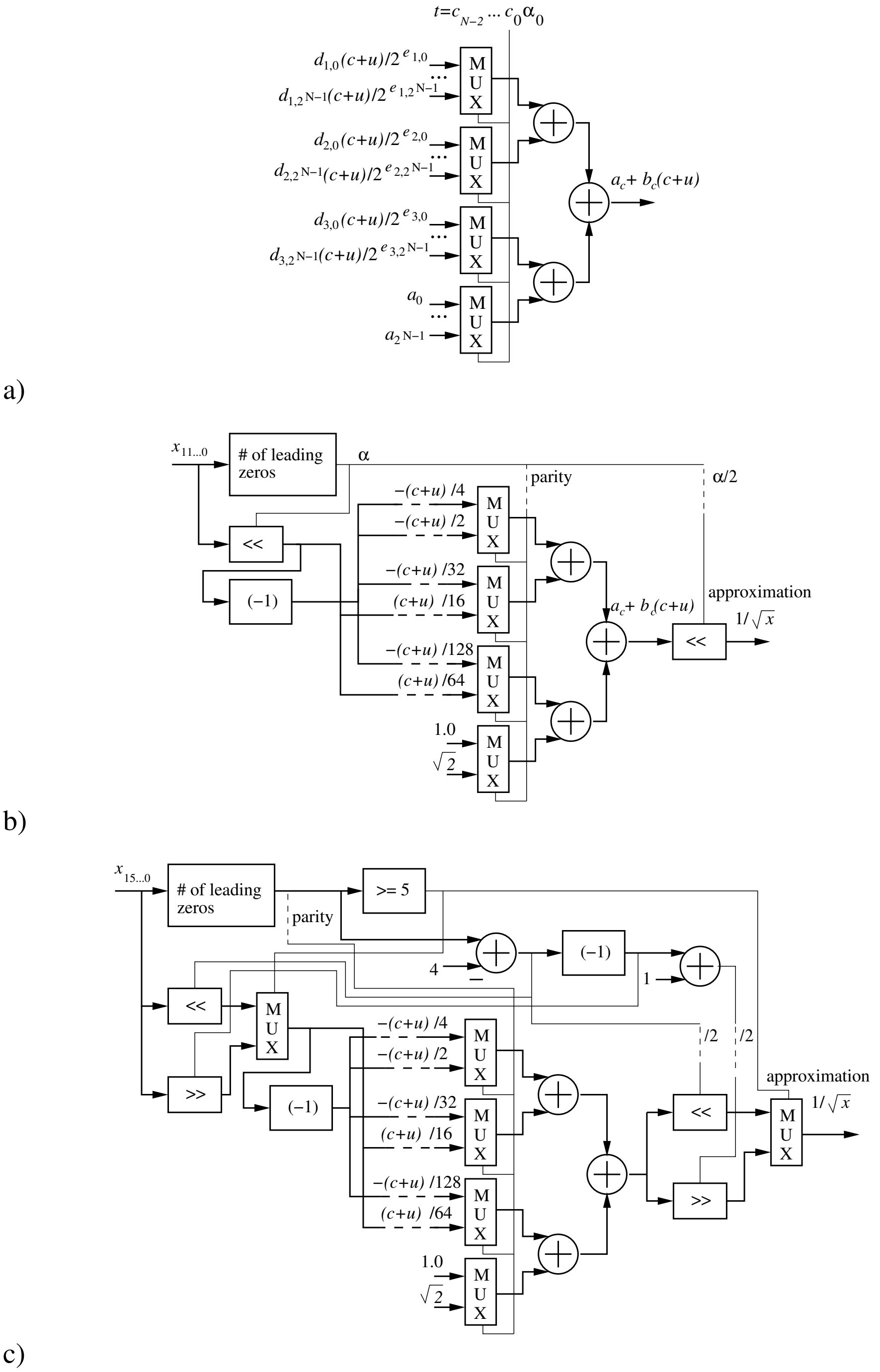


Fig. 46. *Units for: a) linear approximation with* $a_t - b_t(c + u)$*, b) approximation of* $1/\sqrt{x}$ *in subunitary domain, c) approximation in non-subunitary domain. Left and right shifting are denoted with* $<<$ *and* $>>$*, respectively. Negating is marked with* (-1)*.*

Table 13. *Estimated area of basic units and LUTs in compared implementations.*

unit	operands	GE
inverter	1	0.75
XOR	1×1	2.25
full adder	$1 \times 1 \times 1$	6.50
adder	8×6	34.8
adder	19×10	76.50
adder	20×9	78.8
multiplier	7×7	247
multiplier	20×20	2050
multiplier	22×23	2540
multiplier	16×56	4250
multiplier	56×56	14700
multiplier	76×76	26600

LUT size	GE
108×16	489
$2^9 \times 6$	506
216×8	594
$2^9 \times 8$	670
$2^9 \times 10$	872
$2^7 \times 36$	944
$2^9 \times 19$	2260
821×22	3720
$2^{10} \times 23$	4740
$2^{10} \times 25$	5070
$2^{11} \times 23$	9330

7.1.4 Comparison

In this Section, the efficiency of the proposed method is compared. The proposed method is synthesized with 130 nm technology. The areas of other methods are estimated by considering their most area expensive components, such as multipliers and LUTs, unless more accurate details are clearly specified in the referred design. Only the mantissa of floating-point implementations is considered, since its computation is similar in fixed-point number system.

Areas in terms of logic GEs of the synthesized arithmetic and logic operations with different word lengths are given in Table 13. On the contrary to simple cost estimation of LUTs in [86], the areas of all LUTs have been estimated individually. If structures of LUTs are not specified in detail, fair assumptions are made for the referred works. The synthesized LUTs are filled with pseudo random bits. The main reason for more accurate modeling of LUT complexity is that the area of the LUT depends both on the address line width and word length of the data. Estimated areas of all the LUTs are given in Table 13.

Table 14. *Suggestive comparison of inverse square root methods.*

method	FWL of result	area (kGE)	FWL / area
[82]	52	40.67	1.28
[136]	52	38.04	1.37
[86]	52	32.7	1.59
[105]	23	7.12	3.23
[102]	16	4.60	3.48
[40]	23	4.38	5.25
[91]	16	1.6	9.99
Proposed $N = 1$	4	0.41	9.86
Proposed $N = 2$	6	0.44	13.8
Proposed $N = 3$	8	0.51	15.6
Proposed $N = 4$	10	0.62	16.1

Compared Implementations

Since low area is emphasized in the targeted application domain of baseband processing, the methods are compared using the area-efficiency as the ratio of average FWL versus area. The metric is defined as

$$\text{area-efficiency} = 10^3 \frac{\text{FWL}}{\text{area in GEs}} . \tag{122}$$

For single precision (SP) methods the accuracy is 23 bits and for dual precision (DP) methods 52 bits. The area-efficiency results for all the methods are shown in Table 14. The average accuracy of the proposed method in Table 14 is obtained in the subunitary domain. There are four versions of the proposed method with design parameter $N = 1, 2, 3, 4$. The results show that the proposed method has lowest area and even if the accuracy is adjusted with N, the area-efficiency remains highest except with $N = 1$. Naturally, the accuracy is relatively modest, as the lowest area is preferred instead of high accuracy.

The first method in Table 14 is targeted to DP general purpose processor [82]. It requires LUTs of sizes $2^{10} \times 23$ and $2^{11} \times 23$ and a multiplier for 76×76 operands. Since the implementation is targeted to a general purpose processor, the hardware resources are not dedicated only to the inverse square root function. In [136], two 16×56 and one 56×56 multipliers are required. The total memory size is 72192 bits divided into four tables. For smaller gate count, a uniform division to four tables of

18048 bits with 22-bit word length, which is the widest word fetched from the tables, is assumed. High speed is emphasized in [86] and the method with single multiply and accumulate unit is compared, as it has better area-efficiency. The authors also report the complexity of 5030 full adders and, therefore, their value is used. In [105], SP floating-point numbers are targeted. A $2^{10} \times 25$ LUT is required and a 20×20 multiplier. In addition, a requirement of 15 inverters is reported. The STAM is used in [102]. The smallest total LUT size is obtained with four LUTs of sizes $2^9 \times 19$, $2^8 \times 10$, $2^8 \times 8$, and $2^8 \times 6$. In addition, a sum of all the data read from LUTs must be generated, which requires adders with operand sizes 19×10, 8×6, and 20×9. Also a requirement of 45 XOR gates is reported. Both SP and DP are targeted in [40] but the method for SP gives better area-efficiency. The SP method requires $2^7 \times 36$ LUT, four 4×4 multipliers, and one 22×23 multiplier. Fixed-point number systems are targeted in [91]. The method applied STAM enhanced with added correction value. Estimated complexity of 625 GEs and LUT size of 3456 bits are given in [91]. Since the structures of LUTs are not reported, it is assumed that, due to the STAM, the memory is divided at least to two LUTs. Also a 16-bit word length is assumed for the LUTs. Several smaller LUTs or shorter word length would degrade the area-efficiency. The estimated complexity of LUTs is added to the reported gate count.

Power Consumption

The power consumption of the largest proposed unit ($N = 4$) with 100 MHz and 1.35 V voltage is 0.339 mW, which includes the power required by input and output registers. Naturally, the static power consumption is proportional to the area and, therefore, low complexity has been targeted. The dynamic power is proportional both to the area and average switching activity of the gates. Even if the average switching activity of competitive methods cannot be estimated sufficiently accurately, the differences in the area are significant. For example, [91] has the smallest area, 1602 GE, of the referred methods and the average switching activity of [91] should get as low as $622/1602 \times 100\% = 39\%$ of the average switching activity of largest proposed unit (622 GE, $N = 4$) to achieve roughly the same dynamic power consumption.

7.2 QR Decomposition

The QR decomposition is one vital function of the MIMO receiver. In this Section, a processor based complex-valued QR decomposition is presented. The matrix is decomposed according to the modified Gram-Schmidt process [42] and the processor is enhanced with complex arithmetic SFUs and an inverse square root approximation SFU, which applies the principles presented in previous Section. The proposed processor fits well with the throughput requirements of a MIMO–OFDM receiver.

7.2.1 Previous Work

There are several ways to obtain QR decomposition. It can be computed, e.g, using Householder transformations, Givens rotations, or Gram-Schmidt process. The inherent regularity of matrix operations can be utilized with systolic structures [62]. Elementary operations can be alleviated with a coordinate rotation digital computer (CORDIC) algorithm [13] which lends itself to low-complexity hardware realization. Such an approach is followed, e.g., in [10,69,72]. Another way to alleviate hardware complexity is to carry out computations in logarithmic domain. This practice is used in [99].

The MIMO receiver requires relatively small matrix size and low throughput for QR decomposition. Therefore, extensively parallel solutions like systolic array processors in [62, 72] or processor arrays with reduced dimensions [69] can be oversized for such systems. However, even if the throughput were oversized, the short delay obtained with such processors, can be beneficial.

In [37], a complex-valued matrix inversion based on QR decomposition is presented. The method uses squared Givens rotations. Instead of traditionally triangular array of processing elements, a linear array structure is used. Inverting a complex-valued 4×4 matrix takes 175 cycles. As a drawback, such an array processor is not flexibly programmable like ASPs. In [69], a floating-point real-valued programmable ASP for QR and singular-value decomposition is presented. The ASP contains CORDIC module and ASPs can be structured as an array for high throughput. The ASPs are programmable, but the structure resembles array processors as the processing elements are substituted with the presented ASPs. A structure with Nios processor and CORDIC accelerator in FPGA is presented in [10]. The CORDIC elements are used for QR decomposition and the following back substitution for solving a set of equations is computed on Nios processor. As a drawback, the accelerating CORDIC elements are not tightly connected to the data path. Instead, the CORDIC accelerator and Nios processor communicate via memory. Computations in $\log_2$ domain are applied in [99]. The parallel architecture for 4×4 matrix inversion with the aid of QR decomposition takes 72 kGEs and achieves a latency of 0.24 μs.

Typically, floating-point presentation is avoided to save area and power in baseband processing. With fixed-point number system the IWL and FWL must be chosen appropriately to avoid overflows and to maintain desired accuracy. An error analysis of fixed-point implementation of QR decomposition with Gram-Schmidt process is presented in [98].

In this Thesis, the ASP implementation of the QR decomposition targets low complexity, flexibility, and programmability as the main objectives. As all the accelerating SFUs reside on the datapath of the processor, they can be used without extra overhead. The flexibility is a consequence of the programmability and arithmetic SFUs, which can be used also for other computations. The simple approximation method of the $1/\sqrt{x}$ contributes to the low-complexity.

7.2.2 Modified Gram-Schmidt QR Decomposition

In this Thesis, the modified Gram-Schmidt algorithm [42] is used for the proposed QR decomposition. The modified Gram-Schmidt algorithm has better numerical properties than the classical Gram-Schmidt algorithm. As high throughput as achievable with systolic array processors or decomposition level pipelining using Givens rotations is not targeted. Instead, by using modified Gram-Schmidt algorithm the ASP can be composed of a set of FUs or SFUs, which could be used easily also with other applications, in principle. Basically, the algorithm orthogonalizes a set of vectors. QR factorization with modified Gram-Schmidt algorithm is shown in Fig. 47 for square matrix $\boldsymbol{H}_{n\times n}$. It decomposes $\boldsymbol{H}_{n\times n}$ to the orthogonal $\boldsymbol{Q}_{n\times n}$ and upper triangular $\boldsymbol{R}_{n\times n}$. Conjugated transpose is denoted with $(\cdot)^H$. The lines 2 and 3 show that division by square root is required as the elements are divided by diagonals which are norms, $||\cdot||$. The division can be substituted with multiplication by inverse square root.

```
for $k = 1 : n$
    $\boldsymbol{R}_{k,k} = \|\boldsymbol{H}_{1:n,k}\|$
    $\boldsymbol{Q}_{1:n,k} = \boldsymbol{H}_{1:n,k} / \boldsymbol{R}_{k,k}$
    for $j = k + 1 : n$
        $\boldsymbol{R}_{k,j} = \boldsymbol{Q}^{H}_{1:n,k} \boldsymbol{H}_{1:n,j}$
        $\boldsymbol{H}_{1:n,j} = \boldsymbol{H}_{1:n,j} - \boldsymbol{Q}_{1:n,k} \boldsymbol{R}_{k,j}$
    end
end
```

Fig. 47. *Modified Gram-Schmidt algorithm decomposes $\boldsymbol{H}_{n \times n}$ to the orthogonal $\boldsymbol{Q}_{n \times n}$ and upper triangular $\boldsymbol{R}_{n \times n}$. Conjugated transpose is denoted with $(\cdot)^H$.*

7.2.3 Application-Specific Processor for QR Decomposition

The proposed QR decomposition is implemented on a TTA processor [32]. However, the presented principles can also be applied on other customizable processors. The organization of the proposed TTA processor is shown in Fig. 48. There are two dedicated units for complex-valued arithmetic in the processor. The applied practice of using complex numbers as the native data type accelerates computations and simplifies programming significantly. In practice, the complex numbers are presented as 32-bit words where real and imaginary parts use upper and lower 16 bits, respectively. The SFUs extract real and imaginary parts when necessary and compose complex numbers, so all the data passed between the computing resources has always the same representation. Extracting real and imaginary parts or composing complex numbers is very simple in hardware, i.e., hardwiring bits to the new positions in the target signal.

The complex multiplier SFU of the processor supports two operations; it can compute the complex-valued multiplication and complex-valued multiplication with conjugated multiplicand. The modified Gram-Schmidt algorithm in Fig. 47 requires conjugated multiplication for computing the vector norm and for conjugated dot product. The second complex-valued arithmetic SFU supports complex addition and subtraction operations. The modified Gram-Schmidt algorithm requires division with the (real-valued) norm $\| \cdot \|$. However, the multiplication is simpler operation in hardware than division. For this reason, the division is substituted with multiplication with an inverse value, i.e., inverse of Euclidean norm. Thus, it is inverse of the square root. With this substitution two demanding operations, division and square root, are replaced with computation of the inverse square root function, $1/\sqrt{x}$, and multiplication. Naturally, also $1/\sqrt{x}$ is very demanding function but the low com-

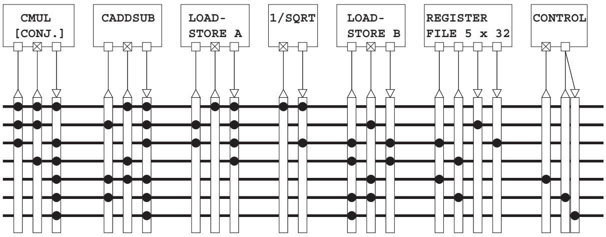


Fig. 48. *Proposed TTA processor for QR decomposition has LSUs, RF, SFU for the approximation of $1/\sqrt{x}$, and SFUs for complex-valued addition, subtraction, and optionally conjugated multiplication. Filled circles denote connections to buses.*

plexity approximation principles presented in the Section 7.1 are used. The approximation of $1/\sqrt{x}$ can be used also for $\sqrt{x}$ function with one multiplication, since $x\frac{1}{\sqrt{x}} = xx^{-\frac{1}{2}} = x^{\frac{1}{2}} = \sqrt{x}$.

7.2.4 Throughput and Complexity

The spectrum of an OFDM signal consists of several subcarriers and the channel matrix $\boldsymbol{R}$ must be computed for all the subcarriers at least within coherence time, t_{coh}, defined in (5). With $f_{carrier} = 2.4$ GHz, $v_r = 250$ km/h, and $c = 3 \times 10^8$ m/s the coherence time $t_{coh} = 1.8$ ms. The proposed QR decomposition processor is synthesized on 130 nm standard cell technology for obtaining complexity and performance estimates. The proposed QR decomposition of complex-valued 4×4 matrix takes 139 clock cycles. So, the decompositions with, e.g., 2048 subcarriers takes

$$T_{QR} = (2048 \times 139)/f_c \tag{123}$$

where f_c is the clock frequency. The execution times for several clock frequencies with 2048 subcarriers are tabulated in the Table 15. The execution time should be compared to the available time frame of 1.8 ms. Naturally, some computing time should be reserved for higher level control flow and polling for a new input matrix, $\boldsymbol{H}$, in real application.

Table 15. *Area of the proposed processor and execution time of 2048 QR decompositions.*

Clock frequency	Area	Execution time T_{QR}
269 MHz	23.2 kGEs	1.058 ms
212 MHz	17.7 kGEs	1.343 ms
160 MHz	16.3 kGEs	1.779 ms

8. LIST PROCESSING FOR SYMBOL DETECTION

MIMO techniques can be used to enable high-rate wireless communication [83]. However, to exploit additional gain of MIMO transmission efficient symbol detection like LSD [50] is required. The LSD algorithm requires maintaining a list of candidate symbols with shortest PEDs to the received symbol and the list processing is likely to be the bottleneck of the LSD and, therefore, there is a strong demand for efficient list processing implementations. For energy efficiency, memory-based list is preferred over registers with long list lengths. On the other hand, with short lists, a register based list processing can be a preferred solution for its rapid processing. In this Chapter, SFUs for alleviating both memory and register based list processing are presented.

8.1 List Processing in List Sphere Decoding

The purpose of LSD is to find the transmitted multidimensional symbol vector without trying every alternative systematically. The LSD computes PEDs by step-wise increasing the dimensionality of the current candidate and it maintains a list of the best candidates which are found so far. The elements of the list contain the distance to the received symbol and the symbol identifier. When a new PED is computed, it is compared with the maximum distance in the list. If the new candidate has shorter distance, the element with the maximum distance is replaced with the new one. More formally, if all the candidates are denoted with $\mathcal{A}$ and $\mathcal{L}$ presents a list of n elements, then the maximum element of $\mathcal{L}$ is less or equal to the minimum of $\mathcal{A}$ from which elements of $\mathcal{L}$ are excluded, i.e.,

$$\max \mathcal{L} \leq \min(\mathcal{A} \setminus \mathcal{L}),\, |\mathcal{L}| = n \tag{124}$$

where maximum, minimum, and comparison consider the distance information of the elements.

If the LSD computes distances and manages the list in a pipelined fashion, then the slower task will determine the throughput of the pipeline. Since the list update process is difficult to parallelize, it is usually the bottleneck, which limits the decoding throughput. In this case, the throughput of the LSD is directly proportional to the throughput of the list updating.

8.2 Memory-Based List Updating for List Sphere Decoders

An obvious solution for fast list updating is a register-based approach where the whole list can be processed in parallel. Even if the complexity of the register-based approach can be reduced [138], it still has high energy consumption. Memories consume less energy and, therefore, they allow long list lengths. As a drawback, the list cannot be processed in fully parallel manner. Using the heap data structure for list updating has been suggested in LSD decoder studies [131, 132] earlier. An off-the-shelf DSP using the heap data structure is applied for LSD in [56] where long latency memory accesses were considered as drawbacks. The heap is a natural choice with long list lengths, since the insertion has logarithmic computational complexity, $O(\log_2 n)$. However, actual implementations of fast heap insertion routines have not been discussed in detail in the previous studies.

In this Section, two SFUs for the memory based list insertion methods are proposed and they are demonstrated with a highly parallel processor template. The list insertion is divided both into hardware unit and software routine. The two units differ in the performance and hardware requirements. The faster method applies principles of software pipelining, even if the list insertion is inherently data dependent operation. It is shown that the list can be maintained with very low computational overhead compared to the $O(\log_2 n)$ order of the algorithm. Thus, the proposed methods and SFUs can be used to accelerate any memory-based LSDs with a long list length.

8.2.1 List Updating with Heap Data Structure

Throughout this Chapter, it is assumed that both the distance and symbol identifier are packed to the same data word, i.e., the element of the list. When any comparisons between elements are made, the distances are compared. When elements are loaded,

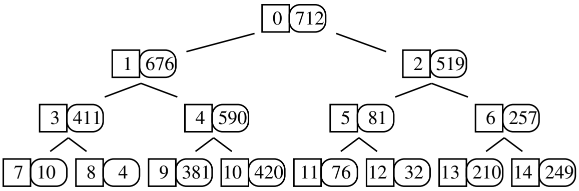


Fig. 49. *An example of heap data structure. Addresses are marked in square boxes and data in rounded boxes.*

stored, or swapped, both the distance and symbol information are carried in the same data word.

Updating a sorted list is straightforward. In that case, the maximum element of the sorted list can be obtained easily and the new candidate element can be compared with the maximum element. As a drawback, sorting either as a separate step or with insertion sort are complex procedures requiring several clock cycles and accesses of list elements. If the list is not sorted, the maximum element must be searched somehow. Again, simple brute force search would require several clock cycles and several memory accesses. However, time consuming sorting and search can be avoided if the applied data structure alleviates fast look-up of the maximum element and if the insertion of a new element is simpler than insertion sort. The heap data structure possesses these features. The maximum element is always in a known position, i.e., the root of the heap, and insertion routine can be accelerated with low complexity hardware unit as will be shown in Section 8.2.2.

Heap Condition

An example of a heap is shown in Fig. 49. The data is organized in binary tree so that each non-root node, d, satisfies the heap condition,

$$val(parent(d)) \geq val(d) \tag{125}$$

where the parent and value of the node are denoted with $parent()$ and $val()$, respectively. Thus, as long as the heap condition (125) is valid, the root node will contain the node with the maximum value. The data structure does not consider whether the left or right child node has respectively greater value. The example heap in Fig. 49 shows also that data cannot be extracted in sorted order straightforwardly from the heap, i.e., maintaining the heap condition (125) does not automatically sort inserted elements.

Data Organization in Memory

The data organization in memory is also showed in Fig. 49 where the addresses of nodes are marked in square boxes. It can be noticed that for each non-leaf node at address, a, the addresses of its left and right children are $2a + 1$ and $2a + 2$, respectively. The low complexity addressing is vital for fast list insertion routine. If nodes contained links to other nodes, the delay of addressing would be manifold due to the additional memory accesses.

Insertion

Insertion to a full heap is carried out as follows:

1. If $val(candidate) < val(root)$, current $root$ is replaced with $candidate$ and it will become the current parent node. Otherwise heap remains untouched.
2. If current parent node is smaller than any of its children nodes, the parent node is swapped with maximum of its children nodes. The swapped child node will become the new parent node, for which the same procedure is repeated.

Naturally, the process is terminated also when the leaf stage of the tree is reached.

Avoiding Partially Filled List

If the heap is not full, the insertion routine begins from the leaf stage and proceeds up to the root by swapping nodes until the heap condition (125) is not violated. This would need an additional step to decide which insertion routine is used. However, such an approach is not efficient. Another way is to fill the heap with maximum values, so that the heap is always full and there will be only one insertion routine.

With list length n, filling the heap takes n clock cycles with single-port memory. The insertion routine takes $f(n)$ cycles per insert. The routine is run a total of kn times.

So, the number of clock cycles with separate filling routine would be

$$\begin{aligned} C_{fill} &= knf(n) + n && (126) \\ &= n(kf(n) + 1)\,. && (127) \end{aligned}$$

If there are two insertion routines and the decision of choosing the right routine and jump latency take b clock cycles, the total execution time will be

$$\begin{aligned} C_{branch} &= kn(f(n) + b) && (128) \\ &= n(kf(n) + kb) && (129) \end{aligned}$$

clock cycles. Typically, $kb > 1$, so it is more efficient to fill the heap as a separate step and to have only one type of insertion routine. With dual-port memory or parallel accessible memory modules, the initial filling takes even less than n clock cycles.

8.2.2 List Updating Special Function Units

Two alternative list updating units are proposed. The main differences are in the memory throughput requirements and the number of clock cycles per insertion.

List Updating SFU for Non-Pipelined Execution

The first list updating unit is used without applying principles of software pipelining. The proposed unit is applied in practice in the K-best LSD TTA processor presented in [14]. The computation proceeds iteratively by alternating load operations and simultaneous computation of new load addresses and store operations. The decision whether to choose the left or right branch is based on the loaded nodes. Thereafter, new children nodes can be loaded again. So, there is a dependency between load addresses and data loaded previously.

Input and output ports of the proposed list updating SFU are shown in Fig. 50(a). Clarifying naming is used to indicate purposes of ports in Fig. 50(a). The unit processes a node triple of parent, and two children at once as indicated by the input ports. On every invocation, the SFU generates operands for LSUs as indicated by the output ports. The SFU is used as follows:

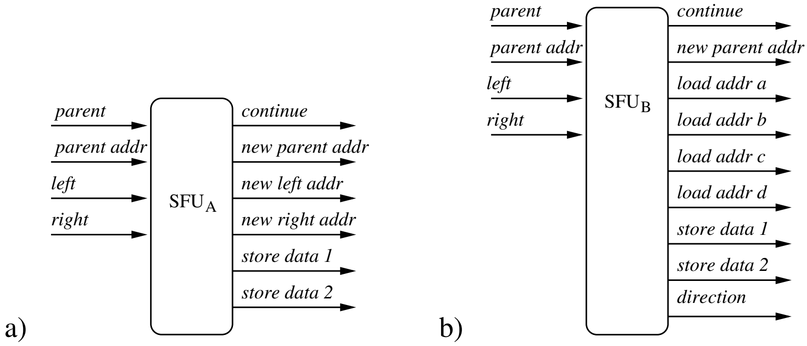


Fig. 50. *List updating SFUs for: a) non-pipelined execution, b) software pipelined execution.*

1. The candidate item is given in a register. Also a copy of the current root element is held in a register for fast access. Therefore, the first stage of the heap is processed as a special case. The SFU is used to compute the minimum of the root and candidate nodes. The result will overwrite the root node. In other words, in the first step it is checked if the candidate can be inserted to the heap at all. On the same clock cycle, the left and right children nodes are loaded.
2. On the second clock cycle the root node is the current parent node. The left and right nodes, the (new) parent node, and the address of the parent are fed to the unit. The addresses of the next left and right nodes are computed. The unit chooses from the left and right the next parent node, whose address is outputted. The unit also outputs data which will overwrite the current parent node data and data which will overwrite the data on the next parent node. In this way the current parent and one of its children can be swapped in memory.
3. On the third clock cycle, data is loaded according to the addresses generated on the previous cycle. Since data is not available on this cycle, there is no further processing.
4. On the fourth clock cycle, the computation continues as on the second clock cycle. In addition, the data generated two clock cycles ago is written to the memory on the current cycle.
5. Thereafter, the computation continues as on the previous two cycles until the last stage is reached. After the last stage, there are no load operations but the data is written to the memory immediately after the last stage.

```
procedure nonPipelined($parent, left, right, parentAddr$)
  if $parent \leq left \wedge left > right$ then
    $storeData1 := left$
    $storeData2 := parent$
    $newParentAddr := 2 \times parentAddr + 1$
    $continue :=$ True
  elsif $parent \leq right \wedge right \geq left$ then
    $storeData1 := right$
    $storeData2 := parent$
    $newParentAddr := 2 \times parentAddr + 2$
    $continue :=$ True
  else
    $storeData1 := parent$
    $storeData2 := left$
    $newParentAddr := 2 \times parentAddr + 1$
    $continue :=$ False
  end
  $rightAddr := 2 \times newParentAddr + 1$
  $leftAddr := 2 \times newParentAddr + 2$
end
```

a)

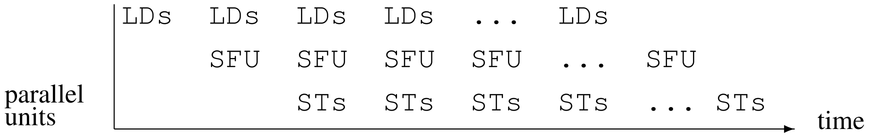
```
procedure swPipelined($parent, left, right, parentAddr$)
  if $parent \leq left \wedge left > right$ then
    $storeData1 := left$
    $storeData2 := parent$
    $newParentAddr := 2 \times parentAddr + 1$
    $direction :=$ toLeft
    $continue :=$ True
  elsif $parent \leq right \wedge right \geq left$ then
    $storeData1 := right$
    $storeData2 := parent$
    $newParentAddr := 2 \times parentAddr + 2$
    $direction :=$ toRight
    $continue :=$ True
  else
    $storeData1 := parent$
    $storeData2 := left$
    $newParentAddr := 2 \times parentAddr + 1$
    $direction :=$ toLeft
    $continue :=$ False
  end
  $loadAddrA := 4 \times newParentAddr + 3$
  $loadAddrB := 4 \times newParentAddr + 4$
  $loadAddrC := 4 \times newParentAddr + 5$
  $loadAddrD := 4 \times newParentAddr + 6$
end
```

b)

Fig. 51. *Functional description of SFUs for: a) non-pipelined execution, b) software pipelined execution.*

There is a status bit *continue* as an output port in Fig. 50(a). On every cycle, the status bit of the SFU indicates whether the processing should be continued or stopped. Because the jump latency is typically several clock cycles, the processing cannot be stopped immediately. However, the proposed SFU preserves the valid heap structure even if the processing should have been stopped. During the clock cycles that have to be waited until the jump is really executed, no nodes are swapped but the values are only rewritten to their original locations.

The functionality of the SFU is described in detail in Fig. 51(a). Only three comparators are required for hardware implementation. On the first clock cycle of the list insertion the minimum of two positive numbers is required. The description in Fig. 51(a) shows that it can be computed if operands are fed to the *parent* and *left* input ports and the *right* input port is set to zero.



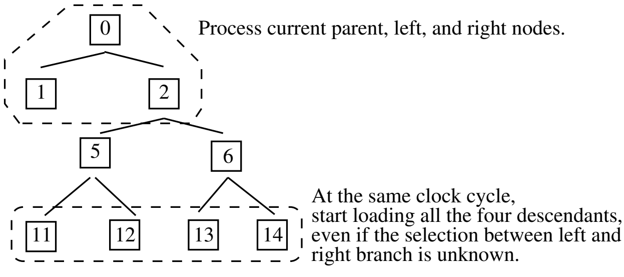
```
         |LDs  LDs  LDs  LDs  ...  LDs
         |     SFU  SFU  SFU  SFU  ...  SFU
parallel |          STs  STs  STs  STs  ...  STs
units    |________________________________________> time
```

Fig. 52. *In software pipelined version all the participating units, the SFU and LSUs with load (LD) and store (ST) operations run in parallel.*

List Updating SFU for Software Pipelined Execution

The second proposed unit can be used in software pipelined way. With software pipelining all the participating units run in parallel during the computation kernel. In this case, computations with the SFU and loading and storing of data are run in parallel, which is exemplified in Fig. 52.

The inherent data dependency between load addresses and previously loaded data makes software pipelining demanding. Even if the SFU cannot generate exactly two valid load addresses due to the data dependency, it is possible to generate a set of four addresses, which includes the required two addresses. Fig. 53 exemplifies this practice. In Fig. 53 it is not known should the pair (11,12) or (13,14) be loaded. For this reason, both pairs are loaded. On the next clock cycle, it is known which pair is selected and the next set of four nodes is loaded again. The selected pair of nodes is fed to the SFU with guarded parallel instructions, so the conditional execution will not take extra clock cycles. In other words, the software pipelined execution is made possible by prefetching four words from the memory on every clock cycle as the heap is traversed. Only the first stage is a special case as there are only two children



Process current parent, left, and right nodes.

At the same clock cycle, start loading all the four descendants, even if the selection between left and right branch is unknown.

Fig. 53. *Prefetching loads four descendants, from which only two will be selected. Selection decision will be available at the next clock cycle.*

nodes to be fetched. The input and output ports of the SFU are shown in Fig. 50(b). The *direction* output indicates which branch was selected and it determines which pair of the two prefetched pairs will be processed next. The functional description in Fig. 51(b) shows that the fundamental operations are almost similar to the non-pipelined version. The main difference is generating addresses one clock cycle ahead of processing.

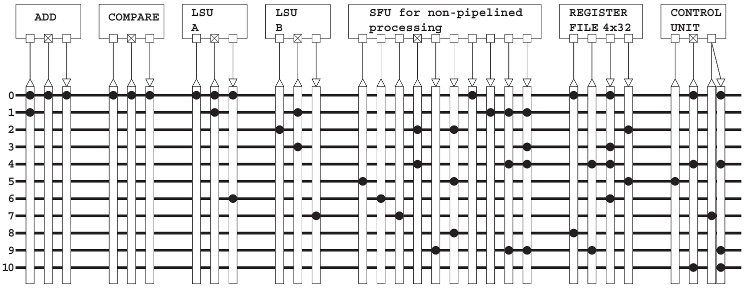
8.2.3 Resource Requirements and Performance

Both list updating methods can be characterized by resource requirements and the worst case throughput. In principle, the proposed units can be used in pure hardware implementations or as SFUs of any customizable processor with sufficient parallelism and memory interfaces. In this Thesis, the list processing with the SFUs is experimented with TTA [32] processors.

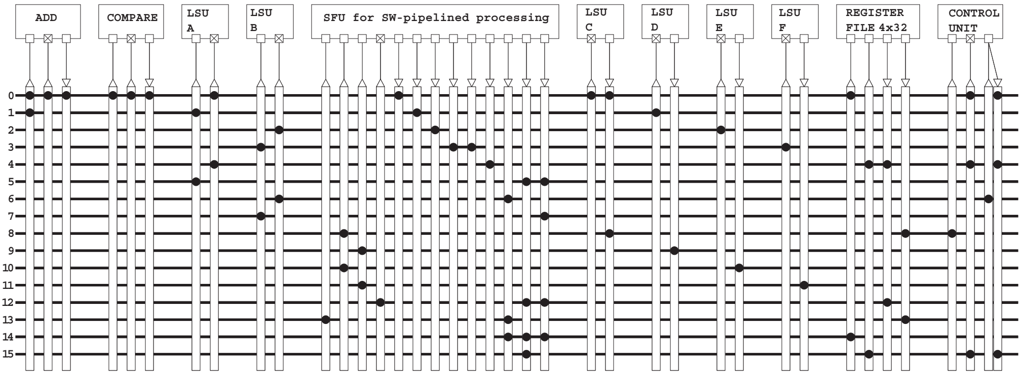
Requirements of Processors Applying the SFUs

The Figs. 54(a) and (b) show structures of TTA processors capable of applying the SFUs. The number of buses in the processor equals to the required number of parallel data transports to use the SFUs. The software part issuing the data transports is an assembly routine, which basically loads and stores data according to the results of the SFU. The next main difference is the number of required LSUs. The SFU for non-pipelined execution requires 11 parallel data transports and two LSUs and the SFU for software pipelined execution requires 16 parallel data transports and six LSUs.

The SFU for non-pipelined execution requires dual-port memory. The Figs. 53 and 49 show that regardless of the selected branch, the SFU for software pipelined execution loads simultaneously descendant nodes whose addresses have the form $a, a+1, a+2, a+3$. Thus, the memory can be divided into four banks of dual-port memory with low-order interleaving bank selection function. Usage of the proposed SFUs requires also that the read cycle time of the memories is one clock cycle. Since the total memory size equals to the list length, the costs of memory do not grow excessively. If the memory has higher latency, the prefetching should be extended to cover more heap stages. As a drawback, such a practice would suffer from a more demanding parallel access pattern. Typically, widely used DSPs are able to fetch even three



a)



b)

Fig. 54. *TTA processors which are capable of exploiting the proposed SFUs: a) processor for non-pipelined processing, b) processor for software pipelined processing. The memory is accessed with LSUs. Filled circles denote connections with buses.*

16-bit operands [114] and high-end DSPs [112] can fetch even four 32-bit words in parallel. However, the latency of the memories in typical DSPs can be high, since the memories are larger and they are not designed particularly for LSD algorithms.

Latency of List Insertion

Computational complexity of insertion to a heap is of order $O(\log_2 n)$ for binary tree-shaped heaps. Since the main term of the number of clock cycles will have the $\log_2$ form in any case, lower order terms and constant coefficient of logarithm term characterize how efficiently the list is processed. In the following, the worst case execution time is analyzed.

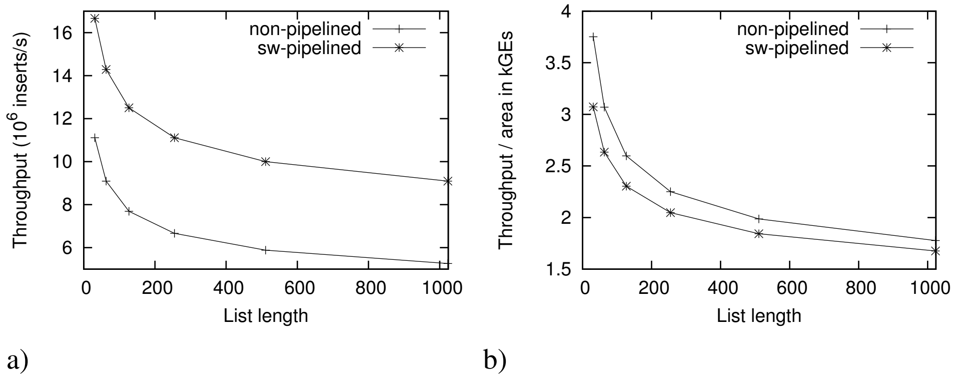


Fig. 55. *Comparison of list processing methods: a) throughput, b) area-efficiency as throughput / area of SFUs and accompanying LSUs.*

It is assumed that the number of elements in the heap is $n = 2^m - 1$ and all the leaves have the same depth. The list insertion routine without software pipelining takes

$$C_{\textit{non-pipelined}} = 2\log_2(n+1) - 1 \tag{130}$$

cycles per insertion with one clock cycle read cycle time of the list memory. Thus, one stage of the heap introduces always a delay of 2 clock cycles. The constant term, -1, originates from the computation of the root stage, which can be computed immediately, since a copy of the root node is cached in a register. Also processing of the last stage is a special case as there are only store operations.

The software pipelined insertion routine takes

$$C_{\textit{SW-pipelined}} = \log_2(n+1) + 1 \tag{131}$$

cycles with one clock cycle read cycle time of the list memory. With this routine, one stage is processed in one clock cycle. The additional constant term, $+1$, originates from the last instruction, which writes the nodes of the last swap operation to the memory.

Discussion

The SFUs and accompanying LSUs are synthesized with 130 nm technology with 100 MHz clock frequency. Naturally, the software pipelined execution has better throughput than without pipelining. The throughput of the SFUs is illustrated in Fig. 55(a) as a function of the list length. The SFUs for non-pipelined and pipelined execution take 1822 and 2007 GEs, respectively. So, the difference between the areas of plain SFUs is low if the accompanied LSUs are not taken into account. The area-efficiency of the units can be defined as throughput / area of the SFUs and accompanying LSUs. The applied LSU consists mainly of registers and minor control logic and it takes 570 GEs with the same operating conditions. The ratio between the areas of SFUs with LSUs is $(1822 + 2 \times 570)/(2007 + 6 \times 570) = 0.546$ but, on the other hand, also $C_{SW\text{-}pipelined}/C_{non\text{-}pipelined} > 0.5$ due to the constant terms -1 and $+1$ in (130) and (131). The area-efficiency is illustrated in Fig. 55(b). The curves intersect with unpractical list length $2^{17} - 1$. Thus, the SFU for software pipelined execution is not as area-efficient as the SFU for non-pipelined execution but the difference in area-efficiency is relatively low.

The main target in the designing of the SFUs and assembly routines was to minimize the constant and coefficient terms of (130) and (131). The obtained results show that the additional overhead is kept at almost minimum. In conclusion, the computations are not the main bottleneck but capability for even more parallel memory accesses limits practical throughput.

8.3 Register-Based List Updating for List Sphere Decoders

The list can be stored also in registers. The main benefit of using registers is their parallel accessibility. The parallelism of memory accesses is limited by the number of memory banks, number of ports of the memory, and by the access sequence. When compared to memories, the main drawback of using the registers is their higher power consumption. For these reasons, the register based list processing is tempting with short list lengths. An obvious register-based list processing method uses an insertion sort which has been applied, e.g., in [15]. Sorting with registers is used also in K-best LSD hardware implementation in [130]. However, just like in the case of heap data structure also register based methods can benefit from the fact that the data does not need to be sorted but it is sufficient to find the maximum element. This practice is applied in the register based structures in Sections 8.3.2 and 8.3.3.

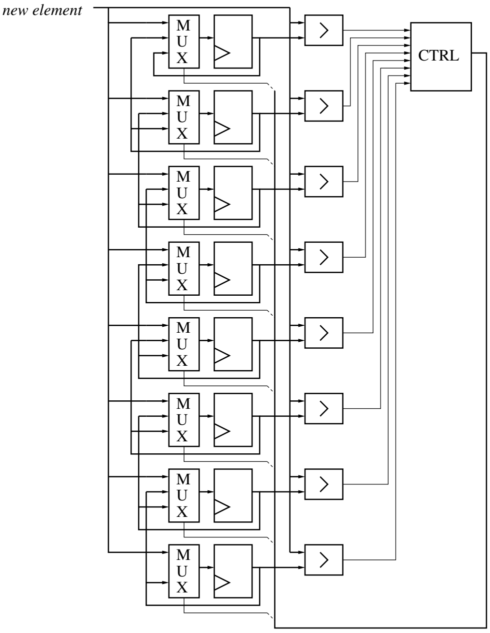


Fig. 56. *Register based insertion sort SFU. The registers contain the sorted list after the insertion operation.*

8.3.1 Insertion Sort SFU

The proposed insertion sort SFU is applied in the K-best LSD TTA processor in [15]. The list is kept in order all the time. The new candidate element is compared with all the elements of the list in parallel. The comparisons indicate, in which place the new element should be inserted or whether it should be discarded. In other words, the candidate x is compared to successive elements e, g with $e \leq g$. If $e \leq x \leq g$, then the x can be inserted between the e and g to preserve the ordering of the list. With register-based list it is natural to move all the elements from the insertion point to the beginning of the list, which can be implemented with a simple shift register structure, i.e., input and output of consecutive registers are connected and the transfer of the element is enabled conditionally.

In principle, the structure of the SFU can be derived systematically for any list length. For simplicity, but without loss of generality, an example structure of the SFU is presented for a list of eight samples in Fig. 56. The results of the parallel comparisons are connected to the *CTRL* block, which contains simple combinatorial logic to gen-

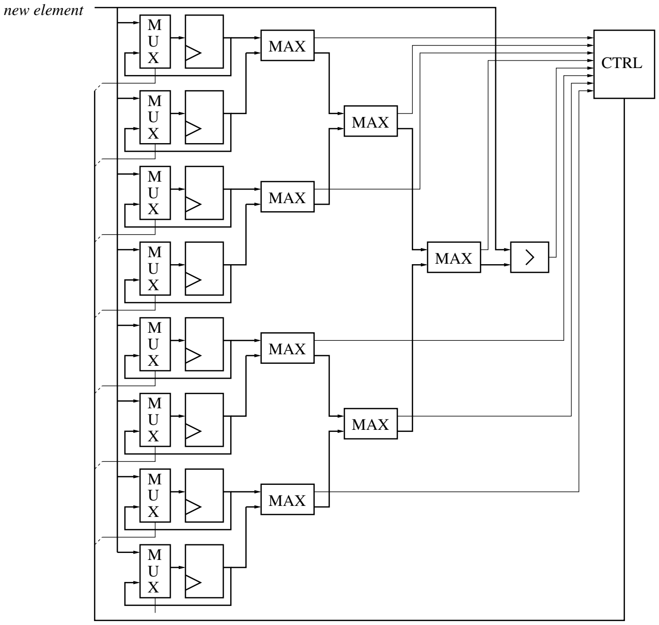


Fig. 57. List processing SFU applying a binary tree of comparisons to find the maximum value.

erate the control signals for the multiplexers, which drive the inputs of registers. The list can be read from the registers, i.e., explicit output signals are not shown in the diagram. Naturally, the comparisons in Fig. 56 use only the PED part of the word in register, the remaining part of the word presents symbol information.

8.3.2 Comparisons with Binary Tree

The insertion with insertion sort SFU may require changing the state of all the registers at maximum. Such a transferring of electric charge between registers consumes energy and, therefore, it would be advantageous if the states of registers remained unchanged most of the time. Even if the sorting without moving samples were impossible it is possible to focus only in finding the maximum element. Like the (124) shows, it is sufficient if the new candidate is compared with the maximum of the list and the maximum is replaced conditionally. In other words, the sorting is not mandatory even if it helps finding the maximum.

The second alternative register-based list processing SFU keeps the list unordered

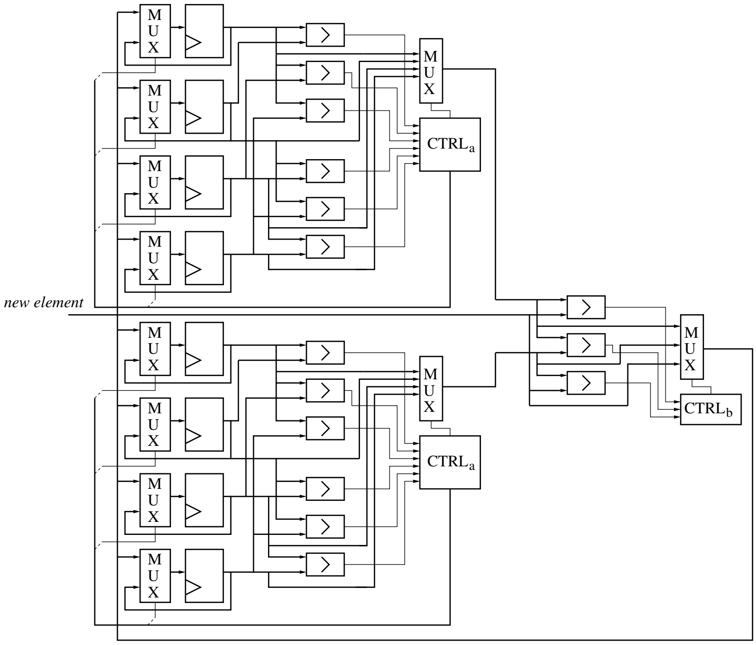


Fig. 58. *List processing SFU applying more parallelism in comparisons.*

and uses a binary tree of maximum operations to find the element with maximum value. The structure of the unit for a list of eight elements is shown in Fig. 57. The diagram shows that with list length, n, the depth of the maximum tree is $\log_2 n$ and in addition to the maximum operations the tree is extended with one comparison operation, which compares the new candidate with the maximum of the list. Again, the *CTRL* block contains combinatorial logic to control the multiplexers, which either preserve the state of the register or replace the content of the register with the new candidate element. The Fig. 57 shows clearly that the main drawback of the maximum tree structure is a long critical path when compared to the insertion sort unit in Fig. 56. The length of the critical path is proportional to the depth of the tree, i.e., $\log_2 n$.

8.3.3 Comparisons with High Parallelism

The third alternative register-based list processing SFU strives to shorten the critical path by applying higher parallelism. In other words, there is a trade-off between the number of parallel computing resources and the critical path, and the critical path is

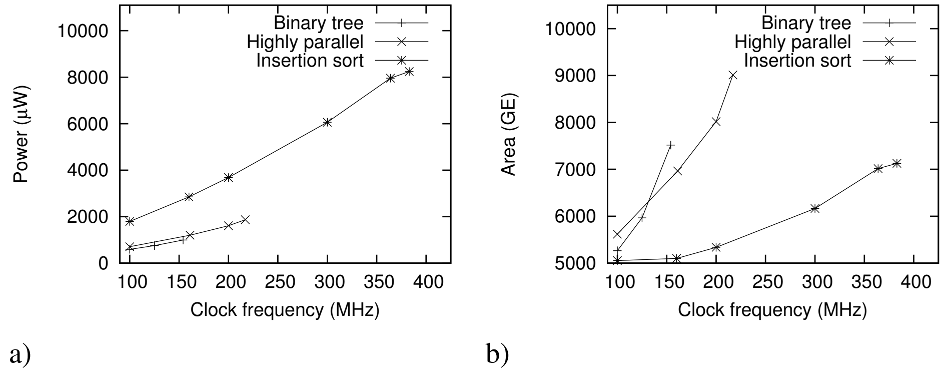


Fig. 59. *Comparison of register based list units with 16-length list: a) power consumption, b) area in terms of logic GEs.*

shortened by using more resources. The proposed structure is shown in Fig. 58. The structure is shown for a list of eight samples. The operations in the first stage find the maximum of every set of four elements with six comparisons per set. For example, with 16-length list the second stage must determine the maximum of five samples since also the new candidate is included in the parallel comparison. Naturally, the critical path is shortened but the number of comparators is increased with the structure in Fig. 58 when compared to Fig. 57.

8.3.4 Power and Complexity Estimates

The alternative register-based list processing methods are compared in terms of power consumption and complexity. As all the methods process the new candidate in one clock cycle the throughput is directly proportional to the clock frequency. For this reason, the complexity estimates as functions of clock frequency are comparable. All the three methods are synthesized with the same operating conditions, 130 nm technology, 1.35 V voltage, clock gating is enabled in the synthesis, and the list length is 16 samples. The word length of the PEDs which is used in comparison units is 16 bits, and the word length of the symbol information is 16 bits.

The power estimation results are shown in Fig. 59(a). The results show that the insertion sort unit has the highest power consumption and it can achieve the highest throughput. The figure shows also that due to the long critical path the structure with the binary tree of maximum operations cannot achieve high clock frequency. The third remark is that the power consumption of the structure with highly parallel comparisons achieves almost as low level as the structure with binary tree comparisons network. This indicates that the major part of the power consumption originates from the registers, as both designs use registers in a similar way, i.e., the ordering of the elements of the list is the same.

The complexity estimation results are shown in Fig. 59(b) in terms of logic GEs. As expected, the insertion sort unit has the lowest complexity. Again, when compared with the power estimation in Fig. 59(a), it can be noticed that the large area does not correspond heavily with high power consumption in this particular case. According to this notion, proper decision can be made when the list processing unit is designed, if the low power is more important than small area.

9. COMPLEXITY AND POWER ESTIMATIONS OF BASEBAND PROCESSING WITH ASPS

In this Chapter, 100 Mbps data rate is targeted and ASP implementations of the four essential baseband functions of the 3G LTE receiver, namely, list sphere decoding, FFT, QR decomposition, and turbo decoding are analyzed. As a result, the design space that describes the essential computational challenges of 3G LTE receivers is clarified and estimates of area, power, and IPC requirements are presented.

On the contrary to focusing on the implementation of solely one function, even a couple of inter-operating functions complicate the design. For example, the number of clock domains and the most suitable clock frequencies must be determined for all the functions. In addition, there is always a trade-off between area and throughput. Furthermore, even if the throughput is adequate, the delay can be too long. Thus, the dimensions of the design space include clock frequency, area, power, parallelism, number of processors, clock domains etc. To find answers to the multivariable and multiobjective design problems, the design space must be explored by focusing on promising candidates, i.e., design alternatives, and analyzing them. Naturally, such analysis is far away from evaluation of a fully functional system-on-chip (SoC) but it provides inevitable insight into the design problem in hand.

In this Chapter, TTA processors are applied to the 3G LTE baseband processing. Baseband functions are separated from system level operations as the area and power analysis focuses on the core computations. The assisting IPC is analyzed in terms of data buffer requirements of ideal IPC links. The presented work forecasts how demanding the implementation of these baseband functions of the 3G LTE receiver would be, and what would be the number of logic GEs, power, number of processors, and IPC requirements with realistic clock frequencies. The results also show how strongly an efficient symbol detection method dominates the total complexity.

9.1 System Model

The high-level description of the targeted two antenna MIMO–OFDM receiver is presented in Fig. 2 in Chapter 2. Naturally, for the turbo decoding and QR decomposition the processors presented in Chapter 5 and in Chapter 7 are applied. The TTA processors presented in [87] and [15] are used for the FFT and symbol detection tasks, respectively.

The applied FFT TTA processor [87] implements mixed-radix FFT consisting of radix-2 and radix-4 computations and it supports several power-of-two transform sizes. It has 11 RFs containing 25 general-purpose registers and three Boolean registers, 17 buses in the interconnect network, a conventional adder, a comparison unit, and two LSUs. The main computations are carried out with the complex-valued adder, complex-valued multiplier, address generator, and coefficient generator SFUs. The processor applies a complex-valued number presentation where the real and imaginary parts both take 16 bits. Data is stored in single-port memory banks and the kernel loop applies the principles of software pipelining. Code compression is applied to enhance the code density and lower the power consumption.

The applied LSD TTA processor [15] generates a 16-element list of candidate solutions to approximate the symbol vector s' in (4). The processor uses 16-bit arithmetic and it is targeted for 2×2 antennas and 64-quadrature amplitude modulation (QAM). Instead of 2×2 complex-valued matrix, a real-valued matrix with doubled dimensions is processed. Therefore, a real-valued 4×4 QR decomposition is required for the LSD. The arithmetic operations are computed with two addition units, a subtraction unit, a multiplier, and a squaring unit. The following SFUs are targeted for the applied K-best algorithm: insertion sorter, PED extractor, storage format composer, and a unit combining a multiplexer and LUT for format conversions. There are three RFs of sizes 16, 10, and 4 registers. On the contrary to conventional processors, the LSD TTA processor does not have LSUs nor data memory, since there is no need for accessing large arrays. The input data is passed via two RFs and the results of the computations are available in the registers of the insertion sorter SFU.

9.2 Processing Requirements and Complexity

The number of processors, their total area and memory requirements, and inter-processor communication requirements are derived from the targeted 100 Mbps throughput.

9.2.1 Time and Throughput Requirements

There are seven OFDM symbols per transmit antenna in 0.5 ms time frame in 3G LTE downlink [16]. Thus, the processing time requirement $T_{\text{FFT}} = 0.5$ ms$/7 = 71$ μs includes also the additional time contributed by the cyclic prefix of the OFDM symbol. The FFT must be computed for both antennas.

The QR decomposition must be processed in the coherence time, t_{coh}, of the channel. If bullet train speed $v_r = 500$ km/h is assumed for the receiver, the coherence time is $t_{coh} = c/(f_{carrier}v_r) = 0.9$ ms where c is the speed of light and $f_{carrier} = 2.4$ GHz is the carrier frequency. However, with a more rapidly varying channel, the QR decomposition must be computed more frequently, i.e., shorter t_{coh} must be used in (133). A single QR decomposition combines information from all the antennas. In other words, the matrix and vector sizes of the QR decomposition depend on the number of antennas.

The LSD must be computed for each subcarrier. So, the time requirement equals to the time requirement of the FFT. However, even if the maximum length of the FFT is 2048, only 1201 subcarriers are in use. A single LSD processes the signals of both antennas, i.e., it outputs estimates of symbols transmitted from both antennas.

Since the turbo decoder processes soft bits instead of QAM symbols, it is meaningful to express throughput as data rate. The throughput requirement of turbo decoding equals the maximum data rate of 100 Mbps. Naturally, with code rate $R = 1/2$ and 64-QAM symbols, the data rate on the LSD side is 200 Mbps and symbol rate 33.3 Msps.

9.2.2 Required Number of Clock Cycles

The FFT TTA processor in [87] takes 12332 clock cycles for the 2048-point transform and the transform must be computed for both antennas. So, the required clock cycles of the FFT task are

$$C_{\mathrm{FFT}} = 2 \times 12332 = 24664\,. \tag{132}$$

The QR decomposition algorithm is of order $O(n^3)$ and the QR decomposition TTA processor in Chapter 7 takes 139 clock cycles for a 4×4 matrix. The dimensions of the decomposed matrix are doubled, since the LSD TTA processor applies real-valued computation. Since the $\boldsymbol{Q}$ matrix is the argument of matrix-vector product in (4), the products are mapped to the same processor. The products must be computed continuously for each received symbol vector, but the QR decomposition only once in the coherence time. So, the average number of clock cycles in T_{FFT} time period, for both computations is approximately

$$C_{\mathrm{QR\_avg}} = 1201 \times (139 \times (T_{\mathrm{FFT}}/t_{coh}) + 16) = 32386 \tag{133}$$

where 4×4 matrix multiplication takes 16 clock cycles. Naturally, with more rapidly varying channel, the C_{QR_avg} increases as the t_{coh} must be decreased. The products take approximately 59% of the $C_{\mathrm{QR\_avg}}$. The maximum number of clock cycles is spent when the decomposition of a new channel matrix is computed for each subcarrier, i.e.,

$$C_{\mathrm{QR}} = 1201 \times (139 + 16) = 186155\,. \tag{134}$$

The average number of clock cycles, $C_{\mathrm{QR\_avg}}$, is only 17% of the maximum, C_{QR}.

The LSD TTA processor in [15] takes 441 clock cycles for processing one symbol vector. Thus, in T_{FFT} time period the number of required clock cycles for the LSD, C_{LSD}, is approximately

$$C_{\mathrm{LSD}} = 1201 \times 441 = 529641\,. \tag{135}$$

Fortunately, the LSD can be parallelized among the subcarriers.

In order to compare turbo decoding with the other baseband functions, the clock cycles of turbo decoding must be normalized to clock cycles, C_{Turbo}, taken in T_{FFT} time frame. The turbo decoder TTA processor in Chapter 5 takes 1.016 clock cycles per trellis stage processed in half iteration. With six iterations each trellis stage is processed 12 times. Therefore,

$$C_{\mathrm{Turbo}} = T_{\mathrm{FFT}} \times 100 \times 10^6 \times 12 \times 1.016 = 86563 \tag{136}$$

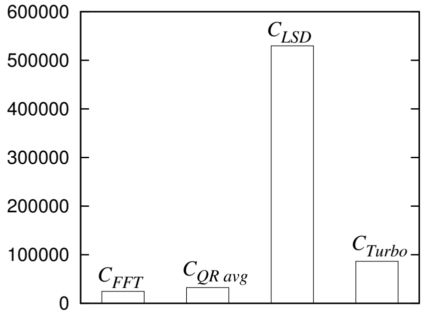


Fig. 60. *Required number of clock cycles for processing the tasks in $T_{\mathrm{FFT}} = 71\ \mu s$ time frame.*

where the first multiplications $T_{\mathrm{FFT}} \times 100 \times 10^6$ express how many bits are processed in T_{FFT}. Turbo decoding can be parallelized to several processors with block-by-block pipelining where each processor decodes a code block of its own independently.

The required number of clock cycles of all the four functions are illustrated in Fig. 60. The figure shows clearly how the LSD dominates the computation load. Obviously, the requirements cannot be met with single processor systems with currently achievable clock frequencies.

9.2.3 Number of Processors

The required minimum number of processors is determined by the throughput per processor, clock frequency, f_i, and parallelization scheme of the targeted functions. If a task i can be parallelized to several processors and the throughput is directly proportional to the number of processors, then the minimum required number of processors, P_i, of the task i taking C_i clock cycles in time frame T_{FFT} is

$$P_i = \lceil (C_i / T_{\mathrm{FFT}}) / f_i \rceil \ . \tag{137}$$

The utilization, U_i, of P_i processors dedicated to task i tells how efficiently the computing resources are used. It can be defined in a similar way as

$$U_i = C_i / (P_i T_{\mathrm{FFT}} f_i) \ . \tag{138}$$

Naturally, $100 \times (1 - U_i)$ tells how many percent of the time the P_i processors idle. For the QR decomposition and matrix-vector product task, the average number of clock cycles, $C_{\mathrm{QR\_avg}}$, is used to calculate the minimum number of processors and utilization. The total utilization of the whole processing chain can be computed as

$$U = \sum_{i \in S_{\mathrm{tasks}}} C_i / (T_{\mathrm{FFT}} \sum_{i \in S_{\mathrm{tasks}}} P_i f_i) \tag{139}$$

where the sums are computed for all the elements of the task set S_{tasks} $=\{$FFT, QR_avg, LSD, Turbo$\}$. The total utilization in (139) expresses the ratio between the required execution cycles of all the tasks and the available execution cycles of all the processors.

9.2.4 Delay

The delay of a task depends on the maximum size of the processed data vector and the scheduling. Except for the first half iteration, the turbo decoder requires that the whole code block is received before decoding. The maximum code block length is 6144 [3], which is about 20% longer than in the current 3G systems. With code rate $R = 1/2$, the required number of soft-bits is naturally $2 \times 6144 = 12288$. For two OFDM symbols, the LSD generates symbol candidate lists, which can be converted to $2 \times 1201 \times 6 = 14412$ soft bit estimates with 64-QAM (6 bits per symbol). Since the number of soft-bits exceeds the required number for the maximum code block length, the analysis of the delay of FFT and LSD can be limited to the processing of two OFDM symbols.

With at maximum two processors, the delay of the FFT is simply

$$D_{\mathrm{FFT}} = C_{\mathrm{FFT}} / (P_{\mathrm{FFT}} f_{\mathrm{FFT}}), \quad P_{\mathrm{FFT}} \in \{1, 2\} \tag{140}$$

and in a similar way the delay of the LSD is

$$D_{\mathrm{LSD}} = C_{\mathrm{LSD}} / (P_{\mathrm{LSD}} f_{\mathrm{LSD}}) \tag{141}$$

where $P_{\mathrm{LSD}} \in \{1, 2, \ldots, 1201\}$ as the LSD can be parallelized among the subcarriers. The QR decomposition processor has two tasks, the QR decomposition and the matrix-vector products, of which the QR decomposition is computed only once in the coherence time, $t_{coh} = 0.9$ ms. Thus, the worst case delay when both tasks are computed is

$$D_{\mathrm{QR}} = C_{\mathrm{QR}} / (P_{\mathrm{QR}} f_{\mathrm{QR}}) \tag{142}$$

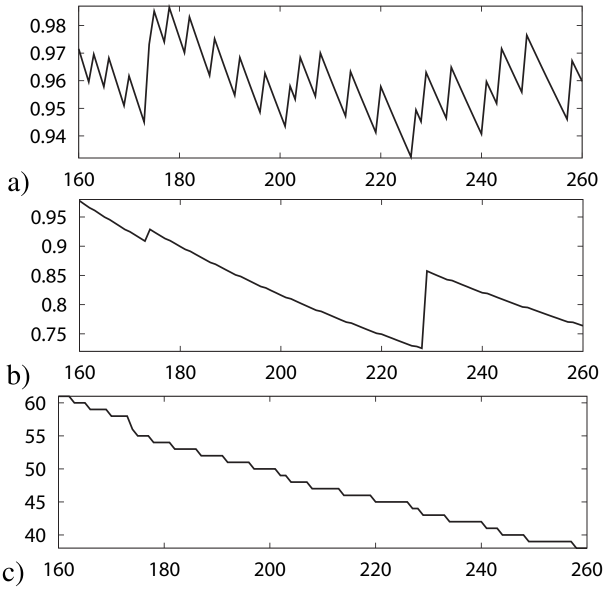


Fig. 61. *Configurations as functions of f_i with single clock domain: a) total utilization, b) total delay in ms, c) the number of processors. The x-axis denotes f_i in MHz.*

where $P_{\mathrm{QR}} \in \{1, 2, \ldots, 1201\}$ as the decompositions and multiplications can be parallelized among the subcarriers. For an average delay, $C_{\mathrm{QR\_avg}}$ can be used in a similar way. The delay of turbo decoding is determined by the maximum code block size, 6144. Thus, the delay with six turbo iterations is

$$D_{\mathrm{Turbo}} = 6144 \times 6 \times 2 \times 1.016/f_{\mathrm{Turbo}} \tag{143}$$

where processing one trellis stage with the turbo decoder TTA processor takes on average 1.016 clock cycles. Distributing the turbo decoding to several processors with block-by-block pipelining would affect only the throughput but not the delay and, therefore, the number of processors is omitted from (143).

9.2.5 ASP Configurations as Functions of Clock Frequency

Utilization, delay, and number of processors are analyzed in Fig. 61 as functions of clock frequency, f_i. The total utilization in Fig. 61(a) shows that the utilization is always greater than 0.93 in the explored clock frequency range. High utilization can be obtained easily, since the LSD dominates the computational load and it can be parallelized with very fine granularity. In other words, since the utilization of the LSD task is always high, also the utilization of the whole processing chain is relatively high. The peaks in the utilization occur, when the number of processors of some task can be decremented. In that case, the utilization grows. On the contrary, if the number of processors remains untouched and the clock frequency is increased the utilization decreases. The discontinuations of delay in Fig. 61(b) originate from the same phenomenon. The greatest discontinuation at 229 MHz takes place as the QR decomposition is mapped from three to two processors. The number of processors in Fig. 61(c) decreases quite steadily, since it is dominated by the LSD task, which requires the largest number of processors.

9.2.6 Analysis

An example configuration of TTA processor based baseband processing chain is presented in Table 16. A single clock domain with $f_i = 250$ MHz is applied and the processors have been synthesized with 130 nm technology for obtaining complexity and power estimates. Even if higher clock frequencies were used in previous Chapters for achieving high throughput, a common clock domain is assumed for eased system integration. The area and power estimates exclude the memories. The power estimates are scaled with the number of respective processors and their utilization in the eighth row of the Table 16. The results in Table 16 show that since the LSD task takes only 441 clock cycles per subcarrier and it can be computed for each subcarrier independently, the task can be easily divided among several processors to achieve a high utilization. On the contrary, it is more difficult to obtain very high utilization for both the FFT and the QR processors with the same clock frequency, as the granularity of the tasks is more coarse. As a second remark, the delay of the QR decomposition is long when compared to other functions, even though the other functions are more complex. However, the QR decomposition must be computed only once in the coherence time $t_{coh} = 0.9$ ms, i.e., the delay in Table 16 is the worst case delay. On average, the delay of the QR decomposition and the matrix-vector products is only 17% of the delay in Table 16.

In principle, the FFT and QR tasks could be mapped to the same processor. The processor should be formed as a hybrid of both processors in this case. The resources could be combined with the aid of multiset unions, i.e., the maximum multiplicity

Table 16. *The baseband processing chain with TTA processors,* 2×2 *antennas, 1201 subcarriers, 64-QAM, 6144-length turbo code block, list length* $n = 16$*, data rate 100 Mbps.*

	FFT	Turbo dec.	QR & prod.	LSD	Total
Clk. freq., f_i (MHz)	250	250	250	250	
Num. procs., P_i	2	5	2	30	39
Utilization, U_i	0.69	0.98	0.91	0.99	0.97
Delay, D_i (ms)	0.049	0.300	0.372	0.071	0.792
Area (kGE)	30.5	35.1	17.7	23.6	
Area $\times P_i$	61.0	175.5	35.4	708.0	979.9
Power est. (mW)	36.3	50.8	13.1	20.8	
Power est. $\times P_i \times U_i$	50.1	248.9	23.8	617.8	940.6
Technology (nm)	130	130	130	130	
Reference	[87]	Chapter 5	Chapter 7	[15]	

of resources of both processors determines how many resources of respective type are instantiated [92]. Since both functions require complex arithmetic, the same resources could be shared efficiently. With $f_i = 402$ MHz, both tasks could be mapped to two hybrid FFT/QR TTA processors and a utilization, $U_{\text{FFT/QR}} = 1.00$, would be obtained.

Mapping the turbo decoding and some other function to the same processor could not benefit as much from sharing the resources, since the turbo decoding requires mostly real-valued ACS operations. Shortening the delay of the turbo decoding is difficult for two reasons. Firstly, turbo decoding is an iterative process where the previous iteration must be finished before the next one can begin. Secondly, the component decoder applying the radix-2 algorithm processes at maximum one trellis stage in one clock cycle. The next path metrics cannot be computed according to (7) and (8) before the previous ones are computed. For these reasons, increasing the clock frequency or applying the radix-4 algorithm are the only ways to shorten the delay of the turbo decoding task in Table 16.

To illustrate more deeply the computational requirements of the baseband processing, example configurations consisting of other implementations are shown in Tables 17–19. As the respective implementations in Tables 17–19 are not necessarily targeted to the 3G LTE system or they are not targeted to operate among each other,

Table 17. *An example baseband processing chain with* 2×2 *antennas, 1201 subcarriers, 16-QAM, 4804-length turbo code block, data rate 68 Mbps.*

	FFT & Turbo dec.	QR	Sphere Dec.	Total
Clk. freq., f_i (MHz)	600 & 300	223	213	
Num. procs., P_i	5	1	1	7
Utilization, U_i	0.36	0.29	0.92	0.38
Delay, D_i (ms)	0.396	0.259	0.065	0.720
Area (kGE)	–	198	61	
Area $\times P_i$	–	198	61	–
Power est. (mW)	718	–	–	
Power est. $\times P_i \times U_i$	1303	–	–	–
Technology (nm)	130	130	130	
Reference	[4]	[26]	[26]	

Table 18. *An example baseband processing chain with* 4×4 *antennas, 601 subcarriers, 16-QAM, 4808-length turbo code block, list length* $n = 10$*, data rate 68 Mbps.*

	FFT	Turbo dec.	QR	LSD	Total
Clk. freq., f_i (MHz)	45	400	80	50	
Num. procs., P_i	2	5	1	2	10
Utilization, U_i	0.63	0.82	0.54	0.85	0.80
Delay, D_i (ms)	0.045	0.288	0.489	0.060	0.882
Area (kGE)	–	64.1	–	132	
Area $\times P_i$	–	320.5	–	264	–
Power est. (mW)	480	–	–	–	
Power est. $\times P_i \times U_i$	608.45	–	–	–	–
Technology (nm)	350	65	250	130	
Reference	[67]	[124]	[84]	[130]	

the Tables 16–19 should be not considered as comparisons of TTA processors and other implementations. Instead the tables show indicative example configurations of baseband processing chains.

For some implementations, all the required information is not available or it is given with different units. The area is reported if it has been given as GEs. For some implementations the performance data is not available for the targeted configuration

Table 19. Requirements of 4G baseband processing chain for 100 Mbps data rate [133].

	FFT	STBC	LDPC	Total
Assumed clk. freq. (MHz)	360	240	385	
Assumed num. procs., P_i	1	1	20	22
MCycles/s	360	240	7700	8300
Utilization, U_i	1.00	1.00	1.00	1.00
Reference	[133]	[133]	[133]	

of 2048-length FFT, 2 × 2 antennas, 64-QAM, and list length 16. For this reason, alternative MIMO–OFDM configurations with lower data rate, 68 Mbps, have been used. Shorter code blocks are assumed for turbo coding in Tables 17 and 18. With shorter code blocks, the delay of the FFT can be limited to processing one OFDM symbol per each antenna.

In the configuration in Table 17, hardware implementations presented in [26] are used for the matrix decomposition and symbol detection. For the FFT and turbo decoding the TI's C6416 DSP has been applied as it can compute the FFT with an efficient software library routine and it includes a turbo co-processor which runs with halved clock frequency. Since the core DSP and turbo co-processor are mapped to the same device, the number of required processors is determined by the more dominating task, i.e., turbo decoding. The idling of the DSP core while turbo decoding is taken into account when the utilization in Table 17 is calculated and, therefore, the utilization is low in Table 17 but still several processors are required. The hardware implementations for QR and symbol detection in Table 17 are targeted for MIMO–OFDM systems [26]. However, the sphere detector applies a different algorithm than the K-best LSD which is used in TTA processor implementations.

In Table 18, a 1024-point FFT is applied. The applied turbo decoder processor supports also Viterbi decoding. The list length of the K-best LSD is 10 elements. In principle, a complex-valued K-best LSD with 64-QAM, 2 antennas, and list length $n = 16$ must process $64 + 16 \times 64 = 1088$ elements and with 16-QAM, 4 antennas, and $n = 10$ it must process $16 + 10 \times 16 + 10 \times 16 + 10 \times 16 = 496$ elements during the symbol detection. Thus, the processing requirements of different symbol detectors can be characterized by the number of visited elements during a tree traversal of the LSD algorithm. The applied QR decomposition hardware accelerator is presented in [84] as a part of MIMO–OFDM transceiver for WLANs. The decomposition takes

Table 20. *Area of the core processor without memories and data memory requirements of the processors.*

TTA processor	Clk. freq. f_i (MHz)	Area (kGE)	Data memory requirements (kbits)
FFT	250	30.5	65.5 divided into 2 single-port memory banks
QR	250	17.7	1.5 dual-port memory
LSD	250	23.6	0.0 (uses only registers)
Turbo decoder	250	35.1	281.7 divided into 16 single-port memory banks

65 clock cycles for 4×4 matrix.

In Table 19, the workload of 4G baseband processing with 100 Mbps is presented in terms of required execution cycles on a SODA architecture [133]. For each task a realistic clock frequency is assumed and the tasks are divided to separate processors. Furthermore, it is assumed that the low-density parity-check (LDPC) error correction decoding task can be parallelized to several processors. The Table 19 shows that the LDPC task dominates clearly the workload.

In conclusion, the results in Tables 16–19 show that in addition to the data rate, the computational requirements depend heavily on the applied algorithms and on the parameters of the algorithms. Furthermore, efficiency in terms of high utilization requires that the tasks can be mapped among the processors or hardware units in a flexible way.

9.2.7 Memory Requirements

The area estimates in Table 16 exclude the memories and memory requirements are reported separately in Table 20. In other words, the area in terms of logic GEs expresses the complexity of the actual computations of baseband processing. The separation eases future comparisons, since the memory requirements depend heavily on the targeted data vector lengths and technology. For example, long code blocks are preferred in turbo decoding, as they enhance the error correction performance. A second reason for separating the memories is that the IPC requires also memories

Table 21. Additional buffer memory requirements for seamless IPC.

IPC buffer	Memory (words)	Memory (kbits)
FFT: next input	2×2048	131.1
FFT: prev. result	2×2048	131.1
QR: $\boldsymbol{R}$, $\boldsymbol{Q}^H \boldsymbol{y}$	$1201 \times (10 + 4)$	538.0
QR: prev. results	$1201 \times (10 + 4)$	538.0
Turbo: next input	3×6114	128.4

and, therefore, the total area with all the memories of the whole baseband processing chain would depend on the implementation method of the IPC.

The data memory requirements in Table 20 show that due to the small matrix size, the QR decomposition requires a very small memory. The LSD processor has no memory requirements at all, as the data is stored in registers. On the other hand, the turbo decoder and the FFT processors require large memories as they have to process long data vectors. The memory of the FFT is divided into two banks and a memory interface hides the banking structure from the programmer, i.e., the memory system imitates dual-port memory.

9.2.8 Inter-Processor Communication Requirements

As the analyzed processors lack extra facilities for IPC, only requirements but not costs can be stated. There exists many methods for SoCs but they are beyond the scope of this Thesis. Therefore, the effects of using some particular method or SoC platform are not considered. In Table 21, the IPC requirements are tabulated for an assumed system using shared memory banks between the processors.

The FFT processor uses an in-place algorithm, i.e., the result overwrites the input vector and processing does not require additional memory. However, passing the data to and from the FFT processors requires buffer memories. In practice, there must be an extra input buffer which is written while the data in the main memory is processed in-place. In a similar way, there must be an extra output buffer, from which the previous result can be read at the same time. The first two buffers in Table 21 are dedicated for such an IPC. The roles of the three memory banks, i.e., the input buffer, the output buffer, and the processing memory, can be interchanged on every two completed OFDM symbols.

The QR processor generates the triangular 4×4 matrix, $\boldsymbol{R}$, with 10 nonzero elements and 4-element vector for each subcarrier. The results are written to one buffer. The other identical buffer holds the previous results which are passed to the LSD processors at the same time. Since there are several QR and LSD processors, the buffer must be divided into several parallel accessible banks. Again, the roles of the buffers can be interchanged on OFDM symbol boundaries.

The turbo decoder processors require an additional input buffer which is filled with the soft-bits while the decoders are processing. There is no need for an additional output buffer, since the decoder overwrites the previous output only on the last half iteration. The buffer size of the turbo decoder input in Table 21 allows code rate $R = 1/3$ with the maximum block size. The input word length of the applied turbo decoder TTA processor is 7 bits but all the other applied TTA processors use 16 bits for the real or imaginary parts.

In general, the complexity of IPC buffers depend on the sizes of memory banks, their throughput or clock frequency, and the number of memory banks as each bank requires interfacing logic. In addition, the IPC increases also the computational load which is not included in Tables 16–19. Therefore, if a fully functional SoC were designed, full utilization should not be targeted when solely the core computations are analyzed. Instead, with lower utilization, computing capacity would be reserved also for the IPC. Also, the delays in Tables 16–19 exclude the effect of IPC. As it is assumed that one buffer is written while the other is read in a pipelined fashion, it can be assumed that the IPC has a constant delay.

Since the workloads of the processors depend only on the applied block lengths, static scheduling could be applied, which would ease synchronization of the tasks. Even if the number of processors is very high, in principle, similar IPC requirements would be met also with smaller number of processors if they applied higher parallelism internally or if they applied higher clock frequency. The first option would require parallel IPC links and the second option would require smaller number of IPC links but higher throughput for each link.

10. CONCLUSIONS

In this Thesis, several implementations for baseband processing of digital receivers were presented. ASPs were the primary target platform, and both assisting hardware units and processor architectures were proposed. The main design goal throughout the design process was the efficiency, in terms of computations per clock cycle with the available computing resources. With this goal, the performance of the implementations was not degraded by the inherent sequentiality of the programmable processor applications. Instead, with sufficiently parallel architectures, the kernel computations could be executed continuously with software pipelined schedules. With this practice of high parallelism and high utilization of the main computing resources, there was no performance degradation due to the programmability. Applying the principles of software pipelining allowed high utilization. To allow continuous loading of operands and storing of results, assisting functions for parallel memory accesses were proposed. Other demanding assisting functions were the approximation of the highly non-linear inverse square root function and the list processing for the LSD. The presentation of the proposed implementations was summarized by estimating the complexity and power consumption of a system with 100 Mbps data rate. In conclusion, the baseband functions can be implemented efficiently with ASPs, as long as sufficient parallelism, flexibility, and customizability are provided by the applied ASP template.

10.1 Possible Future Trends of Baseband Processing Implementations

Alan Kay, a visionary inventor at Xerox PARC, has said, *“The best way to predict the future is to invent it”* [59]. Based on this quote, the best future prediction would rely on the proposed implementations and it would state that in the future there will be baseband processing chains consisting solely of programmable ASPs. Such an implementation would make up an ideal SDR if it could adapt to different transmission techniques by re-programming [75,76]. It can be also predicted that as new standards and transmission techniques are developed, there will be always some gap between the computing requirements and the performance that can be obtained with a processor programmed using a high-level language. In other words, the SDR cannot be programmed only with a high-level language but assembly programming similar to what was applied in this Thesis will be still needed, but only in a minor role.

Another prediction is that the role of electronic system level design tools will increase and future baseband processing chains are designed on a high abstraction level. This conforms with the proposed ASP implementations if they were programmed solely with a high-level language. In that case, the applied framework of ASPs would provide a design flow from a description written in a high-level language to a hardware implementation.

The key benefit of the SDR is the ability to adapt to several transmission techniques. This is required currently and in the near future as the new communication devices must conform also to previous standards. The adaptability is also useful since functions like, e.g., Viterbi decoding and turbo decoding are not run at the same time. In the remote future, the adaptability can be required by cognitive radio systems [77]. The cognitive radio changes its parameters efficiently according to the prevailing conditions of the channel, other users of the channel, and the data rate requirements. In this Thesis, the SFUs and ASPs were synthesized with an ASIC technology. However, modern FPGA devices also provide a lot of computing capacity. Instead of adaptability with programmable SDRs, it might be also possible to obtain the adaptability by reconfiguration of FPGAs. As a drawback, the FPGAs suffer from a high power consumption currently and, therefore, ASIC technology is preferred when intensive computing like baseband processing is required.

It can be assumed fairly that with a reconfigurable implementation, higher performance can be obtained since the whole architecture can be changed instead of changing only the program code. This practice does not exclude programmable implementations. Even if the programs were bound with their respective architectures, the programming is a more flexible and more rapid design method than pure hardware design with low-level hardware description languages. However, regardless of the aforementioned benefits, the reconfigurable baseband processing with FPGA devices cannot be established in the targeted telecommunication systems unless their power consumption achieves lower levels.

BIBLIOGRAPHY

[1] 3GPP, “Universal mobile telecommunications system; requirements for the UMTS terrestrial radio access system (UTRA) (UMTS 21.01 version 3.0.1),” Oct. 1997.

[2] ——, “3GPP TS 25.212 v4.3.0; universal mobile telecommunications system (UMTS); multiplexing and channel coding (FDD),” Dec. 2001.

[3] ——, “3GPP TS 36.212 v1.0.0 3rd generation partnership project; technical specification group radio access network; multiplexing and channel coding (release 8),” Mar. 2007.

[4] S. Agarwala, T. Anderson, A. Hill, M. Ales, R. Damodaran, P. Wiley, S. Mullinnix, J. Leach, L. M. Gill, A. Rajagopal, A. Chachad, M. Agarwala, J. Apostol, M. Krishnan, D. Bui, Q. An, N. Nagaraj, T. Wolf, and T. Elappuparackal, “A 600-MHz VLIW DSP,” *IEEE J. Solid-State Circuits*, vol. 37, no. 11, pp. 1532–1544, Nov. 2002.

[5] I. Ahmed and T. Arslan, “A low energy VLSI design of random block interleaver for 3GPP turbo decoding,” in *Proc. Int. Symp. Circuits and Syst.*, Kos, Greece, May 2006, pp. 285–288.

[6] T. Ahonen and J. Nurmi, “Integration of a NoC-based multimedia processing platform,” in *Proc. Int. Conf. Field Programm. Logic and Applications*, Tampere, Finland, Aug. 2005, pp. 606–611.

[7] I. A. Al-Mohandes and M. I. Elmasry, “Low-energy design of a 3G-compliant turbo decoder,” in *Proc. IEEE Northeast Workshop Circuits Syst.*, Montreal, Canada, June 2005, pp. 153–156.

[8] *Viterbi IP Core Decoder Datasheet*, Aldec, Aug. 2004.

[9] G. Alia and E. Martinelli, “A VLSI modulo m multiplier,” *IEEE Tran. Computers*, vol. 40, no. 7, pp. 873–878, Jul. 1991.

[10] Altera, “Implementation of CORDIC-based QRD-RLS algorithm on Altera Stratix FPGA with embedded Nios soft processor technology,” Altera Corporation, San Jose, CA, USA, white paper, Mar. 2004.

[11] P. Ampadu and K. Kornegay, “An efficient hardware interleaver for 3G turbo decoding,” in *Proc. Radio and Wireless Conf.*, Boston, MA, USA, Aug. 2003, pp. 199–201.

[12] J. B. Anderson and S. Mohan, *Source and Channel Coding: An Algorithmic Approach.* Boston, MA, USA: Kluwer, 1991.

[13] R. Andraka, “A survey of cordic algorithms for FPGA based computers,” in *Proc. Sixth ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Monterey, CA, USA, Feb. 1998, pp. 191–200.

[14] J. Antikainen, P. Salmela, O. Silven, M. Juntti, J. Takala, and M. Myllylä, “Transport triggered architecture implementation of list sphere detector,” in *Proc. Finnish Sig. Proc. Symp.*, Oulu, Finland, Aug. 2007.

[15] J. Antikainen, P. Salmela, O. Silvén, M. Juntti, J. Takala, and M. Myllylä, “Fine-grained application-specific instruction set processor design for the K-best list sphere detector algorithm,” in *Proc. Int. Conf. Embedded Comput. Syst.*, Samos, Greece, July 2008, pp. 108–115.

[16] R. Bachl, P. Gunreben, S. Das, and S. Tatesh, “The long term evolution towards a new 3GPP* air interface standard,” *Bell Labs Technical J.*, vol. 11, no. 4, pp. 25–51, Mar. 2007.

[17] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, “Optimal decoding of linear codes for minimizing symbol error rate,” *IEEE Trans. Information Theory*, vol. 20, no. 2, pp. 284–287, Mar. 1974.

[18] C. Bai, J. Jiang, and P. Zhang, “Hardware implementation of log-MAP turbo decoder for W-CDMA node B with CRC-aided early stopping,” in *Proc. Vehicular Tech. Conf.*, vol. 2, Birmingham, AL, USA, May 2002, pp. 1016–1019.

[19] C. Benkeser, A. Burg, T. Cupaiuolo, and Q. Huang, "A 58mw 1.2mm^2 HSDPA turbo decoder ASIC in 0.13μm CMOS," in *Digest Tech. Papers of IEEE Int. Solid-State Circ. Conf.*, San Francisco, CA, USA, Feb. 2008, pp. 264–265 and 612.

[20] F. Berens, M. J. Thul, F. Gilber, and N. Wehn, "Electronic device avoiding write access conflicts in interleaving, inparticular optimized concurrent interleaving architecture for high throughput turbo-decoding," European Patent Application EP 1 401 108 A1, Mar., 2004.

[21] C. Berrou, A. Glaviex, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes," in *Proc. IEEE Int. Conf. Commun.*, vol. 2, Geneva, Switzerland, May 1993, pp. 1064–1070.

[22] M. Bickerstaff, L. Davis, C. Thomas, D. Garret, and C. Nicol, "A 24Mb/s radix-4 logMAP turbo decoder for 3GPP-HSDPA mobile wireless," in *Digest Tech. Papers Solid-State Circ. Conf.*, San Francisco, CA, USA, Feb. 2003, pp. 150–151.

[23] M. A. Bickerstaff, D. Garret, T. Prokop, C. Thomas, B. Widdup, G. Zhou, L. M. Davis, G. Woodward, C. Nicol, and R.-H. Yan, "A unified turbo/Viterbi channel decoder for 3GPP mobile wireless in 0.18-μm CMOS," *IEEE J. Solid-State Circ.*, vol. 37, no. 11, pp. 1555–1564, Nov. 2002.

[24] E. Boutillon, C. Douillard, and G. Montorsi, "Iterative decoding of concatenated convolutional codes: Implementation issues," *Proc. IEEE*, vol. 95, no. 6, pp. 1201–1227, June 2007.

[25] C.-C-Lin, Y.-H. Shin, H.-C. Chang, and C.-Y. Lee, "A low power turbo/Viterbi decoder for 3GPP2 applications," *IEEE Trans. on VLSI Syst.*, vol. 14, no. 4, pp. 426–430, Apr. 2006.

[26] B. Cerato, G. Masera, and E. Viterbo, "Enabling VLSI processing blocks for MIMO-OFDM," *VLSI Design*, vol. 2008, p. 10 pages, 2008.

[27] R. W. Chang and R. A. Gibby, "A theoretical study of performance of an orthogonal multiplexing data transmission scheme," *IEEE Trans. Commun. Technol.*, vol. 6, no. 4, pp. 529–540, Aug. 1968.

[28] C.-H. Chen and C.-Y. Lee, "A cost effective lighting processor for 3D graphics application," in *Proc. Int. Conf. on Image Processing*, vol. 2, Kobe, Japan, Oct. 1999, pp. 792–796.

[29] Y.-T. Cheng, *TMS320C6000 Integer Division*, Texas Instruments, Oct. 2000, SPRA707.

[30] D. G. Choi, M.-H. Kim, J. H. Jeong, J. W. Jung, J.-T. Bae, S.-S. Choi, and Y. Yun, "An FPGA implementation of high-speed flexible 27-Mbps 8-state turbo decoder," *ETRI J.*, vol. 29, no. 3, pp. 363–370, June 2007.

[31] J. N. Coleman and E. I. Chester, "A 32 bit logarithmic arithmetic unit and its performance compared to floating-point," in *Proc. 14th IEEE Symp. on Computer Arithmetic*, Adelaide, Australia, Apr. 1999, pp. 142–151.

[32] H. Corporaal, "Design of transport triggered architectures," in *Proc. 4th Great Lakes Symp. Design Autom. High Perf. VLSI Syst.*, Notre Dame, IN, USA, Mar. 1994, pp. 130–135.

[33] H. Corporaal and M. Arnold, "Using transport triggered architectures for embedded processor design," *Integrated Computer-Aided Engineering*, vol. 5, no. 1, pp. 19–38, 1998.

[34] P. Darwood, P. Alexander, and I. Oppermann, "LMMSE chip equalization for 3GPP WCDMA downlink receivers with channel coding," in *Proc. IEEE Int. Conf. on Commun.*, vol. 5, Helsinki, Finland, Jun. 2001, pp. 1421–1425.

[35] F. de Dinechin and A. Tisserand, "Some improvements on multipartite table methods," in *Proc. 15th IEEE Symp. on Computer Arithmetic*, Vail, CO, USA, Jun. 2001, pp. 128–135.

[36] P. Duhamel and M. Vetterli, "Fast Fourier tranforms: A tutorial review and a state of the art," *Signal Processing*, vol. 19, no. 4, pp. 259–299, 1990.

[37] F. Edman and V. Öwall, "A scalable pipelined complex valued matrix inversion architecture," in *Proc. IEEE Int. Symp. Circuits and Syst.*, vol. 5, Kobe, Japan, May 2005, pp. 4489–4492.

[38] T. ElGamal, “A public key cryptosystem and a signature scheme based on discrete logarithms,” *IEEE Trans. Information Theory*, vol. 31, no. 4, pp. 469–472, Jul. 1985.

[39] K. M. Elleithy and M. A. Bayoumi, “A systolic architecture for modulo multiplication,” *IEEE Tran. Circuits and Syst. II: Analog and Digital SP*, vol. 42, no. 11, pp. 725–729, Nov. 1995.

[40] M. D. Ercegovac, T. Lang, J.-M. Muller, and A. Tisserand, “Reciprocation, square root, inverse square root, and some elementary functions using small multipliers,” *IEEE Trans. On Computers*, vol. 49, no. 7, pp. 628–637, Jul. 2000.

[41] J. A. Erfanian, S. Pasupathy, and G. Gulak, “Reduced complexity symbol detectors with parallel structures,” in *Proc. Global Telecommm. Conf.*, San Diego, CA, USA, Dec. 1990, pp. 704–708.

[42] G. H. Golub, *Matrix Computations.* Baltimore, MD, USA: John Hopkins University Press, 1989.

[43] J. Guo, K. Dai, and Z. Wang, “A heterogeneous multi-core processor architecture for high performance computing,” in *Advances in Comp. Syst. Architecture*, vol. LNCS 4186. Berlin, Germany: Springer-Verlag, 2006, pp. 359–365.

[44] A. Happonen, E. Hemming, and M. Juntti, “A novel coarse grain reconfigurable processing element architecture,” in *Proc. Midwest Symp. Circuits and Systems*, vol. 3, Cairo, Egypt, Dec. 2003, pp. 827–830.

[45] J. G. Harrison, *Implementation of a 3GPP Turbo Decoder on a Programmable DSP Core*, Oct. 2001, 3DSP Corporation.

[46] M. Haselman, M. Beauchamp, K. Underwood, and K. S. Hemmert, “A comparison of floating point and logarithmic number systems for FPGAs,” in *Proc. 13th Ann. IEEE Symp. on Field-Programmable Custom Computing Machines*, Napa, CA, USA, Apr. 2005, pp. 181–190.

[47] X. He, H. Luo, and H. Zhang, “A novel storage scheme for parallel turbo decoder,” in *Proc. IEEE Vehicular Technol. Conf.*, Dallas, TX, USA, Sept. 2005, pp. 1950–1954.

[48] A. P. Hekstra, “An alternative to metric rescaling in Viterbi decoders,” *IEEE Tran. Commun.*, vol. 37, no. 11, pp. 1220–1222, Nov. 1989.

[49] H. Hendrix, *Viterbi Decoding Techniques for the TMS320C54x DSP Generation*, Texas Instruments, Jan. 2002, SPRA071A.

[50] B. M. Hochwald and S. ten Brink, “Achieving near-capacity on a multiple-antenna channel,” *IEEE Tran. on Communications*, vol. 51, no. 3, pp. 389–399, Mar. 2003.

[51] *Using Streaming SIMD Extensions to Evaluate a Hidden Markov Model with Viterbi Decoding*, Intel, Jan. 1999, AP-811.

[52] P. Ituero and M. López-Vallejo, “New schemes in clustered VLIW processors applied to turbo decoding,” in *Proc. IEEE Int. Conf. Application-Specific Syst. Architectures Processors*, vol. 2, Steamboat Springs, CO, USA, Sept. 2006, pp. 291–296.

[53] ——, “Further specialization of clustered VLIW processors: A MAP decoder for software defined radio,” *ETRI J.*, vol. 30, no. 1, pp. 113–128, Feb. 2008.

[54] P. Jääskeläinen, V. Guzma, A. Clio, T. Pitkänen, and J. Takala, “Codesign toolset for application-specific instruction-set processors,” in *Proc. SPIE, Multimedia Mobile Devices*, vol. 6507, San Jose, CA, USA, Jan. 2007, pp. 05 070X–1–10.

[55] V. K. Jain, S. Shrivastava, A. D. Snider, D. Damerow, and D. Chester, “Hardware implementation of a nonlinear processor,” in *Proc. IEEE Int. Symp. on Circuits and Systems*, vol. 6, Orlando, FL, USA, Jun. 1999, pp. 509–514.

[56] J. Janhunen, O. Silvén, M. Myllylä, and M. Juntti, “A DSP implementation of a k-best list sphere detector algorithm,” in *Proc. Finnish Sign. Proc. Symp.*, Oulu, Finland, Aug. 2007.

[57] T. A. K. K. Loo and S. A. Jumaa, “High performance parallelised 3GPP turbo decoder,” in *Proc. IEE European Personal Mobile Commun. Conf.*, Glasgow, UK, Apr. 2003, pp. 337–342.

[58] I. Karkowski and H. Corporaal, “A framework for design of heterogeneous multi-processor embedded systems,” Delft University of Technology, Tech. Rep. 1-68340-44/1997/12, 1997.

[59] A. C. Kay, “Predicting the future,” *Stanford Engineering*, vol. 1, no. 1, pp. 1–6, 1989.

[60] W. Koch and A. Baier, “Optimum and sub-optimum detection of coded data disturbed by time-varying intersymbol interference,” in *Proc. Global Telecommm. Conf.*, San Diego, CA, USA, Dec. 1990, pp. 1679–1684.

[61] T. Kumura, M. Ikekawa, M. Yoshida, and I. Kuroda, “VLIW DSP for mobile applications,” *IEEE Signal Processing Mag.*, vol. 19, no. 4, pp. 10–21, Jul. 2002.

[62] S. Y. Kung, *VLSI Array Processors*. Upper Saddle River, NJ, USA: Prentice-Hall, 1987.

[63] H. Kwan, R. L. Nelson, and E. E. Swartzlander, “Cascaded implementation of an iterative inverse-square-root algorithm, with overflow lookahead,” in *Proc. 12th IEEE Symp. on Computer Arithmetic*, Bath, UK, Jul. 1995, pp. 115–122.

[64] G. Lakhani, “VLSI design of modulo adders/subtractors,” in *Proc. IEEE Int. Conf. Comp. Design: VLSI Comps. & Procs.*, Cambridge, MA, USA, Oct. 1992, pp. 68–71.

[65] T. Lang and E. Antelo, “Radix-4 reciprocal square-root and its combination with division and square root,” *IEEE Trans. On Computers*, vol. 52, no. 9, pp. 1100–1114, Sep. 2003.

[66] I. Lee and J. L. Sonntag, “A new architecture for the fast Viterbi algorithm,” *IEEE Trans. on Commun.*, vol. 51, no. 10, pp. 1624–1628, Oct. 2003.

[67] Y.-T. Lin, P.-Y. Tsai, and T.-D. Chiueh, “Low-power variable-length fast Fourier transform processor,” *IEE Proc. Comput. Digit. Tech.*, vol. 152, no. 4, pp. 499–506, July 2005.

[68] Y. Lin, S. Mahlke, T. Mudge, and C. Chakrabarti, “Design and implementation of turbo decoders for software defined radio,” in *Proc. IEEE Workshop Sign. Proc. Syst.*, Banff, Canada, Oct. 2006, pp. 22–27.

[69] Z. Liu, K. Dickson, and J. V. McCanny, "Application-specific instruction set processor for SoC implementation of modern signal processing algorithms," *IEEE Tran. Circuits and Syst.*, vol. 52, no. 4, pp. 755–765, Apr. 2006.

[70] C. Lomont, "Fast inverse square root," Department of Mathematics, Purdue University, Tech. Rep., Feb. 2003.

[71] E.-H. Lu, L. Harn, J.-Y. Lee, and W.-Y. Hwang, "A programmable VLSI architecture for computing multiplication and polynomial evaluation modulo a positive integer," *IEEE J. Solid-State Circuits*, vol. 23, no. 1, pp. 204–207, Feb. 1988.

[72] A. Maltsev, V. Pestretsov, R. Maslennikov, and A. Khoryaev, "Triangular systolic array with reduced latency for QR-decomposition of complex matrices," in *Proc. IEEE Inter. Symp. Circuits and Syst.*, Kos, Greece, May 2006, pp. 385–388.

[73] R. J. McEliece, *Finite Fields for Computer Scientists and Engineers*. Boston, USA: Kluwer Academic, 1987.

[74] H. Michel, A. Worm, M. Münch, and N. Wehn, "Hardware/software tradeoffs for advanced 3G channel coding," in *Proc. Design, Automation, and Test in Europe Conf.*, Paris, France, Mar. 2002, pp. 396–401.

[75] J. Mitola III, "Software radios survey, critical evaluation and future directions," *IEEE Aerosp. Electon. Syst. Mag.*, vol. 8, no. 4, pp. 25–36, Apr. 1993.

[76] ——, "The software radio architecture," *IEEE Commun. Mag.*, vol. 33, no. 5, pp. 26–38, May 1995.

[77] ——, "Cognitive radio: Making software radios more personal," *IEEE Personal Commun.*, vol. 6, no. 5, pp. 13–18, Aug. 1999.

[78] M. Myllylä, P. Silvola, M. Juntti, and J. R. Cavallaro, "Comparison of two novel list sphere detector algorithms for MIMO-OFDM systems," in *Proc. Int. Symp. Personal, Indoor, and Mobile Radio Commun.*, Helsinki, Finland, Sept. 2006.

[79] C. Neeb, M. J. Thul, and N. Wehn, "Network-on-chip-centric approach to interleaving in high throughput channel decoders," in *Proc. IEEE Int. Symp. Circuits and Syst.*, Kobe, Japan, May 2005, pp. 1766–1769.

[80] T. Ngo and I. Verbauwhede, "Turbo codes on the fixed point DSP TMS320C55x," in *Proc. IEEE Workshop SiPS 2000*, Oct. 2000, pp. 255–264.

[81] J. Nikolic-Popovic, *Implementing a MAP Decoder for cdma2000^{TM} Turbo Codes on a TMS320C62x DSP Device*, Texas Instruments, May 2000, SPRA629.

[82] S. F. Oberman, "Floating point division and square root algorithms and implementation in the AMD-K7 microprocessor," in *Proc. 14th IEEE Symp. on Computer Arithmetic*, Adelaide, Australia, Apr. 1999, pp. 106–115.

[83] A. J. Paulraj, D. A. Gore, R. U. Nabar, and H. Bölcskei, "An overview of MIMO communications – a key to gigabit wireless," *Proc. IEEE*, vol. 92, no. 2, pp. 198–218, Feb. 2004.

[84] D. Perels, S. Haene, P. Luethi, A. Burg, N. Felber, W. Fichtner, and H. Bölcskei, "ASIC implementation of a MIMO-OFDM transceiver for 192 Mbps WLANs," in *Proc. Europ. Solid-State Circuits Conf.*, Grenoble, France, Sept. 2005, pp. 215–218.

[85] S. J. Piestrak, "Design of squares modulo A with low-level pipelining," *IEEE Tran. Circuits and Syst. II: Analog and Digital SP*, vol. 49, no. 1, pp. 31–41, Jan. 2002.

[86] J.-A. Pineiro and J. Bruguera, "High-speed double-precision computation of reciprocal, division, square root, and inverse square root," *IEEE Trans. On Computers*, vol. 51, no. 12, pp. 1377–1388, Dec. 2002.

[87] T. Pitkänen, R. Mäkinen, J. Heikkinen, T. Partanen, and J. Takala, "Low-power, high-performance TTA processor for 1024-point fast Fourier transform," in *Embed. Comp. Syst.: Architectures, Modelling, and Simulation*, vol. LNCS 4017. Berlin, Germany: Springer-Verlag, 2006, pp. 227–236.

[88] P. Robertson, P. Hoeher, and E. Villebrun, "Optimal and sub-optimal maximum a posteriori algorithms suitable for turbo decoding," *European Trans. on Telecomm.*, vol. 8, pp. 119–125, 1997.

[89] A. L. Rosa, C. Passerone, F. Gregoretti, and L. Lavagno, “Implementation of a UMTS turbo-decoder on dynamically reconfigurable platform,” in *Proc. Design, Automation, and Test in Europe Conf.*, vol. 2, Paris, France, Feb 2004, pp. 1218–1223.

[90] J. Rossier, Y. Thoma, P.-A. Mudry, and G. Tempesti, “MOVE processors that self-replicate and differentiate,” in *Biologically Inspired Approaches to Advanced Inf. Techn.*, vol. LNCS 3853. Berlin, Germany: Springer-Verlag, 2006, pp. 160–175.

[91] K. Rounioja and J. A. Parviainen, “Arithmetic processing unit for reciprocal operations,” in *Proc. International Symp. on System-On-Chip*, Tampere, Finland, Nov. 2003, pp. 109–112.

[92] P. Salmela, C.-C. Shen, S. S. Bhattacharyya, and J. Takala, “Synthesis of DSP architectures using libraries of coarse-grain configurations,” in *Proc. IEEE Workshop Sign. Proc. Syst.*, Shanghai, China, Oct. 2007, pp. 475–480.

[93] M. J. Schulte and K. E. Wires, “High-speed inverse square roots,” in *Proc. 14th IEEE Symp. on Computer Arithmetic*, Adelaide, Australia, Apr. 1999, pp. 124–131.

[94] *UMTS_VIT Viterbi decoder for UMTS*, Sci-worx, VC01B1.011HO.

[95] M.-C. Shin and I.-C. Park, “Processor-based turbo interleaver for multiple third-generation wireless standards,” *IEEE Commun. Letters*, vol. 7, no. 5, pp. 210–212, May 2003.

[96] ——, “SIMD processor-based turbo decoder supporting multiple third-generation wireless standards,” *IEEE Trans. on VLSI Syst.*, vol. 15, no. 7, pp. 801–810, July 2007.

[97] D.-S. Shiu and I. Yao, “Buffer architecture for a turbo decoder,” International Patent Application WO 02/093 755 A1, Nov., 2002.

[98] C. K. Singh, S. H. Prasad, and P. T. Balsara, “A fixed-point implementation for QR decomposition,” in *IEEE Dallas Workshop Design, Applications, Integration and Software*, Dallas, TX, USA, Oct. 2006, pp. 75–78.

[99] ——, “VLSI architecture for matrix inversion using modified Gram-Schmidt based QR decomposition,” in *Proc. Int. Conf. VLSI Design*, Bangalore, India, Jan. 2007, pp. 836–841.

[100] *SiWorks IP Cores Series: 802.11a/g Viterbi Decoder*, Siworks, 2003.

[101] A. Sripad and D. Snyder, “A necessary and sufficient condition for quantization error to be uniform and white,” *IEEE Trans. On Acoustic, Signal, and Speech Processing*, vol. 25, no. 5, pp. 442–448, Oct. 1997.

[102] J. E. Stine and M. J. Schulte, “The symmetric table addition method for accurate function approximation,” *Journal of VLSI Signal Processing*, vol. 21, no. 2, pp. 166–167, Jun. 1999.

[103] J. Sun and O. Y. Takeshita, “Interleavers for turbo codes using permutation polynomials over integer rings,” *IEEE Trans. Information Theory*, vol. 51, no. 1, pp. 101–119, Jan. 2005.

[104] Y. Sun, Y. Zhu, M. Goel, and J. R. Cavallaro, “Configurable and scalable high throughput turbo decoder architecture for multiple 4G wireless standards,” in *Proc. IEEE Int. Conf. Application-Specific Systs., Architectures and Procs.*, Leuven, Belgium, Jul. 2008, pp. 209–214.

[105] N. Takagi, “Generating a power of an operand by a table look-up and multiplication,” in *Proc. 13th IEEE Symp. on Computer Arithmetic*, Asilomar, CA, USA, Jul. 1997, pp. 126–131.

[106] ——, “A hardware algorithm for computing reciprocal square root,” in *Proc. 15th IEEE Symp. on Computer Arithmetic*, Vail, CO, USA, Jun. 2001, pp. 94–100.

[107] ——, “Powering by a table look-up and a multiplication with operand modification,” *IEEE Trans. On Computers*, vol. 47, no. 11, pp. 1216–1222, Nov. 1998.

[108] A. Tarable, S. Benedetto, and G. Montorsi, “Mapping interleaving laws to parallel turbo and LDPC decoder architectures,” *IEEE Trans. Information Theory.*, vol. 50, no. 9, pp. 2002–2009, Sept. 2004.

[109] G. Tempesti, P.-A. Mudry, and R. Hoffman, "A move processor for bio-inspired systems," in *Proc. NASA/DoD Conf. Evolvable Hardware*, Washington DC, USA, June–July 2005, pp. 262–271.

[110] *TMS320C54x DSP Programmer's Guide*, Texas Instruments, Jul. 2001, SPRU538.

[111] *TMS320C55x DSP Programmer's Guide*, Texas Instruments, Aug. 2001, SPRU376A.

[112] *TMS320C64x Technical Overview*, Texas Instruments, Jan. 2001, SPRU395B.

[113] *TMS320C64x DSP Viterbi-Decoder Coprocessor (VCP) Reference Guide*, Texas Instruments, Nov. 2003, SPRU533C.

[114] *TMS320C55x DSP CPU Reference Guide*, Texas Instruments, Feb. 2004, SPRU371F.

[115] *TMS320VC5502 Fixed-Point Digital Signal Processor Data Manual*, Texas Instruments, Nov. 2008, SPRS166K.

[116] M. J. Thul, N. Wehn, and L. P. Rao, "Enabling high-speed turbo-decoding through concurrent interleaving," in *Proc. IEEE Intern. Symp. Circuits and Syst.*, vol. 1, Phoenix, USA, May 2002, pp. 897–900.

[117] M. J. Thul, F. Gilbert, T. Vogt, G. Kreiselmaier, and N. Wehn, "A scalable system architecture for high-throughput turbo decoders," *J. VLSI Signal Processing*, vol. 39, no. 1–2, pp. 63–77, 2005.

[118] K. Turkowski, "Computing the inverse square root," Media Technologies: Computer Graphics Advanced Technology Group Apple Computer, Inc., Tech. Rep. 95, Oct. 1994.

[119] M. C. Valenti and J. Sun, "The UMTS turbo code and an efficient decoder implementation suitable for software-defined radios," *Int. Journ. Wireless Inform. Networks*, vol. 8, no. 4, pp. 203–215, Oct. 2001.

[120] A. J. Viterbi, "An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes," *IEEE J. Select. Areas Commun.*, vol. 16, no. 2, pp. 260–264, Feb. 1998.

[121] ——, “Error bounds for convolutional codes and an asymptotically optimum decoding algorithm,” *IEEE Trans. Information Theory*, vol. IT-13, no. 2, pp. 260–269, Apr. 1967.

[122] J. Vogt and A. Finger, “Improving the max-log-MAP turbo decoder,” *Electronics Letters*, vol. 36, no. 23, pp. 1937–1939, Nov. 2000.

[123] J. Vogt, K. Koora, A. Finger, and G. Fettweis, “Comparison of different turbo decoder realizations for IMT-2000,” in *Proc. GLOBECOM ’99*, vol. 5, Rio de Janeiro, Brazil, Dec. 1999, pp. 2704–2708.

[124] T. Vogt and N. Wehn, “A reconfigurable application specific instruction set processor for Viterbi and log-MAP decoding,” in *Proc. IEEE Workshop Sign. Proc. Syst.*, Banff, Canada, Oct. 2006, pp. 142–147.

[125] Z. Wang, “High-speed recursion architectures for MAP-based turbo decoders,” *IEEE Trans. on VLSI Syst.*, vol. 15, no. 4, pp. 470–474, Apr. 2007.

[126] Z. Wang and K. Parhi, “Efficient interleaver memory architectures for serial turbo decoding,” in *Proc. IEEE Int. Conf. Acoustics, Speech and Sign. Proc.*, vol. 2, Hong Kong, China, Apr. 2003, pp. 629–632.

[127] Z. Wang, H. Suzuki, and K. K. Parhi, “VLSI implementation issues of turbo decoder design for wireless applications,” in *Proc. IEEE Workshop Sign. Proc. Syst.*, Taipei, Taiwan, Oct. 1999, pp. 503–512.

[128] Z. Wang, Y. Tan, and Y. Wang, “Low hardware complexity parallel turbo decoder architecture,” in *Proc. IEEE Int. Symp. Circuits and Syst.*, vol. 2, Bangkok, Thailand, May 2003, pp. 53–56.

[129] Z. Wang, H. Suzuki, and K. K. Parhi, “Finite wordlength analysis and adaptive decoding for turbo/MAP decoders,” *J. VLSI Signal Process. Syst.*, vol. 29, no. 3, pp. 209–221, Nov. 2001.

[130] M. Wenk, M. Zellweger, A. Burg, N. Felber, and W. Fichtner, “K-best MIMO detection VLSI architectures achieving up to 424 Mbps,” in *Proc. Int. Symp. on Circuits and Syst.*, Kos, Greece, May 2006, pp. 1151–1154.

[131] B. Widdup, G. Woodward, and G. Knagge, “A highly-parallel VLSI architecture for a list sphere detector,” in *Proc. IEEE Int. Conf. on Communications*, Paris, France, Jun. 2004, 2720–2725.

[132] A. Wiesel, X. Mestre, A. Pagés, and J. R. Fonollosa, “Efficient implementation of sphere demodulation,” in *Proc. IEEE Workshop on Sign. Poc. Advances in Wireless Comm.*, Rome, Italy, Jun. 2003, 36–40.

[133] M. Woh, S. Seo, H. Lee, Y. Lin, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, “The next generation challenge for software defined radio,” in *Embed. Comp. Syst.: Architectures, Modelling, and Simulation*, vol. LNCS 4599. Berlin, Germany: Springer-Verlag, 2007, pp. 343–354.

[134] T. Wolf, D. Hocevar, A. Gatherer, P. Geremia, and A. Laine, “600 MHz DSP for baseband processing in 3G base stations,” in *Proc. IEEE Custom Integ. Circ. Conf.*, Orlando, FL, USA, May 2002, pp. 393–396.

[135] K.-W. Wong, C.-Y. Tsui, R. S.-K. Cheng, and W.-H. Mow, “A VLSI architecture of a k-best lattice decoding algorithm for MIMO channels,” in *Proc. IEEE Int. Symp. Circuits and Syst.*, Scottsdale, AZ, USA, May 2002, pp. 273–276.

[136] W. F. Wong and E. Goto, “Fast hardware-based algorithms for elementary function computations using rectangular multipliers,” *IEEE Trans. On Computers*, vol. 43, no. 3, pp. 278–294, Mar. 1994.

[137] Xilinx, *3GPP Turbo Decoder v3.1*, May 2007, DS318.

[138] Y. Zhang, J. Tang, and K. K. Parhi, “Low complexity list updating circuits for list sphere decoders,” in *Proc. IEEE Workshop on Sign. Poc. Systs.*, Banff, Canada, Oct. 2006, 28–33.