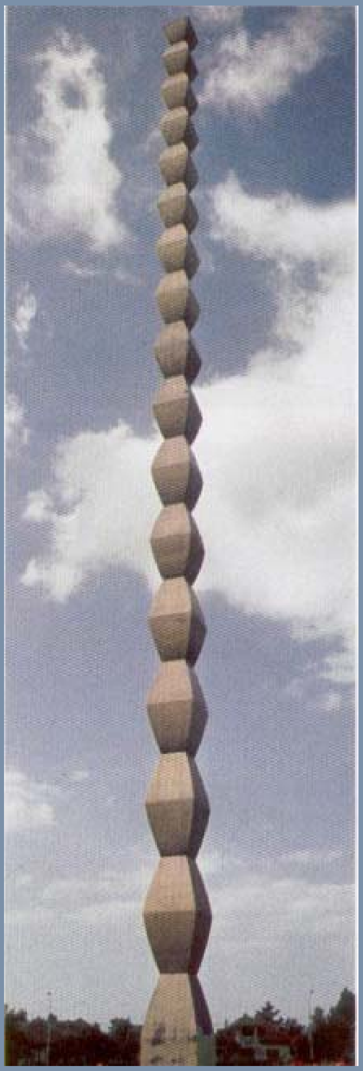
The ρ–TriMedia Processor

Mihai Sima



The ρ–TriMedia Processor

On the cover: Constantin Brâncuşi's monumental *Endless Column* in Târgu-Jiu, Romania. The 29.33-meter column of metal-coated cast-iron modules on a steel spine is part of a sculptural ensemble originally installed in 1937-1938, which also includes the travertine Table of Silence and Gate of the Kiss.

The ρ–TriMedia Processor

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen

op maandag 15 maart 2004 om 13:00 uur

door

Mihai SIMA

inginer
Facultatea de Electronică şi Telecomunicaţii
Institutul Politehnic din Bucureşti
geboren te Boekarest, Roemenië

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr. S. Vassiliadis

Toegevoegd promotor:
Dr. S. Cotofana

Samenstelling promotiecommissie:

Rector Magnificus	Technische Universiteit Delft, voorzitter
Prof.dr. S. Vassiliadis	Technische Universiteit Delft, promotor
Dr. S. Cotofana	Technische Universiteit Delft, toegevoegd promotor
Prof.ir. G.L. Reijns	Technische Universiteit Delft
Prof.dr.dr.h.c.mult. M. Glesner	Darmstadt University of Technology
Prof.dr. R. Stefanelli	Politecnico di Milano
Prof.dr.ir. E. Deprettere	Universiteit Leiden
Dr.ir. J.T.J. van Eijndhoven	Philips Research Laboratories
Prof.dr. C.I.M. Beenakker	Technische Universiteit Delft, reservelid

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Sima, Mihai
The ρ–TriMedia Processor
Mihai Sima. – [S.l. : s.n.]. – Ill.
Thesis Technische Universiteit Delft. – With ref. –
Met samenvatting in het Nederlands. Include un rezumat în limba română. Περιλαμβάνει περίληψη στα Ελληνικά.
ISBN 90-5326-041-2
Subject headings: reconfigurable computing, computer design and engineering
Cover illustration: “Endless Column” by Constantin Brâncuşi, Târgu-Jiu, Romania

Keywords: VLIW processors, reconfigurable hardware, media processing

Copyright © 2004 Mihai SIMA `msima@ece.uvic.ca`
All rights reserved. No part of this dissertation may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the consent of the author.

To the memory of my father

The ρ–TriMedia Processor

Mihai Sima

Abstract

In this dissertation, we present an augmentation of the TriMedia–CPU64™ VLIW processor with a Field-Programmable Gate Array (FPGA), and assess the potential performance of this hybrid for performing media-oriented tasks. Since only minimal modifications of the processor organization are allowed, the FPGA is connected to TriMedia–CPU64 as any other hardwired functional unit. The resulting hybrid is referred to as ρ–TriMedia. We first describe an extension of the TriMedia–CPU64 instruction set architecture that incorporates support for the FPGA. Essentially, a kernel of new instructions denoted as `SET` and `EXECUTE` is provided. The `SET` instruction controls the reconfiguration of the FPGA, and the `EXECUTE` instruction launches the operations performed by the FPGA-mapped computing units. The approach is generic, consequently the user is given the freedom to define and use any customized computing units. Moreover, by using the opcode fields in adjacent VLIW issue-slots to define an argument for the `EXECUTE` opcode, a large number of reconfigurable operations can be encoded, while only a single entry for the `EXECUTE` instruction needs to be allocated in the opcode space. This way, the reconfigurable operation does not create pressure on the instruction decoder, neatly fits in the existing instruction format and the existing connectivity structure to the register file, and hence requires a minimal hardware overhead. To assess the potential performance of ρ–TriMedia, we focus on the MPEG standard, and address a number of computing-intensive media kernels: Inverse Discrete Cosine Transform, Inverse Quantization, Entropy Decoding, and YCC-to-RGB Color Space Conversion. For each kernel, an FPGA-based computing unit is designed. The ACEX™ EP1K100 FPGA from Altera is utilized as a reconfigurable core. The experiments carried out on a TriMedia–CPU64 cycle accurate simulator indicate that a speed-up of more than 40% on ρ–TriMedia over the standard TriMedia–CPU64 is achieved for a number of MPEG2-conformance scenes. Given the fact that TriMedia–CPU64 is a 5-issue slot VLIW processor with a 64-bit datapath and a very rich media-oriented instruction set, such an improvement within its target media processing domain achieved with a relatively small FPGA, indicates that FPGA–augmented TriMedia–CPU64 (ρ–TriMedia) is a promising approach.

Acknowledgments

The work presented in this dissertation contains the results of my research performed at the Computer Engineering Laboratory of the Electrical Engineering Department, Delft University of Technology, and Philips Research Laboratories in Eindhoven, The Netherlands in the last four years. During that time I came across many people who supported and assisted me. I would like to take the opportunity to thank them.

First and foremost, I thank my advisor Prof.dr. Stamatis Vassiliadis for giving me the opportunity to perform my PhD research within his group, as well as for his technical advices and moral support over the years. I am especially indebted to him for guiding me through the fascinating domain of reconfigurable computing. His belief that this new paradigm is about to become the breaking idea in a computer science world almost drained of major ideas is the *wind in poop* without which the 'Computer Engineering Laboratory' ship would never reach its destination.

I would also like to acknowledge the contribution of Dr. Sorin Coţofană. In addition to supporting me during the research activity, he has guided me through the trials and tribulations of the life abroad. He has been an endlessy patient partner and friend always ready to help me. Thank you.

I also thank Prof.dr. Mircea Bodea for giving a hand to my destiny. Without his support I might have never had the chance to start a dissertation at Delft University of Technology.

At Philips Research Laboratories in Eindhoven I hugely enjoyed the company of Dr. Jos van Eijndhoven. His numerous suggestions have improved the content of this dissertation. Thank you.

To my colleagues Stephan Wong, Stephan Suijkerbuijk, and Pyrrhos Stathis who helped me with propositions translation into Dutch, abstract translation into Dutch, and abstract translation into Greek, respectively, a warm thank you.

I want also thank Lidwina Tromp for taking the trouble to help me through the bureaucratic process that I came across. Without her support, I would have not been able to focus strictly on research.

Special thanks go to my good friend Mike Mananedakis, who proved that ‘a friend in need is a friend indeed.’

To my parents, for all their unconditional support they have given me over the years. Their belief in me and my abilities has allowed me to pursue my dreams. For this and their love I owe them a debt of gratitude.

This project was funded by the doctoral fellowship RWC-061-PS-99047-ps from Philips Research Laboratories in Eindhoven, The Netherlands.

Mihai Sima Victoria, B.C., Canada
February 11, 2004

Contents

Abstract **i**

Acknowledgments **iii**

List of Acronyms **1**

List of Figures **3**

List of Tables **7**

List of Algorithms **9**

List of Trademarks **11**

1 Introduction **13**

- 1.1 Problem overview and dissertation scope 15
- 1.2 Open questions and terminology 17
- 1.3 Overview of dissertation 20

2 Reconfigurable Computing Paradigm **23**

- 2.1 Reconfigurable Hardware 24
- 2.2 A microcoded computing machine 26
- 2.3 Field-programmable Custom Computing Machines 29
- 2.4 Conclusion 38

3 The ρ–TriMedia Architecture 39
3.1 Standard TriMedia–CPU64 architecture 40
3.2 The architecture of the multiple-context FPGA 42
3.3 The ρ–TriMedia processor 44
3.4 ρ–TriMedia development flowchart 49
3.5 ρ–TriMedia programming methodology 50
3.6 TriMedia–CPU64 processing domain 57
3.7 Conclusion 58

4 Inverse Discrete Cosine Transform 61
4.1 Theoretical background 61
4.2 2-D IDCT pure-software implementation 63
4.3 1-D IDCT implementation on FPGA 64
4.4 2-D IDCT implementation on ρ–TriMedia 72
4.5 Conclusion 79

5 Inverse Quantization 81
5.1 Theoretical background 82
5.2 IQ pure-software implementation 83
5.3 IQ implementation on FPGA 84
5.4 IQ implementation on ρ–TriMedia 86
5.5 Conclusion 97

6 Entropy Decoding 99
6.1 Theoretical background 100
6.2 Entropy decoder pure-software implementation 103
6.3 Variable-length decoding in FPGA 113
6.4 Entropy decoder implementation on ρ–TriMedia 121
6.5 Experimental results 128
6.6 Conclusion 137

7 Pel Reconstruction 139

7.1 Pel reconstruction theoretical background 139

7.2 Pel reconstruction implementation on ρ–TriMedia 141

7.3 Experimental results 146

7.4 Conclusion 149

8 YCC-to-RGB Color Space Conversion 151

8.1 Theoretical background 152

8.2 YCC-to-RGB pure-software implementation 154

8.3 YCC-to-RGB implementation on FPGA 155

8.4 YCC-to-RGB implementation on ρ–TriMedia 156

8.5 Conclusion 161

9 Conclusions 163

9.1 Summary 163

9.2 Contributions 166

9.3 Proposed research directions 168

A MPEG2 Statistics 169

Bibliography 171

Samenvatting 185

Rezumat în limba română 187

Περίληψη στα ελληνικά 189

List of Publications 191

Curriculum Vitae 195

List of Acronyms

ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction Processor
CPU	Central Processing Unit
CU	Configuration Unit
FCCM	Field-programmable Custom Computing Machines
FPA	Field-Programmable Array
FPGA	Field-Programmable Gate Array
FPL	Field-Programmable Logic
GPP	General-Purpose Processor
IDCT	Inverse Discrete Cosine Transform
IQ	Inverse Quantization, Inverse Quantizer
MPEG	Moving Picture Experts Group
RC	Reconfigurable Computing
RFU	Reconfigurable Functional Unit
RLD	Run-Length Decoder
SRAM	Static Random-Access Memory
VLD	Variable-Length Decoding, Variable-Length Decoder
VLIW	Very-Long Instruction Word

List of Figures

1.1 The performance of FCCMs against GPPs and ASICs. 14

2.1 Context ID broadcasting. 25
2.2 The basic microprogrammed computer. 27
2.3 The microcode concept applied to FCCMs. The $\boldsymbol{\Delta}$ arrangement. . 30
2.4 The PRISC architecture 32
2.5 The MorphoSys architecture 33
2.6 The PipeRench architecture (solid lines are data paths, dashed lines are address and control paths) 34
2.7 The RaPiD engine – adapted from [33] 35
2.8 The symplified organization of the MOLEN processor – from [117]. 36
2.9 The MOLEN instruction format. 37

3.1 TriMedia–CPU64 organization – adapted from [113]. 40
3.2 ACEX 1K logic element structure and timing microparameters. . . 43
3.3 The TriMedia-CPU64 VLIW core extension. 45
3.4 The hardwired double-slot operation instruction format. 48
3.5 The RFU double-slot instruction format. 48
3.6 The syntax and annotation code for a user-defined IDCT operation. 49
3.7 100 MHz reduction modules on Altera's ACEX EP1K100 (speed grade -1) FPGA. 56
3.8 A generic video codec. 57

4.1 The modified 'Loeffler' algorithm. 62

4.2 The butterfly – [67]. 63

4.3 The rotator – [67]. 63

4.4 Three possible implementations of the rotator. 63

4.5 The computing scenario of 8×8 IDCT on the extended TriMedia. 65

4.6 Bypassing the first transposition. 65

4.7 Sign-extension for: (a) – 15-bit multiplier; (b) – 13-bit multiplier. . 67

4.8 *Rounding-To-Nearest* implementation. 67

4.9 Rounding a real value to the nearest integer. 67

4.10 The partial product matrix and the selected reduction steps for multiplication by the constant C'_0. 68

4.11 The partial product matrix and the selected reduction steps for multiplication by the constant C'_1. 69

4.12 The partial product matrix and the selected reduction steps for multiplication by the constant S'_1. 69

4.13 The partial product matrix and the selected reduction steps for multiplication by the constant C'_3. 70

4.14 The partial product matrix and the selected reduction steps for multiplication by the constant S'_3. 70

4.15 The partial product matrix and the selected reduction steps for multiplication by the constant C'_6. 71

4.16 The partial product matrix and the selected reduction steps for multiplication by the constant S'_6. 71

4.17 The 1-D IDCT pipeline. 72

4.18 Schedule result for a 1-D IDCT having the latency of 16 and recovery of 2 (LD stands for `LOAD`, RD for read, WR for write, ST for `STORE`, and T for `TRANSPOSE`). 73

4.19 The speed-up of various 2-D IDCT implementations on several state-of-the-art architectures relative to standard TriMedia–CPU64. 80

5.1 The IQ-4 implementation on FPGA. 84

5.2 Schedule result for an IQ-4 unit having the latency of 18 and recovery of 2 (LD stands for `LOAD`, RD for read, WR for write, and ST for `STORE`). 87

5.3 Schedule result for an IQ-8 unit having the latency of 18 and recovery of 2 (LD stands for `LOAD`, RD for read, WR for write, and ST for `STORE`). 89

5.4 Schedule result for an IQ-8 unit having the latency of 18 and recovery of 2. Two 8×8 blocks are processed per loop iteration (LD stands for `LOAD`, RD for read, WR for write, and ST for `STORE`). 90

5.5 Schedule result for two IQ-4 units, each having the latency of 18 and recovery of 2 (LD stands for `LOAD`, RD for read, WR for write, and ST for `STORE`). 91

5.6 Schedule result for two IQ-4 units, each having the latency of 18 and recovery of 2. A single 8×8 block is processed per loop iteration (LD stands for `LOAD`, RD for read, WR for write, and ST for `STORE`). 93

5.7 Schedule result for four IQ-4 units, each having the latency of 18 and recovery of 2 (LD stands for `LOAD`, RD for read, WR for write, and ST for `STORE`). 95

6.1 Variable-length decoding principle. 100

6.2 Run-length decoding principle. 101

6.3 MPEG2–compliant entropy decoding example. 102

6.4 The flowchart of the variable-length decoding procedure. 112

6.5 The conceptual VLD-1 implementation on FPGA. 117

6.6 The detailed VLD-1 implementation on FPGA. 118

6.7 The FPGA–based VLD-2. 120

6.8 The *original bitfunshift* and *extended bitfunshift* operations. 122

6.9 Updating the *register* copy of the leading (most significant) 64-bit chunks of the VLC bit-string. 122

6.10 The dataflow of the VLD-1–based entropy decoder: (a) – without grafting; (b) – with grafting. 125

6.11 The firing-up process of the entropy decoder software pipeline. . . 127

6.12 The format of the *nz_coeff_pos* relative to a macroblock. 129

7.1 Pel reconstruction conceptual diagram – adapted from [70]. 141

7.2 The pieces of code that benefit from FPGA support. 142

7.3 Three possible computing scenarios of pel reconstruction. 143
7.4 The winning computing scenario of pel reconstruction. 147
7.5 ρ–TriMedia versus standard TriMedia speed-up. 148

8.1 Two–dimensional 2–fold upsampling by replication. 153
8.2 Color space conversion strategy for the inner loop. 155
8.3 The color space converter implementation on FPGA. 157
8.4 Scheduling result for a CSC unit having the latency of 10 and recovery of 2. 158

List of Tables

2.1 The PRISC PFU instruction format 33

3.1 Performances of several reduction modules for ACEX EP1K100 FPGA (Speed Grade -1). 54

4.1 The average number of coded blocks per slice for a number of MPEG-conformance bit-strings. 75

4.2 Performance figures for 8×8 IDCT on (FPGA-augmented) TriMedia. 76

4.2 Performance figures for 8×8 IDCT on (FPGA-augmented) TriMedia (contd.). 77

4.3 2-D IDCT performance figures on state-of-the-art architectures. . 79

5.1 The average number of coded blocks per slice – from Appendix A. 88

5.2 Performance figures for IQ. 96

6.1 Selection of VLC tables 101

6.2 The original VLC table format. 104

6.3 The proposed VLC table format. 107

6.4 Number of address lines, size, and offset for each VLC table. . . . 108

6.5 First VLC partition 109

6.6 Second VLC partition 109

6.7 Third VLC partition 109

6.8 Forth VLC partition 109

6.9 Fifth VLC partition 109

6.10 Sixth VLC partition 110
6.11 Seventh VLC partition 110
6.12 Eighth VLC partition 110
6.13 Partitioning of the Tables B14 and B15 into groups and classes. . . 116
6.14 Intrinsic MPEG-2 stream statistics – excerpt from Appendix A. . . 129
6.15 VLD-2 – The format of the first argument register – Ry (`uint64`). 131
6.16 VLD-2 – The format of the second argument register – Ryy (`uint64`). 131
6.17 VLD-2 – The format of the rezult register Rz (`vec64sh`). 131
6.18 VLD-2 – The format of the rezult register Rw (`vec64ub`). 132
6.19 Entropy decoding experimental results. 134
6.20 Relative efficiency of a hardwired VLD-2. 136

7.1 MPEG-2 statistics for several conformance bit-strings – excerpt from Table A.2. 144
7.2 Total number of blocks, macroblocks, slices, and pictures/frames for several MPEG-2 conformance scenes – excerpt from Table A.1. 144
7.3 Cache miss penalty, FPGA context switching overhead, and pipeline fire-up + flushing overhead for different computing scenarios. 145
7.4 Pel reconstruction experimental results for the winning computing scenario. 145
7.5 Pel reconstruction relative improvement. 148

8.1 Performance figures for $Y'CbCr$-to-$R'G'B'$ conversion. 160

A.1 Total number of variable-length symbols (DCT coefficients and *end-of-block*), blocks, macroblocks, slices, and pictures/frames for several MPEG-2 conformance bit-strings. 169
A.2 MPEG-2 statistics for several conformance bit-strings. 170

List of Algorithms

1 An example of a C-level code calling the RFU 51

2 An example of a deep software pipeline calling long-latency RFU-based operation 53

3 IQ-4 with four phases 94

4 Entropy decoder routine – reference implementation 105

5 Converting the *level* from 8-bit unsigned to a 16-bit signed integer 107

6 TriMedia-specific code for testing the *end-of-block* condition . . . 108

7 Entropy decoder routine with the prologue exposed to compiler . . 114

8 Color space conversion – pure software solution. 154

List of Trademarks

All trademarks and registered trademarks are the property of their respective holders and are acknowledged.

TriMedia–CPU64™ is the property of Royal Philips Electronics and designates a processor model targeted to research activities.

ACEX 1K® family of programmable logic devices is the property of Altera Corporation.

MAX+PLUS-II® programmable logic design environment is also the property of Altera Corporation.

Leonardo Spectrum® synthesis environment is the property of Mentor Graphics.

ModelSim® digital simulation tool is also the property of Mentor Graphics.

Chapter 1

Introduction

One of the fundamental trade-offs in the design of a computing machine involves the balance between flexibility and performance. At one extreme of the processing spectrum a *General-Purpose Processor* (GPP) provides flexibility at the expense of performance. At the other extreme *Application-Specific Integrated Circuits* (ASIC), achieve high performance at the expense of flexibility. Concerning multimedia, which is the main domain of this dissertation, we would like to mention that the digital processing of audio, video, speech, and graphics data places a high demands on devices for transmission, storage, and computation. Especially the video requires a large amount of information bandwidth unless compression technology is used. In turn, the compression technology calls for a large amount of processing. For example, National Television Systems Committee (NTSC) resolution MPEG-2 [74] decoding requires more than 400 MOPS, and 30 GOPS are required for encoding in real time. For this reason, the multimedia functions have been traditionally implemented in ASICs, or in hardwired-assists in Application-Specific Instruction Processors (ASIP). Due to the ASIC/ASIP hardwired-assist lack of flexibility, a different full-custom circuit is needed for each particular task. Also, even with slight improvements and/or changes over existing devices, the ASIC has to be redesigned, which translates to a considerable engineering effort. With today's rapidly evolving standards and functional requirements, these fixed-function devices are prone to rapid obsolence. On the other side, a programmable computing platform allows functions to be implemented in software rather than in custom hardware. This dramatically reduces the development cost and time-to-market versus the traditional fixed-function design approach, and ensures that a single device can be applied in a range of different products and adapted to quickly evolving standards in the media domain.

The ability for providing a hardware platform which can be metamorphosed under software control has established *Reconfigurable Computing* ($\mathcal{RC}$) [68], [119], [69], as an emerging computing paradigm for more than ten years. According to this paradigm, the main idea in improving the performance of a computing machine is to define custom computing resources on a per-application basis, and to dynamically configure them onto an *Field-Programmable Gate Array* (FPGA) [14]. In this way, a large number of application-geared computing units can be emulated. Therefore, the reconfigurable computing paradigm appears to be the potential solution to design a computing engine that acts like an ASIC but has the GPP flexibility.

As a general view, a computing machine working under the Reconfigurable Computing paradigm typically includes a *General-Purpose Processor* (GPP) augmented with an FPGA. The basic idea is to exploit both the GPP flexibility to achieve good performance for a large class of control-dominant applications, and FPGA capability to implement application-specific computations for data-dominant applications. Such a hybrid is referred to as a *Field-Programmable Custom Computing Machine* (FCCM) [15], [41]. The synergism of GPP and FPGA may provide the possibility to achieve orders of magnitude improvements in performance over a GPP alone, while preserving the flexibility of the programmed machines over ASICs in implementing a large number of applications. However, the FCCM performance in terms of speed and power may still be lower than the performance of an ASIC. A map of the peformance of FCCMs against GPPs and ASICs is depicted in Figure 1.1.

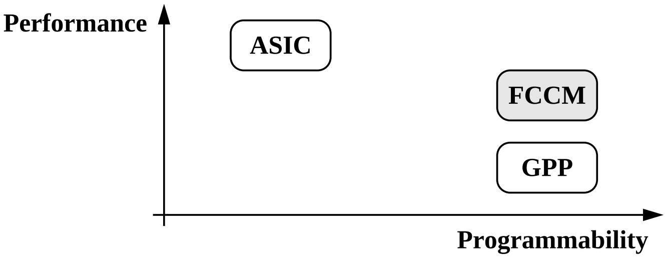


Figure 1.1: **The performance of FCCMs against GPPs and ASICs.**

In this dissertation we focus on *Media FCCMs* – reconfigurable computing platforms that accelerate the digital processing of audio, video, and graphics. In particular, we analyze the augmentation of TriMedia–CPU64™ – a media-oriented 64-bit 5 issue-slot VLIW processor – with a field-programmable gate array, and assess the potential performance such hybrid achieves when performing media-oriented tasks, e.g., MPEG decoding.

In this chapter, we highlight the initial requirements and freedom degrees of our research activity, that define the dissertation scope. We especially pose three fundamental research questions that are to be answered throughout the presentation. The chapter is organized as follows. The problem overview and the dissertation scope are presented in Section 1.1. Major open questions are enumerated in Section 1.2. The usage of particular words is discussed in the same section. Section 1.3 completes the chapter with an overview of the dissertation.

1.1 Problem overview and dissertation scope

A significant amount of work has been carried out in reconfigurable computing area for more than a decade. Esentially, the research activity connected to media processing domain has mainly focused on two directions:

1. Building a reconfigurable device, which is tuned to stream-processing tasks [43, 33, 72, 37, 101].
2. Augmenting a general-purpose processor, e.g., MIPS, with a reconfigurable core [87, 124, 48, 92, 72, 59, 101].

The first direction fails the full programmability requirement and consequently it is not discussed any longer. Concerning the second research direction, we would like to emphasize that thus far it has been assumed a rather simple general-purpose processor, for which any FPGA-based assist is likely to be of great help. In this dissertation we are concerned with powerfull processors that may not be easily helped with FPGA addition. Moreover, current proposals do not consider the media processing domain extensively; in particular, tasks that do not exhibit instruction-level paralellism, e.g., entropy decoding, or exhibit data- and/or instruction-level paralellism only in certain regions, e.g., pel reconstruction, have not been considered. Assuming a powerful processor (in particular, a *well-engineered VLIW engine* as DeHon mentioned [28]) that is highly optimized to media processing domain, achieving a significant performance improvement within the considered processing domain is much more challenging. For this dissertation, TriMedia–CPU64™ VLIW core, which is itself roughly three times faster than the commercially-available TriMedia-1000 processor, is subject to improvement by augmentation with a reconfigurable core. The initial requirements and freedom degrees of our research activity [81] can be summarized as follows:

1. Investigate the reconfigurable computing paradigm and assess what gains can be expected from it in the framework of a media-oriented VLIW core.

2. Propose an extension of the instruction set architecture that incorporates a reconfigurable functional unit into the VLIW core.

3. Use the TriMedia–CPU64 architecture as an experimental platform for the media-oriented VLIW core.

4. Quantify the TriMedia–CPU64 + FPGA hybrid with realistic performance figures.

Based on these requirements and the available development tools for TriMedia and FPGA, we restrict our dissertation scope as follows.

- Since a processor from the media-oriented VLIW class has to be addressed, we do not analyse superscalar general-purpose processors augmented with multimedia-assist instructions, e.g., MMX-extended Pentium. The FPGA augmentation of such superscalar processors is a parallel research direction that is currently carried out by other members of the Computer Engineering Laboratory at Delft University of Technology [20].

- As mentioned, a primary goal is to augment TriMedia–CPU64 with a reconfigurable core while only minimal changes in the basic architecture, compiler, scheduler, and simulator are needed. Thus, we assume that TriMedia–CPU64 is augmented with a Reconfigurable Functional Unit (RFU) that works synchronously and under the direct supervision of the host processor as any other hardwired functional unit. That is, the RFU reads in and writes back registers as specified by the syntax of the instruction issued by TriMedia instruction decoder. Also, the latency of an operation executed by RFU is deterministic (i.e., it does not depend on the input data), and therefore known at compile time. However, re-entrant or non-re-entrant computing units can be mapped on the RFU as long as these constraints are fulfilled.

- We would like to emphasize that we address only single-processor systems in this dissertation. While the augmentation the TriMedia–CPU64 with an FPGA-based coprocessor working asynchronously with the host processor is an interesting research direction, such approach resembles somehow a dual-processor system. Since the efficiency of a processor–coprocessor system is not necessarily an issue related to the reconfigurable computing paradigm, we will not consider it any further. However, an FPGA-augmented coprocessor can actually be thought of as a host controller (which can be even TriMedia-based, as in the one envisioned to act as a VLD engine [40]) that is augmented with a reconfigurable functional unit. Therefore, our achievements concerning FPGA-augmented TriMedia–CPU64 can easily be used to build an FPGA-augmented co-processor as well.

- A *fine-grain* FPGA (ACEX EP1K100 from Altera) is used as an experimental reconfigurable platform. Although the architectural extension we propose in the dissertation conceptually allows for augmentation with a coarse-grain FPGA as well, the rationale for using a fine-grain device is that we aim to assess the performance of the TriMedia + FPGA hybrid, while a large flexibility in defining circuits on the reconfigurable core is preserved. We would also like to note that the debate ”fine-grain or coarse-grain reconfiguration core” is a research direction by itself that we do not address in this dissertation. Since we envision large silicon area as an implementation commodity, the possible larger area of a fine-grain device with respect to a coarse-grain device is likely not to be a serious restriction [81].

- A hypothetical 4-context FPGA having the architecture of the raw hardware and the context-reconfiguration scheme of an ACEX EP1K100 device is assumed. Although a multiple-context FPGA is not commercially available for the time being, the research activity that has been carried out by the scientific comunity in the last 10 years encourages us to consider multiple-context reconfigurable cores as being implementable in the near future.

- The evaluation of the FPGA-augmented TriMedia–CPU64 performance is carried out within TriMedia–CPU64 media-processing domain. In particular, we focus on MPEG video decompression tasks. As a general view, compression removes redundancy in the signal to be compressed. A decompression task is basically a serial process and, thus, lacks instruction-level parallelism. Since a VLIW processor has to benefit from instruction-level parallelism in order to be efficient, the decompression task is likely to be an intricate function on TriMedia–CPU64. This is the main reason why MPEG decoding is a proper benchmark for evaluating the FPGA-augmented TriMedia–CPU64.

1.2 Open questions and terminology

As indicated earlier, the main idea of the reconfigurable computing paradigm is to define new computing units on a per-application basis, and to dynamically configure them onto FPGA. At the architectural level, a common approach to manage this process is to introduce a new instruction for each portion of the application mapped into FPGA [47, 5]. That is, a new opcode is used for each new reconfigurable operation, which severely restrict the operation to be performed in FPGA.

An architectural extension which requires a single operation code while a relative large number of reconfigurable operations can be encoded within the standard RISC instruction format has been proposed by Razdan in the Programmable RISC (PRISC) project [87]. However, the relative large number of reconfigurable operations (namely 2^{11} = 2048) is achieved at the expenses of a small-width opcode field (thus, only a small number of opcodes are still available to define the hard-wired operations, namely 2^6 = 64), and also of small-width fields for the source and destination registers (thus, only a small number of operands can be encoded, namely 2^5 = 32). A better approach than PRISC is provided by the MOLEN $\rho\mu$–coded processor [117], which trade-offs register encoding for reconfigurable operations encoding. In MOLEN, all the bits which are not used for the opcode point to a microcode address where a microcode routine performs the desired operations. The expenses induced by the MOLEN approach are additional specialized registers and proper `MOVE` instructions between the standard and these additional registers. In this dissertation we use a MOLEN subset to support reconfigurability in TriMedia–CPU64.

Based on these considerations, the following major open question can be posed with respect to media-oriented VLIW processors in general, and TriMedia–CPU64 processor in particular:

1. **What is the minimal set of architectural changes needed for incorporating the reconfigurable array into the TriMedia–CPU64 VLIW core?**

We investigate such question and propose a way to encode a large number of reconfigurable operations while only a single entry in the opcode space is needed. In the same time, we show that the TriMedia–CPU64 instruction format is preserved. Related to the first major open question is:

2. **What is the impact of the architectural changes on the compilation and simulation tool chain of the TriMedia–CPU64 core?**

For historical reasons, the TriMedia–CPU64 development tools contain a retargetable compiler and simulator, which were used during the processor development stages. It is this retargetability attribute of the development tools that proves to be easily adaptable to support the FPGA-dedicated operations, as we will show later on.

Once the mechanism that incorporates the reconfigurable array into the VLIW core is provided, a natural major open question to be posed is:

3. **What is the influence of the reconfigurable array on the TriMedia–CPU64 computing performance?**

To answer to these questions, our research activity calls for a high-level architecture design, and new implementation of the multimedia kernels. Consequently, it includes algorithm research, and VHDL design. As it is shown later on, the augmentation of the TriMedia–CPU64 processor with a field-programmable gate array results in significant performance-wise advantages for a typical set of multimedia kernels.

Before we present the overview of the dissertation, we discuss our usage of particular words and terminology.

Terminology: In the discipline of computer engineering, the term *architecture* is typically used as an abbreviation for *computer architecture,* which is defined as the *conceptual structure, attributes, and behavior of a computer as seen by a machine-language programmer* [6]. A computer, in turn, consists of three major components: the processor that includes a *central processing unit* and a number of on- or off-chip coprocessors, memory, and input/output system. For this thesis, we examine only the design of the central processing unit, which is also commonly referred to as *processor core*. Hence, in this dissertation, we use the term *architecture* as an abbreviation for *processor core architecture* rather than an entire computer if we do not specify otherwise.

Programmable-Logic Devices (PLD) and *Field-Programmable Gate Arrays* (FPGA) are both reconfigurable devices. However, we pre-eminently above all use the term of FPGA hereafter to refer to a reconfigurable device. The higher logic capacity of FPGAs and the efforts of the scientific comunity to augment FPGAs with PLD-like programmable logic in order to make use of both FPGA and PLD characteristics, support our choice for this terminology.

A *reconfigurable processor* is a programmable computing machine working under the reconfigurable computing paradigm. We make a distinction between a *reconfigurable design* and an *FPGA-mapped computing unit*. A *reconfigurable design* is a hardware-software compound that consists of FPGA configuration information defining custom computing units, and software routines including calls to FPGA-mapped computing units.

Finally, we use the term *functional unit* to refer to a hardwired resource within the CPU, such as the arithmetic and logic unit, for which the syntax and semantics of the supported operations are both defined at manufacturing time. In turn, we use the term *reconfigurable functional unit* (RFU) to refer to a functional unit containing a reconfigurable core, for which only the syntax of the custom operations are defined at manufacturing time. The semantics can be defined by the user as needed, and changed at boot and/or run time. We use the term *latency* to refer to the time lag expressed in clock cycles between the issue of an operation and availability of its result. We use the term *recovery* to refer to the minimum number of clock cycles between the issue of successive operations on the same functional unit.

1.3 Overview of dissertation

In the second chapter, we address the *Reconfigurable Computing* paradigm and introduce a formalism based on microcode according to which any custom operation performed by an FPGA-mapped computing unit is executed as a microprogram with two basic stages: `SET CONFIGURATION` and `EXECUTE CUSTOM OPERATION`. Based on the `SET/EXECUTE` formalism, we provide a brief survey of field-programmable custom computing machines, with an emphasize on the reconfigurable microcoded MOLEN processor. The survey we propose is organized as a taxonomy. The two classification criteria we use are: (1) the verticality and horizontality of the microcode, and (2) the availability of an explicit `SET` instruction that is exposed to the programmer. The net effect of this approach is a view on FCCMs at the architectural level, while the implementation and realization details are hidden. In this way, the principal interactions within a processor system are revealed without the need to refer to a particular user environment.

In Chapter 3 we describe the organization of the FPGA-augmented TriMedia–CPU64 processor, which encompasses an FPGA-based Reconfigurable Functional Unit, and a Configuration Unit managing the reconfiguration of the FPGA. Subsequently we propose an extension of TriMedia–CPU64 instruction set architecture, which incorporates support for reconfigurable hardware. The extension is fully compatible with the existing TriMedia–CPU64 instruction format and requires a single opcode entry for all reconfigurable operations. We also present a programming methodology for FPGA-augmented TriMedia, highlighting the high-level C as well as FPGA programming techniques. We complete the chapter with an overview of the TriMedia–CPU64 media processing domain.

The Chapters 4 and 5 address two computational-demanding tasks, which the standard TriMedia–CPU64 is highly optimized for: Inverse Discrete Cosine Transform (IDCT) and Inverse Quantization (IQ). Both tasks exhibit a large data-level parallelism, thus are suitable for SIMD-style processing with TriMedia operations. We first present the implementation details of two FPGA-mapped computing units that support the computation of the mentioned tasks. Then we analize the organization of several software pipeline loops calling the FPGA-mapped units, and assess their performance achieved on the FPGA-augmented TriMedia–CPU64. We show that a significant improvement of 40% for IDCT and 32% for IQ is achieved on FPGA-augmented TriMedia–CPU64 over standard TriMedia–CPU64.

In Chapter 6 we show that significant improvement can also be achieved on FPGA-augmented TriMedia–CPU64 for a task that is strictly sequential, and thus does not exhibit instruction-level parallelism: Entropy Decoding. We first propose a strategy to partially break the data dependency related to variable-length decoding. Then we show that an FPGA-based Variable Length Decoder, which decodes two variable-length symbols per call (VLD-2), leads to the most efficient entropy decoding in terms of instruction cycles. Specifically, between 7.8 and 9.1 cycles are needed per symbol, which translates to 50% improvement over a pure-software implementation.

Chapter 7 analyses TriMedia–CPU64 extended with the Entropy Decoding, IQ, and IDCT reconfigurable designs described in the previous chapters, and assesses the performance improvement such extensions have when performing MPEG2–compliant pel reconstruction. In particular, we consider three computing scenarios for pel reconstruction: the entire (1) macroblock, (2) slice, and (3) picture/frame is fully processed before the next reconfigurable design is launched. By decoding a set of five MPEG-conformance bit-strings on the TriMedia–CPU64 cycle-accurate simulator, we show that processing at slice level is the winner scenario, and a speed-up of 1.4× is achieved on FPGA-augmented TriMedia–CPU64 over standard TriMedia–CPU64.

Chapter 8 presents a reconfigurable design for YCC-to-RGB color space conversion – a computing-intensive video processing task. Strictly speaking, this task is not part of MPEG decoding process. However, since it is always carried out following MPEG decoding, we consider that it is worth to be analized. For this function, we also show that a significant improvement of 44% can be achieved over a pure-software implementation.

Chapter 9 concludes the dissertation summarizing our findings, discussing the main contributions, and suggesting open areas for further research.

Chapter 2

Reconfigurable Computing Paradigm

The ability for providing a hardware platform which can be metamorphosed under software control has established *Reconfigurable Computing* ($\mathcal{RC}$) [38], [68], [119], [69], [46], [45], [60] as an emerging computing paradigm. Typically, a processor working under the $\mathcal{RC}$ paradigm includes a reconfigurable core, which different computing units can be statically or dynamically instantiated on and then activated at application run-time. By taking advantage of the freedom to adapt these computing units to application, the execution of the critical parts of the application is accelerated. With the Reconfigurable Computing paradigm the user can navigate on two dimensions to implement application-geared computing facilities: a *spatial dimension* and a *temporal dimension*. In the *spatial dimension*, tasks which are computational demanding can be efficiently executed by a properly designed FPGA-mapped computing unit. In the *temporal dimension*, a sequential reconfiguration strategy can be used in order to deal with insufficient configurable hardware. In this way, by swapping the configurations in and out of the FPGA upon demand, only the necessary hardware is instantiated at any given time. It is interesting to note that, as opposed to a program of a classical processor which includes only a software image of the algorithm for the static hardware platform, an *FCCM program* includes a hardware image of the computing resources, and also a software image that will run on those resources.

In connection to the spatial and temporal dimensions, we would also like to mention that a mask-programmed gate array (MPGA), a fused- or PROM-based array [46], or an ASIC exhibit only spatiality, as they cannot be reprogrammed by the end user. On the other hand, a microprocessor-based FPGA emulator, such as that mentioned in [107], exhibits at most temporality, as it cannot provide support for adaptive hardware. Thus, we can state that:

> *The computing paradigm, which gives the programmer the freedom to navigate along both spatial and temporal dimensions, can be considered to be reconfigurable computing.*

Due to these considerations, FPGAs which are not SRAM-based cannot be used with the new paradigm. Therefore, we will refer to SRAM-based FPGAs as simple FPGAs hereafter, and will provide additional explanations only when necessary in order to avoid confusions. It is interesting to note that the *in-system* reprogramming capability of SRAM-based FPGAs has been initially considered as a weakness due to the volatility of programming data, but eventually it has been proven to be the key issue in *Reconfigurable Computing*.

Numerous computing machines working under the reconfigurable computing paradigm have been proposed in the recent past. In this chapter we present an overview of the most significant achievements in the Reconfigurable Computing domain. In that sense we introduce a formalism based on microcode, which allows a characterization of the reconfigurable computing domain at the architectural level, while the implementation and realization details are hidden. The chapter is organized as follows. In Sections 2.1 and 2.2 we review the terminology and concepts of the reconfigurable hardware and microcoded engines, respectively. A taxonomy of field-programmable custom computing machines with an emphasize on the MOLEN architectural and programming model is presented in Section 2.3. The last section completes the chapter with closing remarks.

2.1 Reconfigurable hardware – terminology and concept

A device which can be configured *in the field* by the end user is usually referred to as a *Field-Programmable Device* (FPD) [28], [46], [13]. Generally speaking, the constituents of an FPD are *Raw Hardware* and *Configuration Memory*. The function performed by the raw hardware is defined by the information stored into the configuration memory. The field-programmable devices can be classified in two major classes: *Programmable Logic Devices* (PLD) and *Field-Programmable Gate Arrays* (FPGA). Details on each class can be found for example in [14]. Although both PLD and FPGA devices can be used to implement digital logic circuits, we will pre-eminently above all use the term of *FPGA* hereafter to refer to a programmable device. The higher logic capacity of FPGAs and the numerous attempts to augment the FPGA devices with PLD-like programmable logic in order to make use of both FPGA and PLD characteristics, support our choice for this terminology.

Some FPGAs can be configured only once, for example, by burning fuses. Other FPGAs can be reconfigured any number of times, since their configuration is stored in an SRAM. Initially considered as a weakness due to the volatility of configuration data, the reprogramming capabilities of SRAM-based FPGAs led to the new $\mathcal{RC}$ paradigm. By reconfiguring the FPGA under software control, custom computing units can be implemented on-the-fly. This way, computationally-demanding operations can be executed in hardware rather in software.

The huge reconfiguration data rate that is needed to achieve a run-time reconfiguration [11] constitutes the major drawback of the $\mathcal{RC}$ paradigm. Attempts to overcome this drawback led to different reconfiguration strategies which, in turn, induced the name of the major FPGA architectural classes: *Single-Context*, *Multiple-Context*, *Partial Reconfigurable*. In a *single-context* device, the complexity of the circuitry for reconfiguration is kept at a minimum, a global reconfiguration of the array being needed even for changing 1 bit of configuration information. At the expense of an enlarged configuration memory, a *multiple-context* FPGA stores multiple layers of configuration information referred to as *contexts*, only one of them being active at a time. An extremely fast context switch is possible, e.g., by broadcasting a *context identifier* (Context ID) on a global selection bus as depicted in Figure 2.1. In this way, a fast global reconfiguration of the processing elements and interconnection switches [27], [110] or only of the interconnection switches [9] is provided. However, loading a new configuration from off-chip is still limited by the low off-chip reconfiguration bandwidth. Usually, each layer of the configuration memory can be independently written. Thus, the circuit defined by the active configuration layer may continue its execution, while the non-active configuration layers are being reconfigured. In a *partially reconfigurable* device, means for a selective reconfiguration of a context is provided [35], [89], [25], [24], [105], [106]. The highest flexibility in partial reconfiguration is achieved by a random reconfiguration technique, which gives access to the configuration storage space much like a random access memory. Despite of these characteristics, the reconfiguration speed is limited by the narrow interface to the configuration memory (typically 32-bit bus). We would like to mention that the portion of the context that is not being configured may continue its execution.

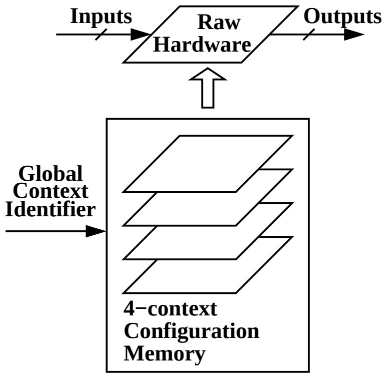


Figure 2.1: **Context ID broadcasting.**

Therefore, as in a multiple-context FPGA, the computation and reconfiguration can be overlapped, but now this happens within the same context.

Four FPGA-mapped circuits are presented in the next chapters: 1-D IDCT, inverse quantizer, variable-length decoder, and color space converter. As it will become relevant throughout the dissertation, there are almost no commonalities among these computing units. Consequently, to switch the active circuit, say, from inverse quantizer to variable-length decoder, a global context reconfiguration has to be carried out. Thus, partial reconfigurability does not bring any advantages in terms of reconfiguration efficiency. This is the rationale for which only the multiple-context reconfiguration is considered hereafter.

By defining and mapping new computing resources onto FPGA, the architecture of the computing machine can be adapted to application. The idea of adaptating the architecture to application is not new. Till the new $\mathcal{RC}$ paradigm has emerged, this adaptation has been performed by emulation on programmable hardware. The code performing the emulation is denoted as microcode. The basics of the microcoded computing machines and the flexibility to adapt their architecture to application are the subject of the next section.

2.2 A microcoded computing machine

Figure 2.2 depicts the organization of a microprogrammed computing machine as it has been described in [86]. In the figure, the following acronyms are used: GPR – General Purpose Registers, ACC – Accumulator, CR – Control Registers, and PC – Program Counter. For such a computer, a microprogram in *Control Store* (CS) is associated with each incoming instruction to be emulated. This microprogram is to be executed under the control of the *Sequencer*, as follows:

1. The sequencer maps the incoming instruction code into a control store address, and stores this address into the *Control Store Address Register* (CSAR).

2. The microinstruction addressed by CSAR is read from CS into the *MicroInstruction Register* (MIR).

3. The microoperations specified by the microinstruction in MIR are decoded, and the control signals are subsequently generated.

4. The computing units perform the computation according to control signals.

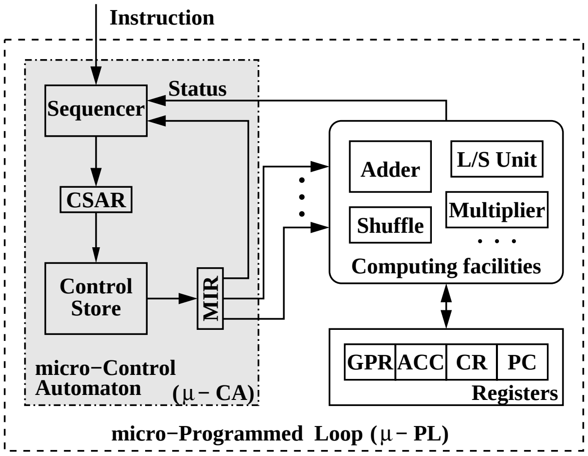


Figure 2.2: **The basic microprogrammed computer.**

5. The sequencer uses status information generated by the computing facilities and some information originating from MIR to prepare the address of the next microinstruction. This address is then stored into CSAR.

6. When an *end-of-operation* microinstruction is detected, a jump is executed to an instruction fetch microroutine. At the end of this microroutine, the new incoming instruction initiates a new cycle of the microprogrammed loop.

In connection to this mechanism, let us assume we have a *Computing Machine* (CM) and its instruction set. An *implementation* of the CM can be formalized by means of the doublet:

$$\mathrm{CM} = \{\mu P\,,\, \mathcal{R}\} \tag{2.1}$$

where μP is a microprogram which includes all the microroutines for implementing the instruction set, and $\mathcal{R}$ is the set of N *computing (micro-)resources* or *facilities* which are controlled by the microinstructions in the microprogram:

$$\mathcal{R} = \{r_1\,,\, r_2\,,\, \ldots\,,\, r_N\} \tag{2.2}$$

Let us assume the computing resources are hardwired. If the microcode[1] is exposed to the *user*, i.e., the instruction set is composed of microinstructions, there is no way to adapt the architecture to application but by custom-redesigning the

---

[1] In this presentation, by *microcode* we will refer to both microinstructions and microprogram. The meaning of the microcode will become obvious from the context.

computing facilities set, $\mathcal{R}$. On the other hand, if the microcode is not exposed to the *user*, i.e., each instruction is emulated by a microroutine, then the architecture can be adapted by rewriting the microprogram μP.

Since the architecture of the microinstructions associated to the hardwired computing resources is fixed, the adaptation procedure by rewriting the microcode could have a limited efficiency: new microprograms are created to emulate the behavior of instructions on fixed (i.e., inflexible) computing resources.

If the resources themselves are microcoded, the formalism recursively propagates to lower levels. Thus, the implementation of each resource can be viewed as a doublet composed of a *nanoprogram* (nP) and a *nano-resource set* ($n\mathcal{R}$):

$$r_i = \{nP\,,\, n\mathcal{R}\}\,, \quad i = 1, 2, \ldots, N \tag{2.3}$$

Now, the rewriting of the nanocode is limited by the fixed set of nano-resources.

At this point, we want to emphasize that the microcode is a *recursive formalism*. The *micro* and *nano* prefixes should be used against an *implementation reference level*[2] (IRL). Once such a level is set, the operations performed at this level are specified by *instructions*, and are under the explicit control of the *user*. Therefore, the operations below this level are specified by *microinstructions*, those on the subsequent level are specified by *nanoinstructions*, and so on.

In connection with Figure 2.2, of particular interest is the number of computing resources each microinstruction controls. In this respect, the microinstructions can be classified by the number of controlled resources according to the following definition:

Definition 2.1 *Given a hardware implementation which provides a number of computing units, the amount of explicitly controlled units during the same "issue" time unit (cycle) determines the verticality or horizontality of the microcode, as follows:*

- **A microinstruction that controls multiple units in one cycle is *horizontal*.**
- **A microinstruction that controls a single unit is *vertical*.**

Finally, we would like to mention that by eliminating instructions that require emulation and by exposing vertical microcode to the programmer is having a RISC architecture. When exposure regards horizontal microcode, it is having a VLIW architecture [118].

---

[2] If not specified explicitly, the IRL is considered as being the level defined by the instruction set. For example, although the microcode is exposed to the *user* in the RISC machines, the RISC operations are specified by *instructions*, rather than by microinstructions.

2.3 Field-programmable Custom Computing Machines

The presence of the reconfigurable hardware (customizable hardware rather than fixed hardware for emulation) opens up new ways to adapt the processor designs. Assuming the resources are implemented on a programmable array, adapting the resources to the application is entire flexible and can be performed on-line. In this situation, the resource set $\mathcal{R}$ can be metamorphosed into a new one, $\mathcal{R}^*$:

$$\mathcal{R} \longrightarrow \mathcal{R}^* = \{r_1^*, r_2^*, \ldots, r_M^*\}, \tag{2.4}$$

and so does the set of associated (micro)-instructions. It is entirely possible that writing new (micro)-programs with application-specific (micro)-instructions may be more effective than with fixed (micro)-instructions.

To have a view on FCCMs at the architectural level, while the implementation and realization details remains hidden, we further introduce a formalism by which an FCCM architecture can be analyzed from the microcode point of view. This formalism originates in the observation that every FPGA-related instruction of an FCCM can be emulated, thus it can be mapped into a microprogram. With this in mind, in the remaining we will provide a survey of numerous FCCMs that have been proposed in the recent past. The survey is organized as a taxonomy of FCCMs, in which the classification criteria are microcode related.

A microcode-based formalism for FCCMs. As we already mentioned, by making use of the FPGA capability to change its functionality in pursuance of a reconfiguring process, adapting both the functionality of *computing facilities* and *microprogram in the control store* to the application characteristics becomes possible with the new $\mathcal{RC}$ paradigm. For the information stored in FPGA's configuration memory determines the functionality of the raw hardware, the dynamic implementation of an instruction on FPGA can be formalized by means of a microcoded structure. Assuming the FPGA configuration memory is written under the control of a *Loading Unit*, the control automaton, the FPGA, and the loading unit may have a Δ arrangement, as depicted in Figure 2.3. The circuits configured on the raw hardware and the loading unit(s) are all regarded as controlled resources in the proposed formalism. Each of the previously mentioned resources is given a special class of microinstructions: `SET` for the loading unit, which initiates the reconfiguration of the raw hardware, and `EXECUTE` for the circuits configured on raw hardware, which launches the custom operations.

In this way, any custom operation of an FCCM can be executed in

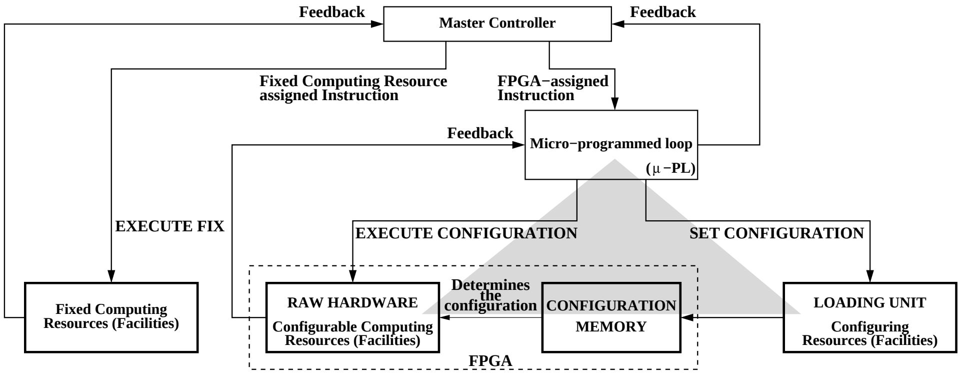

Feedback
Master Controller
Feedback
Fixed Computing Resource assigned Instruction
FPGA−assigned Instruction
Feedback
Micro−programmed loop
(μ −PL)
EXECUTE FIX
EXECUTE CONFIGURATION
SET CONFIGURATION
Fixed Computing Resources (Facilities)
RAW HARDWARE
Configurable Computing Resources (Facilities)
Determines the configuration
CONFIGURATION MEMORY
LOADING UNIT
Configuring Resources (Facilities)
FPGA

Figure 2.3: **The microcode concept applied to FCCMs. The $\boldsymbol{\Delta}$ arrangement.**

a reconfigurable manner, in which the execution is carried out as a microprogrammed sequence with two basic stages: `SET CONFIGURATION`, and `EXECUTE CUSTOM OPERATION`. It is the `SET/EXECUTE` formalism we use in building the taxonomy of FCCMs. The net effect of this approach is to allow a view on an FCCM at the level defined by the reference of the user, i.e., the architectural level, decoupled from lower implementation and realization hierarchical levels. In this way, the principal interactions within a processor system are revealed without the need to refer to a particular user environment. We would like to mention that our approach resembles the *requestor/server* formalism proposed by Flynn [34].

It is worth to specify that only `EXECUTE FIXED OPERATION` microinstructions can be associated with fixed computing facilities, because such facilities cannot be reconfigured. Also, assuming that a multiple-context FPGA [28] is used, activating an idle context is performed by an `ACTIVATE CONFIGURATION` microinstruction, which is actually a flavor of the `SET CONFIGURATION` microinstruction. Hereafter, we will refer to all loading unit(s) and resource(s) for activating the idle context as *Configuring Resources (Facilities)*.

Since an FCCM includes both computing and configuring facilities, the statement regarding the verticality or horizontality of the microcode as stated in Definition 2.1 needs to be adjusted, as follows:

Definition 2.2 *For an FCCM hardware implementation which provides a number of* computing and configuring facilities, *the amount of explicitly controlled* computing and/or configuring facilities *during the same "issue" time unit (cycle) determines the verticality or horizontality of the microcode.*

Therefore, any of the `SET CONFIGURATION`, `EXECUTE CUSTOM OPERATION`, and

`EXECUTE FIXED OPERATION` microinstructions can be either vertical or horizontal, and may participate in a horizontal microinstruction.

Let us set the *implementation reference level* as being the level of instructions in Figure 2.3. In the particular case when the microcode is not exposed to the upper level, an explicit `SET` instruction is not available to the *user.* Consequently, the system performs by itself the management of the active configuration, i.e., without an explicit control provided by *user.* In this case, the *user* "sees" only the FPGA-assigned instruction which can be regarded as an `EXECUTE CUSTOM OPERATION` microinstruction visible to the instruction level. Here, we would like to note that the `EXECUTE FIXED OPERATION` microinstruction is always visible to the *user.* Conversely, when the microcode is exposed to the upper level, an explicit `SET` instruction is available, and the management of the active configuration becomes the responsibility of the *user.*

A Proposed Taxonomy of FCCMs Before introducing our taxonomy, we would like to overview the previous work in FCCM classification.

In [39] two parameters for classifying FCCMs are used: *Reconfigurable Processing Unit (RPU) size* (*small* or *large*) and *availability of RPU-dedicated local memory*. Consequently, FCCMs are divided into four classes. Since what exactly means *small* and what exactly means *large* is subject to the complexity of the algorithms being implemented and the available technology, the differences between classes are rather fuzzy. Also, providing dedicated RPU memory is an issue which belongs to *implementation level* of a machine; consequently, the implications to the *architectural level*, if any, are not clear.

The *Processing Element (PE) granularity*, *RPU integration level* with a host processor, and the *reconfigurability of the external interconnection network* are used as classification criteria in [84]. According to the first criterion, the FCCMs are classified as *fine*-, *medium*-, and *coarse-grain* systems. The second criterion divides the machines into *dynamic* systems that are not controlled by external devices, *closely-coupled static* systems in which the RPUs are coupled on the processor's datapath, and *loosely-coupled static* systems that have RPUs attached to the host as coprocessors. According to the last criterion, the FCCMs have a *reconfigurable* or *fixed* interconnection network.

In order to classify the FCCMs, the *loosely coupling* versus *tightly coupling* criterion is used by other members of the FCCM community, e.g., [60], [124], [56], [101]. In the loosely coupling embodiment, the RPU is connected via a bus to, and operates asynchronously with the host processor. In the tightly coupling embodiment, the RPU is used as a *functional unit*.

We emphasize that all these taxonomies are build using *implementation* criteria. As the user observes only the architecture of a computing machine, classifying the FCCMs according to architectural criteria is more appropriate. Since FCCMs are microcoded machines, we propose to classify the FCCMs according to the following criteria:

- The verticality/horizontality of the microcode.
- The explicit availability of a `SET` instruction.

While the first criterion is a direct consequence of the proposed formalism, several comments regarding the second criterion are worth to be provided. An user-exposed `SET` instruction allows the reconfiguration management to be done explicitly in software, thus being subject to compile-time optimization. The drawback is that a more complex compiler is needed for scheduling the `SET` instruction at a proper location in time. Conversely, if no architectural support for reconfiguration management is provided, i.e., `SET` is not exposed to the user, such management is carried out in hardware at run-time. Since the instruction window for hardware-based scheduling is usually much smaller than that for software-based scheduling, while the `SET` latency is on the order of hundreds or thousands of cycles, run-time managed reconfiguration may lead higher reconfiguration penalty. However, with a hardware-based management, the code compatibility between FCCMs with different FPGA size and reconfiguration pattern can be preserved. In this later case, the user has no concern about the reconfiguration, and the configuration management is an implementation issue, much like the cache management in a conventional processor is. In order to describe the classification process, several classification examples are provided subsequently. We have to mention that, for each system, the implementation reference level has been chosen such that as much FCCM-specific information as possible is revealed.

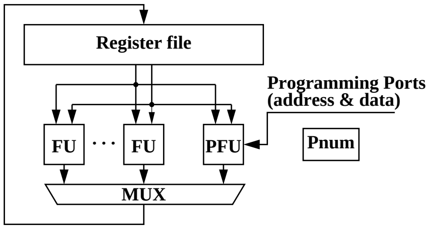


Figure 2.4: **The PRISC architecture**

PRISC [87] is a RISC processor augmented with Programmable Functional Unit (PFU), which is connected to the register file of the host (Figure 2.4). On such programmable unit, application-specific instructions can be implemented. To control the PFU, the fixed-format of 32-bit R-type RISC instructions (two input and one output registers) is used (Table 2.1). When the value of the *opcode* field is equal with `expfu` (*execute PFU*), then a PFU instruction is being called. Further, the 11-bit value of the *Logical PFU* *function* (LPnum) indicates the required PFU configuration, i.e., the function to be executed. Since a single fixed or programmable functional unit can be explicitly controlled during a time cycle, the microcode is vertical.

Instruction format	expfu	rs	rt	rd	LPnum
Number of bits	6	5	5	5	11

Table 2.1: **The PRISC PFU instruction format**

The PFU is implemented on a global-reconfigurable single-context FPGA. The active configuration identifier is stored in a dedicated register (Pnum). If the value of LPnum $\neq$ value of Pnum, a reconfiguration is needed. An exception is then raised, the processor is stalled, and a reconfiguration process is initiated. As the reconfiguration process is not under the control of the user, being managed in hardware, a dedicated instruction for reconfiguration, i.e., a `SET CONFIGURATION`, is not available.

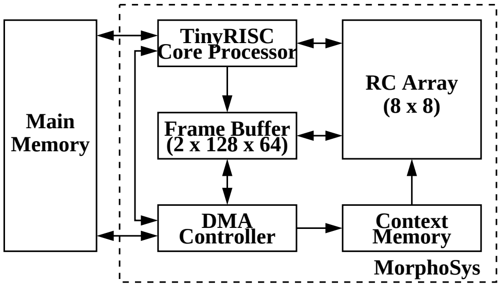


Figure 2.5: **The MorphoSys architecture**

As depicted in Figure 2.5, the *Morpho*ing *Sys*tem (MorphoSys) [101] comprises five major components: a MIPS-like RISC processor core, called TinyRISC, an 8×8 reconfigurable array (RC-Array) of reconfigurable cells and a Context Memory, a Frame Buffer, and a DMA Controller. In addition to the standard RISC instructions, TinyRISC instruction set is augmented with two classes of specific instructions for controlling the MorphoSys components: DMA instructions and RC-Array instructions. *DMA instructions* initiate data transfers between main memory and the Frame Buffer, and also the loading of context words from main memory into the Context Memory. RC-Array instructions specify which of the Context Memory configuration plane is executed and also the mode of context broadcast (vertical or horizontal). An entire line of cells of the 8×8 reconfigurable array is controlled by a single context, but rather in an SIMD-style. Therefore, we classify MorphoSys as a vertical coded machine. Since the user can specify what context to prefetch and what context to be executed, we conclude that an explicit `SET` instruction is provided to the user.

The *PipeRench* coprocessor [16] consists of a set of identical physical *Stripes* which can be configured under the supervision of a *Configuration Controller* at run-time, a *Configuration Memory* storing the stripe configurations, four *Data Controllers* (DC), a *State Memory*, an *Address Translation Table* (ATT), and a *Memory Bus Controller*, as presented in Figure 2.6. A single physical stripe can be configured per cycle; therefore, the reconfiguration of a stripe takes place concurrently with execution of the other stripes. Pipelines of arbitrary length can be implemented on PipeRench. A program for this device is a chained list of configuration words, each of which includes three fields: configuration bits for each virtual pipeline stage of the application, a *next-address* field which points to the next virtual stripe, and a set of flags for the configuration controller and four *Data Controllers*. Therefore, the configuration word is a horizontal instruction. Since the configuration controller handles the multiplexing of the application's stripes onto the physical fabric, the scheduling of the stripes, and the management of the on-chip configuration memory, while the user has only to provide the chained list of the configuration words, we can conclude that there is no user-exposed `SET` instruction.

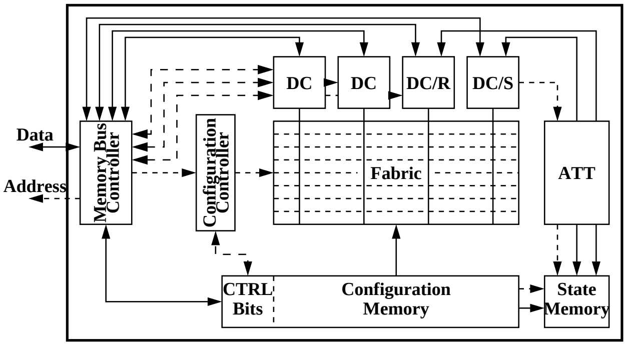


Figure 2.6: **The PipeRench architecture (solid lines are data paths, dashed lines are address and control paths)**

The (re)configuration of the Nano-Processor [123] is initiated by a master unit at application load-time. The system may be used, at least theoretically, in a multi-tasking environment, in which the applications are active or idle. Since no further details whether the user can or cannot manage the reconfiguration are given, we classify this systems as not providing information about an explicit `SET` instruction.

The *Reconfigurable Pipelined Datapath* (RaPiD) is a field-programmable array of coarse-grained functional units (e.g., ALUs, multipliers, registers, RAMs) that is intended to implement deep linear (1-D) pipelines, much like those encountered in DSP applications [33, 26]. The functional units are interconnected in a mostly nearest neighbor fashion, through a set of segmented buses that run over the length of the datapath, as depicted in Figure 2.7.

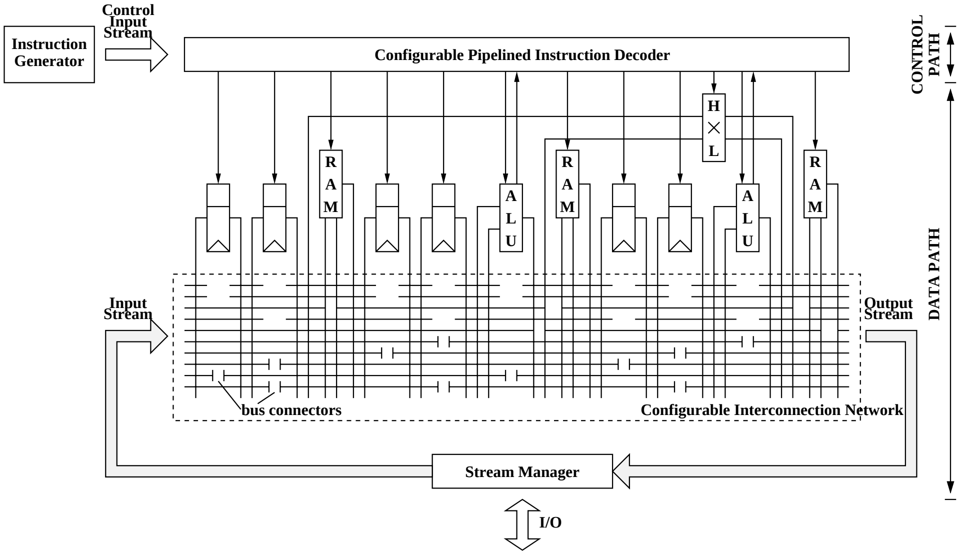


Figure 2.7: **The RaPiD engine – adapted from [33]**

The datapath is controlled using a combination of static and dynamic control signals. The static control signals are defined by RaPiD configuration, and determine the configuration of the interconnection network, and, therefore, the structure of the pipeline. The RaPiD configuration is stored in a configuration memory as in ordinary FPGAs, is loaded at application launch-time, and remains constant for the entire duration of the application. The dynamic control signals schedule the datapath operations over time. These signals are issued by a *control path* which stretches parallel with the datapath, as it is depicted in Figure 2.7. The control path which is also a configurable pipelined structure consisting of segmented buses, has to be configured at application launch-time, too.

The dynamic control is managed outside of the array. The dynamic control signals are inserted at one end of the control path by an *Instruction Generator*, and are passed from stage to stage of the control path pipeline where they are sent to functional units. The control path pipeline plays the role of a MIR with serial loading, where multiple computing units are controlled at a time. Therefore, we classify microcode as being horizontal. A `SET` instruction that can change the static configuration is not claimed by the RaPiD's architects, i.e., no instruction generated by the instruction generator can reconfigure the array. However, the master processor supervizing the RaPiD can initiate a reconfiguration process only at the application launch-time. For this reason, we classify RaPiD as not providing obvious information about an explicit `SET` instruction.

The Colt/Wormhole FPGA [10] is an array of Reconfigurable Processing Units interconnected through a mesh network. Multiple independent streams can be injected into the fabric. Each stream contains information needed to route the stream through the fabric and to configure all RFUs along the path, as well as data to be processed. In this way, the streams are self-steering, and can simultaneously configure the fabric and initiate the computation. Therefore, `SET` is explicit and the microcode is horizontal.

The MOLEN reconfigurable microcoded ($\rho\mu$–coded) processor utilizes a microcoded engine to incorporate architectural support for the reconfigurable hardware [117]. Its symplified organization is depicted in Figure 2.8.

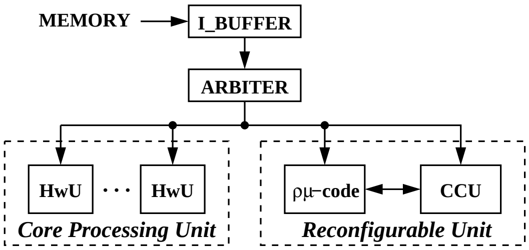


Figure 2.8: **The symplified organization of the MOLEN processor – from [117].**

In this organization, instructions are first fetched from the memory and stored into the instruction buffer (I_BUFFER). Then, the ARBITER performs a partial decoding in order to determine where the instructions should be issued. Instructions that have been implemented in fixed hardware are issued to the Core Processing Unit, where they are executed on Hardwired functional Units (HwU). The instructions that benefit from reconfigurable hardware support are issued to and executed on the Reconfigurable Unit, as described subsequently.

The reconfigurable unit consists of an FPGA-based Custom Configured Unit (CCU) and the $\rho\mu$–code unit. An operation (which can be as simple as an instruction or as complex as an entire piece of code) executed by the reconfigurable unit is divided into two distinct phases: **set** and **execute**. The **set** phase, which is responsible for reconfiguring the CCU hardware and, thus, for enabling the execution of the operation, is performed by executing *reconfiguration microcode* within the $\rho\mu$–code unit. In the **execute** phase, the configured operation is performed by executing *execution microcode* within the same $\rho\mu$–code unit. This approach is generic; by loading different reconfiguration and execution routines, new operations can be implemented. Concerning the MOLEN instruction set architecture, we would like to mention that, in addition to the fixed instructions that are core-dependent, two new instructions are defined to control the described phases: `SET` and `EXECUTE`. Their argument points to the location where the reconfiguration or execution microcode resides, as depicted in Figure 2.9. This way, instead of specifying a new instruction for each new operation (requiring instruction opcode space), microcode locations are simply pointed to.

OPCODE field	argument field
SET	reconfiguration microcode address
EXECUTE	execution microcode address

Figure 2.9: **The MOLEN instruction format.**

Obviously, the `SET` instruction is exposed to the programmer. Concerning the microcode, either a vertical or horizontal microcoded engine are supported by the MOLEN architecture, according to which we classify MOLEN as being either vertical or horizontal machine.

Following the above mentioned methodology, the most well known FCCMs can be classified as follows:

1. Vertically microcoded FCCMs
   - (a) With explicit `SET` instruction: PRISM [7], PRISM-II/RASC [120], [121], RISA$'$ [108], RISA$''$ [108], MIPS + REMARC [71], MIPS + Garp [48], OneChip-98$''$ [56], URISC [31], Gilson's FCCM [36], Xputer/rALU [42], Molen vertically-coded processor [117], MorphoSys system [101].
   - (b) Without explicit `SET` instruction: PRISC [87], OneChip [124], ConCISe [59], OneChip-98$'$ [56], DISC [122], Multiple-RISA [109], Chimaera [47].
   - (c) Not obvious information about an explicit `SET` instruction: Virtual Computer [17], [123], Functional Memory [64], CCSimP (load-time reconfiguration) [91], NAPA [90].
2. Horizontally microcoded FCCMs
   - (a) With explicit `SET` instruction: CoMPARE [92], Alippi's VLIW [5], RISA$'''$ [108], VEGA [58], Colt/Wormhole FPGA [10], rDPA [43], FPGA-augmented TriMedia/CPU64 [95], Molen horizontally-coded processor [117].
   - (b) Without explicit `SET` instruction: PipeRench [16].

(c) Not obvious information about an explicit `SET` instruction: Spyder [54], RaPiD (load-time reconfiguration) [26].

We would like to mention that applying the classification criteria on OneChip-98 machine introduced in [56], we determined that an explicit `SET` instruction was not provided to the user in one embodiment of OneChip-98, while such an instruction was provided to the user in another embodiment. It seems that two architectures were claimed in the same paper. We referred to them as OneChip-98$'$ and OneChip-98$''$. The same ambiguous way to propose multiple architectures under the same name is employed in [108]. For the Reconfigurable Instruction Set Accelerator (RISA), our taxonomy provides three entries (RISA$'$, RISA$''$, RISA$'''$).

2.4 Conclusion

In this chapter we reviewed the classical way of adapting the architecture to application provided by a microcoded implementation of the instruction set. To analize the phenomena inside FCCMs at the architectural level, yet without reference to a particular instruction set, we also proposed a formalism based on microcode, according to which the execution of an FPGA-based operations is carried out in two stages: **set** and **execute**. Based on this formalism, a survey of FCCMs proposed in the literature, which is organized as a taxonomy of FCCMs, has been proposed. Two classification criteria were extracted from the microcoded-based formalism: (1) the verticality/horizontality of the microcode, and (2) the explicit availability of a `SET` instruction. In terms of the first criterion, the FCCMs were classified in vertically or horizontally microcoded machines. In terms of the second criterion, the FCCMs were classified in machines with or without an explicit `SET` instruction. The taxonomy we proposed is architectural consistent, and can be easily extended to embed other criteria.

In the next chapter we will describe the TriMedia–CPU64 + FPGA hybrid, which is a horizontally coded FCCM having the `SET` instruction exposed to the machine-level programmer.

Chapter 3

The ρ–TriMedia Architecture

In order to compute in hardware multimedia tasks with real-time requirements, while preserving the programmability over ASICs, we propose to augment the TriMedia–CPU64 processor with an FPGA that is connected to the host processor as a functional unit – a hybrid that is referred to as ρ–TriMedia. The decision for coupling the FPGA as a functional unit is the result of the constraints defined by the research project [81] that includes minimal modifications of the basic architecture, and the associated compiler and scheduler. In this chapter, we address the first two general research questions connected to ρ–TriMedia, which have been raised in Chapter 1:

1. What are the architectural changes needed for incorporating the reconfigurable array into TriMedia–CPU64 VLIW core?
2. What is the impact of the architectural changes on the compilation and simulation tool chain of the TriMedia–CPU64 core?

The chapter is organized as follows. A brief presentation of the standard TriMedia–CPU64 architecture and the associated programming environment is provided in Section 3.1. Section 3.2 outlines the architecture of the FPGA we use as an experimental reconfigurable core. The architecture of the FPGA-augmented TriMedia–CPU64 is presented in Section 3.3. Section 3.4 outlines the major steps required to implement a task on ρ–TriMedia, while the FPGA-augmented TriMedia–CPU64 programming methodology is described in Section 3.5. Several considerations on the media processing domain, which the ρ–TriMedia performance is evaluated on, are provided in Section 3.6. The final section completes the chapter with some conclusions and closing remarks.

3.1 Standard TriMedia–CPU64 architecture

TriMedia–CPU64 architecture features a rich instruction set optimized for media processing. Specifically, it is a 5 issue-slot VLIW, launching a long instruction every clock cycle [113]. It has a uniform 64-bit wordsize through all functional units, the register file, load/store units, internal and external buses. Each of the five operations in a single instruction can in principle read two register arguments and write one register result every clock cycle. In addition, each operation can be optionally guarded with the least-significant bit of a fourth register to allow for conditional execution without branch penalty. With the exception of floating point divide and square root unit, all functional units have a recovery of 1, while their latency ranges from 1 to 4. The TriMedia core is assumed to support multiple-slot operations, or super-operations [115]. Such a super-operation occupies two or more adjacent slots in the VLIW instruction, and maps to a wider functional unit. This way, operations with more than two arguments and one result are possible. The architecture also supports subword parallelism (SIMD-style) on byte, half-word, or word entities. The current TriMedia–CPU64 organization is presented in Figure 3.1.

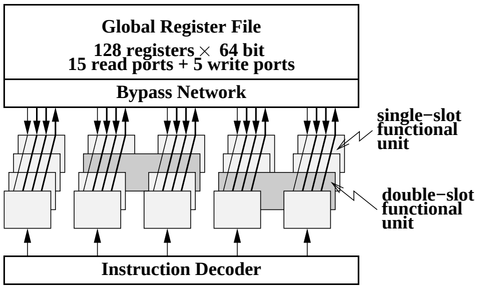

Global Register File
128 registers × 64 bit
15 read ports + 5 write ports
Bypass Network
single−slot functional unit
double−slot functional unit
Instruction Decoder

Figure 3.1: **TriMedia–CPU64 organization – adapted from [113].**

Like a superscalar processor, TriMedia–CPU64 can perform several operations in parallel, and thus can exploit the Instruction-Level Parallelism (ILP) the program exhibits. However, unlike a superscalar processor, these operations are scheduled at compile time. Since many of the applications within the media domain contain instances of data that need similar processing, such as pixels in a stream of digital video, another type of parallelism, called Data-Level Parallelism (DLP) is encountered, too. Thus, it is important to pack adjacent elements into vectors of data to facilitate simultaneous processing by means of a single (SIMD-style) instruction.

To support SIMD-style processing on smaller entities inside a 64-bit word, a special set of operations is implemented. In addition, operations oriented to signal-processing are also supported. All these operations are selected by the programmer at the C–language level by corresponding function calls, as described later this section. The instruction set covers most combinations of the following options:

- a 64-bit word is processed as a vector of 8-, 16-, or 32-bit elements;
- two's complement C-style wrap-around arithmetic or clipping against maximum and minimum integer values on the result is possible;
- the data are interpreted as signed or unsigned values, making a difference for operations that clip their results, and for operations that perform accurate rounding of least-significant bits that would otherwise be lost;
- a complete set of operations, such as add, subtract, min, max, abs, equal, greater, shift, multiply, type conversion, element shuffles, loads, and stores is provided.
- multiplication that returns only the upper half of the muliply result;
- multiply-and-sum to perform an element-wise multiply and summing together all products, in order to return a single integer with the *inner product* value in full precision;
- sum-of-absolute-differences, determining the absolute values of element-wise differences, and summing them together to a single integer;
- special vector shuffle operations for transpose operations on matrices aimed to support 2-dimensional filtering;
- look-up-table operations, where a single vector provides in parallel four short unsigned integers as indices to a table, and four new values are read from the table to constitute the resulting vector.

The TriMedia–CPU64 programming method is a best-of-both-worlds approach to standard C and assembly programming. The C compiler is responsible to detect the ILP that is implicit in program and schedule the extracted machine-level operations. In the same time, in order to explicitly exploit DLP within the C code, the programmer is given direct access to the hardware operations at C level by means of so-called *custom operations* which are recognized by the C compiler. The supported vector types include arrays of signed and unsigned 8-, 16-, and 32-bit integers, and 32-bit floating point vectors. For example, **`vec64ub`** referrs to a 64-bit VECtor containing eight Unsigned Bytes, **`vec64sh`** referrs to a 64-bit VECtor containing four Signed (16-bit) sHort integers, **`vec64uw`** referrs to a 64-bit VECtor containing two Unsigned (32-bit integer) Words, etc. These types are build into the TriMedia–CPU64 compiler, so they can be used directly in C as arguments and return values of the custom operations. An example of a custom operations is

```
vec64sh sh_add( vec64sh x, vec64sh y)
```

that returns the element-wise sum of two 4-element 16-bit signed integer vectors. The compiler then takes care of the correct translation of the program constructs, scoping rules for variables, stack handling, register allocation, and operation scheduling. The programmer can thus concentrate on choosing appropriate data representations, and calling the appropriate custom operations to accomplish the necessary calculations.

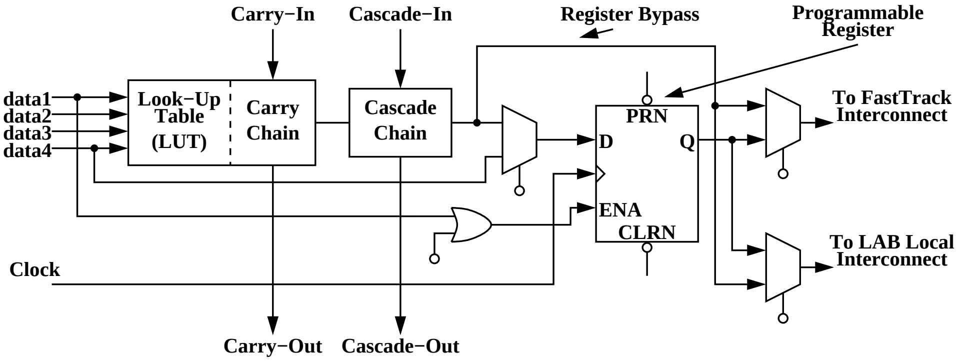
The TriMedia–CPU64 development tools contain a retargetable compiler and simulator, which were used during the development stage of the processor for Design Space Exploration [49]. Mainly, the number, type, and issue-slot allocation of the functional units supporting custom operations were explored. The retargetable tools read in a *Machine Description File*, so that they can adapt their behavior to the particular instance of the processor under test. Among the parameters of the machine description file are the number of instances of a functional unit, in which issue slot the functional unit is available, and which operations the functional unit supports. For most custom operations, the TriMedia–CPU64 compiler only needs to know of their existence, their latency, and the number and types of arguments and results, i.e., their syntax. The compiler does not need to know about the actual function implemented by most of the custom operations, i.e., their semantics. We anticipate and state that the retargetability of the C compiler for custom operations and the lack of the need to know at compile time the semantics of a custom operation provides a native support for the compilation of FPGA-dedicated operations.

3.2 The architecture of the multiple-context FPGA

As described, Field-Programmable Gate Arrays (FPGA) [14] are devices which can be configured *in the field* by the end user. In essence, an FPGA is composed of two constituents: *Raw Hardware* and *Configuration Memory*. The information stored into the configuration memory defines the function performed by the raw hardware. In this dissertation, we assume that a *multiple-context FPGA* is embedded with TriMedia–CPU64 on the same die. Essentially, a multiple-context FPGA [27, 110, 93] has its configuration memory replicated in order to contain several configurations for the raw hardware, which are referred to as *contexts* hereafter. That is, a cache of contexts is available on-chip only one being active at a time. Such a cache allows a context switch to occur on the order of nanoseconds [110].

However, loading a new configuration from off-chip is still limited by low off-chip bandwidth that has an average rate on the order of 50 ns/byte [21].

For experimental purpose, we assume that at most four contexts can be stored simultaneously on the FPGA chip. Our assumption does not violate the general accepted figure regarding multiple-context FPGAs – see for example [28, 110]. Since a multiple-context FPGA is not commercially available for the time being, we also assume a **hypothetical FPGA** having the architecture of the raw hardware and context reconfiguration scheme identical to those of a (single-context) ACEX 1K device from Altera [22]. Our choice could allow future single-chip integration, as both ACEX 1K FPGA and TriMedia families are manufactured in the same TSMC technological process. Briefly, an ACEX 1K device contains an array of Logic Elements or Cells, a number of Embedded Array Blocks (EAB), each EAB being a RAM block with 8 inputs and 16 outputs, and an interconnection network. The structure and timing microparameters (speed grade -1) of the ACEX 1K logic element are presented in Figure 3.2. Notably, a fast carry-chain having a propagation time of only 0.2 ns per logic element facilitates the implementation of arithmetic functions, such as adders, counters, and comparators.



- LUT-input to LUT-output delay (t_{LUT}): max. 0.6 ns
- Carry-input to LUT-output delay (t_{CLUT}): max. 0.5 ns
- LAB local interconnect to Carry-output delay (t_{CGEN}): max. 0.5 ns
- Carry-input to Carry-output delay (t_{CICO}): max. 0.2 ns
- Cascade-input to Cascade-output delay (t_{CASC}): max. 0.8 ns

Figure 3.2: **ACEX 1K logic element structure and timing microparameters.**

The reconfiguration of an ACEX 1K device can be performed according to the common *Passive Parallel Asynchronous* scheme, in which a master unit treats the

FPGA as memory [22]. That is, the master unit drives data to the FPGA serially, one word at a time. There are no partial reconfiguration capabilities for the considered device; thus, a global reconfiguration of a context is required even for changing 1 bit of its configuration data. Since we envision that circuits without many commonalities are to be configured on the programmable array, this limitation is not a serious restriction.

Subsequently, we also assume that a context can be configured only when it is not active. Thus, the active context can continue its operation while the idle contexts are being reconfigured. As mentioned, the FPGA context switching occurs on the order of nanoseconds. Trimberger *et al.* [110] claim that a context switching can be completed in 30ns for a 20×20 array of Logic Blocks. For an ACEX EP1K100 device which includes about 4,992 LUTs and 12 EABs, that is, for an FPGA which is more than 15 times larger, we make a conservative assumption and consider that the context switching penality is on the order of 500 ns. The rationale behind the assumption that the reconfiguration time increases linearly with the FPGA size is related to power consumption, which can become a concern with large FPGAs as identified by Trimberger *et al.* [110].

The architectures of standard TriMedia–CPU64 and FPGA have been presented. Subsequently, we introduce the architectural extension for the TriMedia-CPU64 that incorporates support for the reconfigurable hardware.

3.3 ρ–TriMedia – the FPGA–extended TriMedia–CPU64

In this section we propose to augment the TriMedia-CPU64 processor with a Reconfigurable Functional Unit (RFU) consisting of a multiple-context FPGA core and its associated Controller, and a Configuration Unit (CU) managing the reconfiguration of the FPGA. This hybrid will be referred to as ρ–TriMedia hereafter. Both RFU and CU are embedded into TriMedia as any other hardwired functional units, i.e., they receive instructions from the instruction decoder, read their input arguments from and write the computed values back to the register file, as depicted in Figure 3.3. Since the latency and recovery of the hardwired operations are not embedded into the VLIW core controller but rather into the functional units themselves, only minimal modifications of the basic architecture, and the associated compiler and scheduler are required.

In order to use the RFU, the user is provided a kernel of new instructions: `SET_CONTEXT`, `ACTIVATE_CONTEXT`, and `EXECUTE`. This kernel constitutes the extension of the TriMedia instruction set architecture we propose. Loading context information into the FPGA configuration memory is performed by the Con-

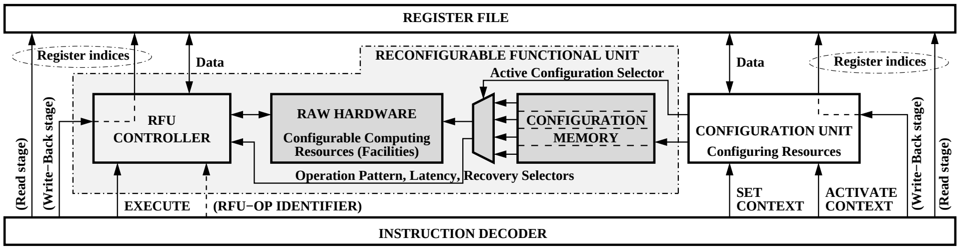


Figure 3.3: **The TriMedia-CPU64 VLIW core extension.**

figuration Unit under the command of a `SET_CONTEXT` instruction, while the `ACTIVATE_CONTEXT` instruction swaps the active configuration with an on-chip idle one. `EXECUTE` instructions launch the operations performed by the FPGA-mapped computing units [100]. With these instructions, the user is given the freedom to define and use any computing unit subject to FPGA size and TriMedia organization.

Uploading configuration information to the CU is performed under the command of a double-slot instruction issued on Slot pair 1+2:

SET_CONTEXT (context) Rs1, Rs2, Rs3 → Rd

where `context` is the fourth (immediate) argument specifying the context to be reconfigured, while the registers Rs1, Rs2, and Rs3 contain 192 bits of configuration information. If the instruction completes successfully then the register Rd contains 0, otherwise it contains an error code. For example, if an attempt to reconfigure the active context is made, the instruction has no effect on the configuration and returns 1.

Subject to the architecture of the FPGA, the configuration information being uploaded by `SET_CONTEXT` instructions can be interpreted in different ways before it is sent to the RFU. Thus, different configuration patterns can be supported. For example, assuming an FPGA has partial reconfiguration capabilities (e.g., XC6200 family from Xilinx [25]), or incorporates means to reconfigure only the cells that are different from the current configuration (e.g., AT6000 family from Atmel [23]), complex reconfiguration patterns can be generated by a microprogrammable CU [117]. This case, the `SET_CONTEXT` instructions may also upload a *reconfigurable microprogram* to the CU, giving the user a large flexibility to reconfigure the FPGA.

However, for a global-reconfigurable FPGA with a serial context reconfiguration scheme, which in fact we assume in our subsequent experiment, the CU can be as simple as a parallel-to-serial converter. As mentioned in Section 3.2, the average latency for loading new FPGA configuration information from off-chip is about 50 ns/byte, that is 10 cycles/byte. Since the `SET_CONTEXT` instruction places 192 bits = 24 bytes on the CU at a time, it has a latency of 240 cycles. For an EP1K100 FPGA, which has a configuration file of 1,337,000 bit [22], 6,964 `SET_CONTEXT` instructions or 6,964 $\times$ 240 = 1,671,360 cycles are needed to completely reconfigure a context.

For FPGA context switching, a single-slot instruction issued either on Slot 1 or Slot 2 is provided:

ACTIVATE_CONTEXT (context) → Rd

where `context` is an immediate argument specifying the context that is being activated. If the context is already active, the instruction has no effect. An attempt to activate a context prior to its complete reconfiguration has also no effect on the active context, and is signaled by loading an error code into Rd. If the activation completes successfully, the Rd contains 0. Given the fact that the FPGA context switching penality is 500 ns (Section 3.2), the `ACTIVATE_CONTEXT` instruction has a latency of 100 TriMedia@200 MHz cycles.

Conceptually speaking, computing units of user-definable computing pattern[1], latency, recovery, and slot-assignment[2] can be configured on RFU. Thus, the RFU can act as five independent single-slot functional units each of them executing a different custom operation, a mixture of single- and multiple-slot functional units, or even a five-slot functional unit. In all these situations, the RFU may receive `EXECUTE` instructions issued on any of the five TriMedia slots, and use all 10 read and 5 write ports of the register file per call.

In connection to the FPGA-augmented TriMedia implementation, we would like to note that the flexibility in defining slot-width and slot-assignments for RFU-mapped operations determines the implementation cost. For example, assuming the maximum freedom degree in defining slot-assignments for RFU operations, a separate RFU controller has to be placed on each issue slot. In addition, the TriMedia instruction decoder has to be able to decode `EXECUTE` instructions on each of the five issue slots. Moreover, since only single- and double-slot operations are currently supported by the compiler and scheduler for the time being, the toolchain has to be modified to support 3-, 4-, and 5-issue slot operations. However, we would like to note that the later requirement is likely not to pose serious difficulties.

Although the maximum flexibility in defining RFU-based operations may be of theoretical value, it is not of practical relevance in the context of our current

---

[1] i.e., the operation slot-width and the number of input and output registers.

[2] i.e., the issuing slot(s) that the computing facility is sensitive to.

investigations. As one can notice in the next sections, all the RFU operations for the considered media-processing domain can be implemented with single-, double-, and triple-slot operations. For this reason, in this dissertation we consider only a particular instance of FPGA-augmented TriMedia, in which only a single-slot instruction on Slot 1, a double-slot instruction on Slot pair 1+2, and a triple-slot instruction on Slot group 1+2+3 can be issued to the RFU.

For each of the single-slot, double-slot, and triple-slot RFU instructions, a separate operation code is allocated: `EXECUTE_1`, `EXECUTE_2`, and `EXECUTE_3`, respectively. In all cases, the standard TriMedia–CPU64 instruction format is preserved: the opcode is a 9-bit field, and each and every source and destination registers is specified by a 7-bit field. Up to two inputs and one output, four inputs and two outputs, and six inputs and three outputs can be specified by the single-, double-, and triple-slot instructions, respectively:

EXECUTE_1 Rs1, Rs2 → Rd1
EXECUTE_2 Rs1, Rs2, Rs3, Rs4 → Rd1, Rd2
EXECUTE_3 Rs1, Rs2, Rs3, Rs4, Rs5, Rs6 → Rd1, Rd2, Rd3

The `EXECUTE` instructions are generic, since their semantics can be redefined. By reconfiguring the raw hardware with a sequence of `SET_CONTEXT` and `ACTIVATE_CONTEXT` instructions, followed by issuing an `EXECUTE` instruction, any new user-defined operation subject to FPGA size and TriMedia organization can be launched into execution, while only a single entry in the opcode space is needed to encode the `EXECUTE` instruction. Since all the fields in the `EXECUTE_1` instruction format except for the opcode field encode the input and output registers, there are no provisions for additional encoding. Thus, only a single operation per context can be encoded within `EXECUTE_1`. That is, if a different single-slot operation is to be launched, then a reconfiguration of the raw hardware must be carried out beforehand by `SET`/`ACTIVATE` instructions. However, as we describe subsequently, more operations per context can be encoded within a multiple-slot `EXECUTE` instruction. This may reduce the number of reconfigurations when a large FPGA is available.

In standard TriMedia–CPU64, only one of the *opcode* fields in a multiple-slot instruction defines the operation, all the others being set `NOP`s (Fig. 3.4). By using these unused fields as an argument for the RFU OPCODE (Fig. 3.5), a large number of RFU operations can be encoded, while only a single entry for the `EXECUTE` instruction needs to be allocated in the opcode space. Assuming, for example, a double-slot operation, the 9-bit additional opcode (which is subsequently referred to as an `RFU–OP IDENTIFIER` or simple `RFU–OP–ID`) can specify 512 different operations.

We would like to mention that the two parts of the double-slot operation are

Slot 1		**Slot 2**		**3, 4, 5**
HW OPCODE	**src. & dest.**	**NOP**	**src. & dest.**	**...**

Figure 3.4: **The hardwired double-slot operation instruction format.**

Slot 1		**Slot 2**		**3, 4, 5**
RFU OPCODE	**src. & dest.**	***RFU-OP ID***	**src. & dest.**	**...**

Figure 3.5: **The RFU double-slot instruction format.**

decoded separately, and only when the first part specifies an `EXECUTE_2` opcode, the second opcode is interpreted as an `RFU–OP IDENTIFIER`, and thus decoded locally at the RFU by the RFU controller. This way, an RFU super-operation does not create pressure on the instruction decoder, neatly fits in the existing instruction format, fits the existing connectivity structure to the register file, and hence requires only a little hardware overhead.

The RFU controller itself can be a simple decoder for the `RFU–OP–ID` field, a finite-state machine having the `RFU–OP–ID` as argument, or even a *microcoded* engine for which the `RFU–OP–ID` points to the address of a micro-routine [117]. In the later case, the microcode within the RFU controller becomes part of the RFU configuration, and, therefore, subject to reconfiguration by means of `SET_CONTEXT` and `ACTIVATE_CONTEXT` instructions.

Due to TriMedia organization constraints, the RFU operands, i.e., the operation's input and output registers, are encoded within the TriMedia instruction set. That is, if new operands are required from, and/or additional results have to written back to the register file, a new `SET_CONTEXT` or `EXECUTE` instruction has to be issued (`ACTIVATE_CONTEXT` requires only a single immediate argument, as previously described). Thus, the FPGA reconfiguration and execution of FPGA-mapped operations are performed within the TriMedia instruction set, and both the *reconfiguration microcode* and *execution microcode* can be thought of as being exposed to the instruction-level programmer. This is distinct from the $\rho\mu$-coded MOLEN processor [117] that does not encode arguments and destinations within the `SET` and `EXECUTE` instructions.

In connection to the `EXECUTE` instructions, we would like to emphasize that their semantics, number of operands (or pattern), latency, and recovery are all explicitly user-definable, while the slot-width is defined implicitly by the particular `EXECUTE_1`, `EXECUTE_2`, or `EXECUTE_3` opcode. Therefore, it is the responsibility of the programmer to augment the Machine Description File with appropriate information [80]. Assuming a user-defined `IDCT` operation, a way to specify such information is to annotate the source code, as depicted in Figure 3.6. At the machine implementation level, these parameters are set by means of *Selectors*, which become part of the RFU configuration, as presented in Figure 3.3. A different {*operation pattern*, *latency*, *recovery*} set can be defined for each `RFU–OP–ID`. With such mechanism, an `EXECUTE` instruction is trully generic, and the programmer is able to adjust its behavior as needed.

```
                  EXECUTE_2 <IDCT>   Rs1, Rs2 → Rd1, Rd2

.rfu_op_id        IDCT   IDCT_OP_ID    ; specifies the IDCT OP_ID
.pattern          IDCT   2i+2o         ; IDCT has 2 input- and 2 output-registers
.latency          IDCT   7             ; specifies the IDCT latency
.recovery         IDCT   2             ; specifies the IDCT recovery
```

Figure 3.6: **The syntax and annotation code for a user-defined IDCT operation.**

Finally, several considerations about the latency of an RFU-mapped computing unit are worth to be provided. Due to realization constraints, the RFU is likely to be located far away from the Register File (RF) in the TriMedia-CPU64 floorplan. The immediate effect is that there will be large delays in transferring data between the RFU and RF, and the RFU will not benefit from bypassing capabilities of the RF [113]. Consequently, *read* and *write back* cycles have explicitely to be provided. In such circumstances, the latency of an RFU-mapped computing unit is composed of 1 cycle for reading the input arguments from register file, the number of cycles corresponding to the computation delay on FPGA, and 1 cycle for writing back the results to the register file. Since an RFU call is quite expensive, it is recommended to minimize the number of RFU calls, i.e., computing units which can perform complex operations are recommended to be configured on the RFU.

To give an indication of the programming complexity on ρ–TriMedia, in the next section, we outline the flowchart for task implementation on ρ–TriMedia.

3.4 ρ–TriMedia development flowchart

Given an application, the programming flowchart that the programmer has to pursue in order to make use of ρ–TriMedia at best can be summarized as follows.

1. Profile the code and identify (large) functions that worth benefiting from reconfigurable hardware support, thus accelerated.

2. For each of such function:
    - write VHDL code, synthesize, place and route on FPGA;
    - write C-level code that emulates the VHDL code in order to augment the TriMedia-CPU64 cycle-accurate simulator[3];
    - rewrite the initial C-level code for standard TriMedia-CPU64 in order to call the FPGA-mapped computing units.
3. Compile, simulate, and extract the performance figures.
4. Go to Step 1 if the performance figures are not satisfactory yet.

Therefore, as opposed to the standard TriMedia-CPU64 user, the ρ–TriMedia user should be both a hardware and software designer. That is, the ρ–TriMedia user has to write VHDL code, compile it, and map it on the particular FPGA core. Then, the ρ–TriMedia user has to perform the hardware-software integration, and call the new FPGA-based computing units within software routines.

We would like to mention that the process of programming for ρ–TriMedia is semiautomatic, and several iterations through the stages mentioned above might be needed to achieve a good program. Designing a fully automatic programming environment is far beyond the scope of the dissertation. Consequently, we leave it for further work.

3.5 ρ–TriMedia programming methodology

Instead of building an automatic tool we provide a programming methodology, which is aimed to help the programmer to take the advantage of the new reconfigurable functional unit. Next, we describe the ρ–TriMedia C-level programming model. According to this model, the programmer is able to call FPGA–based operations by means of *custom operations*. Then, we address programming for reconfigurable hardware and outline some issues connected to mapping on FPGA. We complete the section with a set of recommendations for code developing.

ρ–TriMedia programming model. For the architectural extension to be effective, the user needs a programming model. At the C-level, we propose to call RFU-based functions by defining additional *custom operations*. As mentioned,

---

[3] If the simulator cannot perform C and VHDL co-simulation.

since custom operations are already widely used by the standard TriMedia, granting the C-level programmer a direct access to hardware operations [80], only minimal modifications of the standard compilation toolchain are required. In addition to the standard definition of custom operations, the programmer has also to specify the pattern, latency, recovery and slot assignment for the new RFU-based operation, as well as the `RFU_OP_ID` if the operation is multiple-slot. A common way to specify these parameters is by using pragmas [73]. An example of a C-level code calling RFU is presented in Algorithm 1.

Algorithm 1 An example of a C-level code calling the RFU

```
#include <stdio.h>
#include "trimedia.h"

vec64sh Rx_input, Ry_input, Rz_output, Rw_output;

int main( void) {
  ...
  #pragma RFU_OP (pattern=2,2i+2o,latency=16,recovery=2)
  IDCT( &Rz_output, &Rw_output, Rx_input, Ry_input);
  ...
}
```

In this example, Rx_input, Ry_input, Rz_output, and Rw_output specify each a vector of four 16-bit signed integers. Thus, IDCT operation reads in eight 16-bit signed integers and computes eight 16-bit signed integers. Assuming a standard TriMedia processor, the compiler does not recognize the pragma; therefore, the IDCT call is compiled into a function call, and the portability of the C-level code is ensured. Considering an FPGA-augmented TriMedia, the compiler does recognize the `RFU_OP` pragma, and generates a machine-level instruction having the `EXECUTE` opcode, and `IDCT` as `RFU–OP–ID`. As specified by the *pattern* field, this instruction designates a double-slot operation having two register inputs and two register outputs. Since the pool of RFU-based functions which can be coded is quite large ($2^9 = 512$ for a double-slot RFU instruction, $2^{18} = 262,144$ for a triple-slot RFU instruction, etc.), a large flexibility to allocate a different `RFU–OP–ID` for each FPGA-mapped computing unit is available. This task is done by the compiler. Based on the *pattern*, *latency*, and *recovery* fields, the `EXECUTE_2` $<$ `IDCT` $>$ operation is scheduled on slot pair 1+2 as any other hardwired operation having a latency of 16 and a recovery of 2.

TriMedia is geared to the *media* domain, in which large sets of data are manipulated in a repetitive fashion, basically around loops. Since complex long-latency operations are envisioned to be configured on the RFU, the Instruction-Level Parallelism (ILP) within a single loop iteration containing RFU calls is expected to diminish. In order to expose to the compiler the ILP that is still available across the loop boundary, two strategies can be employed: (1) *loop unrolling and/or grafting*, and the more efficient (2) *software pipelining* (which can be thought of as infinite loop unrolling). While the first strategy trade-offs code size (and, thus, the overhead associated to the additional instruction cache misses) with ILP [4], the second strategy significantly increases ILP while maintaining about the same code size. Moreover, due to its smaller code size with respect to loop unrolling, software pipelining allows the programmer to control the real time response with higher granularity [3]. Indeed, since an *event* (e.g., a pending interrupt) is handled only when an *interruptible jump* is encountered, the cycling interruptible jump scheduled at the end of the loop is executed more often in a software pipelined loop than in an unrolled/grafted loop. In addition, the programmer is able to control the number of uninterrupted executions of the unrolled/grafted or software pipeline loop by forcing a jump to be *interruptible* or *non-interruptible*. Assuming for example the software pipelining strategy, this control can be performed by declaring the loop as *atomic*, as presented in Algorithm 2. The TriMedia compiler will recognize the pragma *TCS_atomic*, and generate *non-interruptible jump*s for all the jumps in the IDCT_function, with the exception of the *return* jump.

Despite of its advantages, the software pipelining strategy cannot be easily employed for the time being, since software pipelined machine-level code is more difficult to generate than its unrolled/grafted counterpart. Indeed, the current TriMedia scheduler uses the *decision tree* as a scheduling unit [50]. Thus, all operations return their computed values in the same decision tree that they are launched, even though the TriMedia architecture does not forbid the contrary. Since generating software pipeline loops essentially requires for returns of the computed values beyond the decision tree boundary, decision tree-based scheduling is the major limiting factor in generating tight software pipelined loops containing long-latency operations. However, the loop containing RFU operations may be very simple and symetrical (see, for, example [95]); thus, programming in assembly is indeed feasible despite of the fact that the host is a complex VLIW processor. Conversely, we will use C-level loop unrolling where programming directly in assembly proves to be too complex for generating tight loops.

FPGA mapping issues. To efficiently map circuits on FPGA, the programmer should take advantage of the FPGA architectural features. Since synthesizing behavioral VHDL code might not utilize all the FPGA architectural features, the re-

Algorithm 2 An example of a deep software pipeline calling long-latency RFU-based operation

```
#include <stdio.h>
#include "trimedia.h"

vec64sh Rx_input, Ry_input, Rz_output, Rw_output;

#pragma TCS_atomic
IDCT_function( vec64sh Rx_input, vec64sh Ry_input,
                    vec64sh Rz_output, vec64sh Rw_output) {

  for( i=0; i<N; i++) {
    ...
    #pragma RFU_OP (pattern=2,2i+2o,latency=16,recovery=2)
    IDCT( &Rz_output, &Rw_output, Rx_input, Ry_input);
    ...
  } /* the looping jump is translated into a ’non-interruptible jump’ */

} /* the ’return’ is translated into an ’interruptible jump’ */

int main( void) {
  ...
  IDCT_function( Rx_input, Ry_input, Rz_output, Rw_output);
  ...
}
```

sulting circuits usually do not exhibit satisfactory latency and reconfigurable hardware usage. Therefore, a hand-crafted VHDL design, which exposes to the synthesizer those computing primitives efficiently supported by the considered FPGA, is needed.

As it can be observed in Figure 3.2, one of the architectural features of the ACEX 1K family is the fast carry-chain path, which is dedicated for implementing arithmetic functions such as adders, counters, and comparators. This feature will be particulary useful in Chapters 4, 5, and 8 when custom implementations for multipliers-by-constant are considered. Another important architectural feature of the considered FPGA is the fast cascade-chain path, which is dedicated for implementing logic functions that have a wide fan-in. Adjacent LUTs are used to compute portions of the function in parallel, while the cascade chain serially

Table 3.1: **Performances of several reduction modules for ACEX EP1K100 FPGA (Speed Grade -1).**

Reduction module		Performance f_{max} – MHz		
		Leonardo-Spectrum(1)	MaxPlus-II WYSIWYG (2)	MaxPlus-II FAST (3)
Two-operand	16-bit adder	136	140	140
Three-operand		104	107	117
Four-operand		104	103	109
Five-operand		84	81	81
Six-operand		84	76	76
Two-operand	24-bit adder	112	114	114
Three-operand		89	94	94
Four-operand		89	86	90
Two-operand	28-bit adder	102	103	103
Three-operand		83	85	83
Four-operand		83	77	81
Two-operand	30-bit adder	98	102	102
Three-operand		88	93	91
Five-operand	3-bit adder	108	147	138
Six-operand		108	131	121
Seven-operand		108	128	116
Five-operand	4-bit adder	105	126	113
Six-operand		105	126	107
Seven-operand		105	111	114
Five-operand	6-bit adder	101	113	107
Six-operand		101	97	105
Seven-operand		101	94	97
Three inputs	Dadda's population counter	*231*	*250*	*250*
Four inputs		*228*	*250*	*250*
Five inputs		155	175	169
Six inputs		155	*188*	*188*

the intermediate values. We anticipate and state that this architectural feature can be used to compute the number of zero-leading bits in a string, as we describe in Chapter 6.

Based on the ability to implement fast ripple-carry adders and wide fan-in logic functions, the programmer has now to focus on mapping more complex functions. For example, one of the most common circuit in the media domain is the multiplier, which an optimization of the partial product matrix reduction has to be carried out for. In connection to the partial product matrix, measured performances of several reduction modules on ACEX 1K are presented in Table 3.1. All the figures correspond to synchronous designs, i.e., both inputs and outputs are registered. The estimations have been obtained by compiling VHDL code with Leonardo Spectrum™ from Exemplar, followed by placing and routing with MAX+PLUS II™ from Altera. Since the maximum operating frequency for the ACEX 1K is 180 MHz for the time being, we make a conservative assumption and consider the figures typed in italics as being too optimistic, although they have been generated by software tools. The following settings of the software tools have been used: (1) Leonardo-Spectrum™: *Lock LCELLs: NO*, *Map Cascades: YES*, *Extended Optimization Effort*, *Optimize for Delay*, *Hierarchy: Flatten*, *Add I/O Pads: NO*; (2) MaxPlus-II: *WYSIWYG*, *Optimize = 10 (Speed)*; (3) MaxPlus-II: *FAST*, *Optimize = 10 (Speed)*.

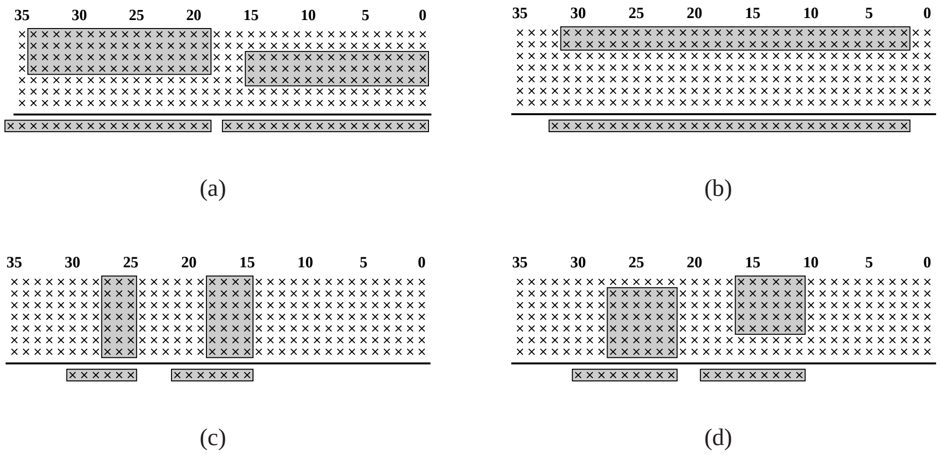
A pipelined FPGA implementation of a computing unit having a recovery of 1 implies that the FPGA clock frequency is equal with the TriMedia clock frequency. Nowadays, the upper limit of the clock frequency in TriMedia family is around 300 MHz, while the maximum clock frequency for ACEX 1K FPGA family is 180 MHz. Thus, a hypothetical computing unit having a recovery of 1 is not a realistic scenario, and a recovery of 2 or more is mandatory for the time being. Hereafter, we assume a recovery of 2 for the FPGA-based computing units, which translates into an FPGA cycle time to TriMedia cycle time ratio of 2. Our assumption does not violate the general accepted ratio figure of FPGA-mapped logic versus hardwired logic [28]. Considering a TriMedia running at 200 MHz, an FPGA–mapped pipeline will work with a clock frequency of 100 MHz.

In order to implement a pipeline at 100 MHz, reduction modules which can run at 100 MHz or more should be considered. These modules are summarized below:

- Horizontal reductions of three, or four 16-bit lines to one line (Fig. 3.7(a)).
- Horizontal reduction of only two 30-bit lines to one line (Fig. 3.7(b)).
- Vertical reductions of three or four 7-bit columns to one line (Fig. 3.7(c)).
- Vertical reductions of six 5- or 6-bit columns to one line (Fig. 3.7(d)).

Based on these considerations, we can state that, as a rule of thumb for the ACEX 1K family, the reduction is more efficiently horizontally than vertically. This is why the appropriate reduction strategy (ripple-carry or carry-save) should be re-evaluated with respect to a particular FPGA architecture.

Several recommendations for code developing. According to the methodology we propose, the programmer can obtain better results if he/she follows a set of



(a) (b)

(c) (d)

Figure 3.7: **100 MHz reduction modules on Altera's ACEX EP1K100 (speed grade -1) FPGA.**

recommendations for code development. These recommendations are enumerated subsequently.

1. Use the highest clock frequency allowed by FPGA in order to build deep pipelines for stream processing tasks. For such tasks, the recovery of the FPGA–mapped computing units is usually more important than latency. Examples are provided in Chapters 4, 5, and 8.
2. Minimize the latency rather the recovery when a sequential task is to benefit from reconfigurable hardware support, e.g., variable-length decoder. An example is provided in Chapter 6.
3. Try to program directly in assembly when the software pipeline loop is simple and symetrical in order to generate tight software pipeline loops. As we show subsequently, this approach is indeed possible although the computing engine is a complex VLIW processor. Examples are provided in Chapters 4, 5, and 8.
4. Use C-level custom operations to lauch FPGA–based operations as described earlier this section if programming in assembly is too difficult or time-consuming.

To assess the performance of the FPGA-augmented TriMedia–CPU64 hybrid, we will address the TriMedia processing domain, and propose a number of reconfigurable designs for several multimedia-oriented kernels. To make the presentation self-consistent, several issues concerning the MPEG standard, which is extensively addressed in the dissertation, are provided in the next section.

3.6 TriMedia–CPU64 processing domain

The application domain of the TriMedia–CPU64 processor includes multimedia-oriented tasks, e.g., video coding and processing, audio coding, data communications, and graphics [88]. A typical mode of operation for a TriMedia-based system is to serve as a video-decompression engine on a PCI card in a PC. Video decompression begins when the PC operating systems hands the TriMedia a pointer to compressed video data. The most common standard for video compression is MPEG, for which we provide a brief outline.

MPEG standard. The MPEG standard [70, 44] uses a large number of compression techniques to decrease the amount of data. Data compression is the reduction of redundancy in data representation, carried out to decrease data storage requirements and data communication costs.

A typical video codec system is presented in Figure 3.8 [103, 70]. The lossy source coder performs filtering, transformation (such as Discrete Cosine Transform (DCT), subband decomposition, or differential pulse-code modulation), quantization, etc. The output of the source coder still exhibits various kinds of statistical dependencies. The (lossless) entropy coder exploits the statistical properties of data and removes the remaining redundancy after the lossy coding.

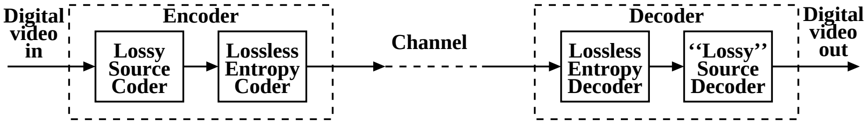


Figure 3.8: **A generic video codec.**

In MPEG, the couple DCT + Quantization is used as a lossy coding technique. The DCT algorithm processes the video data in blocks of 8×8 pixels, decomposing each block into a weighted sum of amplitudes of 64 spatial frequencies. At the output of DCT, the data is also organized as 8×8 blocks of coefficients, each coefficient representing the contribution of a spatial frequency for the video block being analyzed. Since the human eye cannot readily perceive high frequency activity, a quantization step is then carried out. The goal is to force as many DCT coefficients as possible to zero within the boundaries of the prescribed video quality. Then, a zig-zag operation transforms the matrix into a vector of coefficients which contains large series of zeros. This vector is further compressed by an Entropy Coder which consists of a Run-Length Coder (RLC) and a Variable-Length Coder (VLC). The RLC represents consecutive zeros by their run lengths; thus the number of samples is reduced. The RLC output data are composite words, referred to as *symbols*, which describe a *run-level* pair. The *run* value indicates the number of zeros by which a (non-zero) DCT coefficient is preceeded. The *level* value represents the value of the DCT coefficient. When all the remaining coefficients in a vector are zero, they are all coded by the special symbol *end-of-block*. Variable-length coding is a mapping process between *run-level*/*end-of-block* symbols and *variable-length codewords*, which is carried out according to a set of tables defined by the standard. Not every *run-level* pair has a variable-length codeword to represent it, only the frequent used ones do. For those rare combinations, an *escape* code is given. After an *escape* code, the *run*- and *level*-value are coded using fixed-length codes.

In this dissertation we focus on MPEG decoding, i.e., on the operation inverse to MPEG encoding. More details regarding the MPEG standard will be provided at each chapter as they become relevant.

3.7 Conclusion

We have described an extension of the TriMedia–CPU64 instruction set architecture that incorporates support for the reconfigurable core. Our extension is particularly convenient since it does not create pressure on the instruction decoder, neatly fits in the existing instruction format, fits the existing connectivity structure to the register file, and hence requires very little hardware overhead. In addition, since the C-level custom operations are directly translated into machine-level operations, the TriMedia–CPU64 compiler needs only to know their syntax. By using this characteristic in connection to the retargetability of the compiler, new RFU-specific operations can be easily specified at C level. Therefore, we can conclude that the incorporation of the reconfigurable core has only a little impact on the compilation and simulation tool chain.

So far, we have answered to the first two research questions we posed in the Introduction. To answer to the third question, that is,

3. What is the influence of the reconfigurable array on the TriMedia–CPU64 computing performance?

we will propose a number of reconfigurable design and will assess the performance improvement than can be achieved on ρ–TriMedia over TriMedia–CPU64. This is the subject of the next chapters.

Chapter 4

Inverse Discrete Cosine Transform

Inverse Discrete Cosine Transform (IDCT) constitutes an important operation of MPEG-related standards and has found wide applications in other fields (e.g., digital filtering) as well. Traditionally, IDCT has been implemented in hardware for Application-Specific Instruction Processors (ASIP), or in software in media-domain processors. In this chapter, we describe a reconfigurable IDCT design for ρ–TriMedia. Essentially, we implement an 1-D IDCT operation on the reconfigurable functional unit, and establish the gains in performance when computing a 2-D (8×8) IDCT.

The chapter is organized as follows. For background purpose, we present the most important issues related to IDCT theory in Section 4.1. Several considerations regarding the 2-D IDCT implementation on standard TriMedia-CPU64 are discussed in Section 4.2. Implementation issues of the 1-D IDCT computing resource on FPGA are presented in Section 4.3. The execution scenario of the 2-D IDCT on ρ–TriMedia, as well as experimental results are presented in Section 4.4. Section 4.5 completes the chapter with some conclusions and closing remarks.

4.1 Theoretical background

The transformation for an N point 1-D IDCT is defined by [85]:

$$x_i = \frac{2}{N} \sum_{u=0}^{N-1} K_u X_u \cos \frac{(2i+1)u\pi}{2N}$$

where X_u are the inputs, x_i are the outputs, and $K_u = \sqrt{1/2}$ for $u = 0$, otherwise is 1. For MPEG, a 2-D IDCT processes an 8×8 matrix X [70]:

$$x_{i,j} = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} K_u K_v X_{u,v} \cos \frac{(2i+1)u\pi}{16} \cos \frac{(2j+1)v\pi}{16}$$

A standard strategy to compute the 2-D IDCT is the row-column separation. The 2-D transform is performed by applying the 1-D transform to each row (*horizontal* IDCTs) and subsequently to each column (*vertical* IDCTs) of the data matrix. This strategy can be combined with different 1-D IDCT algorithms to further reduce the computational complexity. One of the most efficient 1-D IDCT algorithms has been proposed by Loeffler [67]. It has to be mentioned that the Loeffler algorithm has an intrinsic gain of $2\sqrt{2}$. Thus, after the vertical and horizontal 1-D IDCTs have been computed, a factor of $2\sqrt{2} \times 2\sqrt{2} = 8$ has to be compensated out. This can be easily achieved by right-shifting by three positions.

To fulfill the IEEE numerical accuracy requirements for IDCT in MPEG applications [1] when all the computations are carried out with 16-bit signed values, van Eijndhoven and Sijstermans proposed a slightly different version of the Loeffler algorithm in which the $\sqrt{2}$ factors are moved around [114], as depicted in Figure 4.1. Due to its higher numerical accuracy, we will use this modified algorithm in the subsequent experiment. We would like to mention that the modified algorithm has also an intrinsic gain of $2\sqrt{2}$.

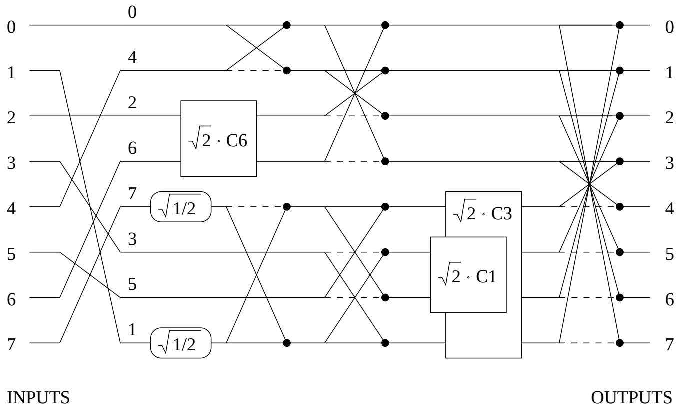
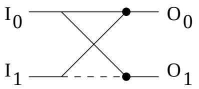


Figure 4.1: **The modified 'Loeffler' algorithm.**

In the figure, the round block signifies a multiplication by $C_0' = \sqrt{1/2}$. The butterfly block and the associated equations are presented in Figure 4.2.

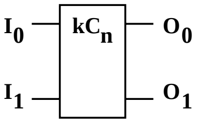
A square block depicts a rotation which transforms a pair $[I_0, I_1]$ into $[O_0, O_1]$.



$$O_0 = I_0 + I_1$$
$$O_1 = I_0 - I_1$$

Figure 4.2: **The butterfly – [67].**

The symbol of a rotator and the associated equations are presented in Figure 4.3.



$$O_0 = I_0 k \cos \frac{n\pi}{16} - I_1 k \sin \frac{n\pi}{16} = kC_n I_0 - kS_n I_1 = C'_n I_0 - S'_n I_1$$
$$O_1 = I_0 k \sin \frac{n\pi}{16} + I_1 k \cos \frac{n\pi}{16} = kS_n I_0 + kC_n I_1 = S'_n I_0 + C'_n I_1$$

Figure 4.3: **The rotator – [67].**

Although an implementation of such a rotator with three multiplications and three additions is possible (Fig. 4.4 – a, b), we used the direct implementation of the rotator with four multiplications and two additions (Fig. 4.4 – c), because it shortens critical path and improves numerical accuracy. Indeed, there are three operations (two additions and a multiplication) on the critical path of the implementations with three multipliers, while the critical path of the implementation with four multipliers contains only two operations (a multiplication and an addition). Also, the initial addition involved by the three-multiplier implementations may lead to an overflow when fixed-point arithmetic is carried out.

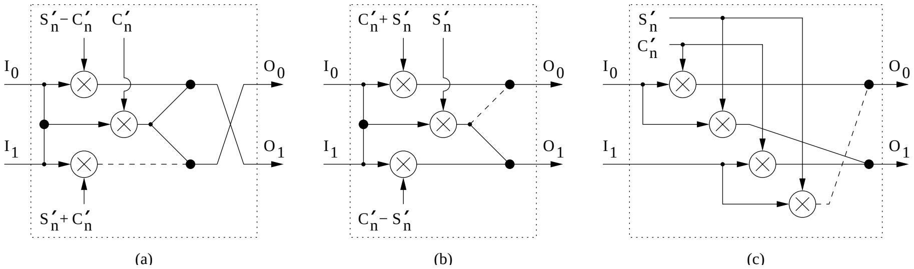


Figure 4.4: **Three possible implementations of the rotator.**

4.2 2-D IDCT pure-software implementation

In the 2-D IDCT implementation on standard TriMedia–CPU64, all computations are done with 16-bit values, and make intense use of four-way SIMD-style operations. The 8×8 matrix is stored in sixteen 64-bit words, each word containing a half row of four 16-bit elements. Therefore, four 16-bit elements can be processed in parallel by a single word-wide operation. Next to that, since TriMedia is a 5-issue slot VLIW processor, five such operations can be executed per clock cycle.

To calculate the 2-D IDCT, eight 1-D IDCTs are first computed using the modified 'Loeffler' algorithm [114]. By using the 4-way SIMD operations, four IDCTs are effectively computed concurrently. That is, eight 1-D IDCTs are calculated as two SIMD 1-D IDCTs. Then, the transpose of the 8×8 matrix is performed by a transpose functional unit which covers a double issue slot. The `TRANSPOSE` double-slot operation can generate the upper respectively lower two words of a transposed 4×4 matrix in one cycle. Therefore, the 8×8 matrix transpose is computed in eight basic operations. Finally, eight 1-D IDCTs (two SIMD 1-D IDCTs) are computed with the results generated by the transposition. Following the described procedure, a complete 2-D IDCT including the `LOAD` and `STORE` operations (cache misses are not counted) can be performed in 56 cycles [113]. 48 `NOP`s are inserted by the TriMedia scheduler in the 56-cycle routine (`NOP`s associated to super-operations are not counted), which translates to an average utilization of 4.14 out of 5 operations per instruction.

4.3 1-D IDCT implementation on FPGA

Since the standard TriMedia provides a good support for transposition and matrix storage, we expect to get little benefit if we configure the entire 2-D IDCT into FPGA. Our goal is to balance the cost of storing the intermediate 2-D IDCT results into an FPGA-resident transpose matrix memory against obtaining free slots into TriMedia. Consequently, in our implementation on the extended TriMedia, we configure only an 1-D IDCT 2-slot computing resource on the RFU. By launching an 1-D IDCT double-slot operation having two 64-bit inputs and two 64-bit outputs, an 1-D IDCT is computed on eight 16-bit values. To calculate the 2-D IDCT, eight 1-D IDCT are firstly computed. Then a transpose is performed on the 8×8 data matrix using TriMedia native `TRANSPOSE` double-slot operations. Finally, eight 1-D IDCT are again computed. This execution scenario is presented in Figure 4.5.

Let us assume that a horizontal packing of the data is needed at the output of the 8×8 IDCT, i.e., four elements in the same matrix line are to be stored into a (64-bit) double word. If the input data is also horizontally packed, then two transposition stages are needed (Fig. 4.6 – left), otherwise, only the middle transposition stage is required (Fig. 4.6 – right). In connection to the target media domain of TriMedia, in particular the MPEG decoding task [70], we notice that the front transposition stage

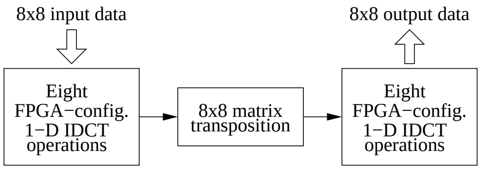

8x8 input data
Eight FPGA−config. 1−D IDCT operations
8x8 matrix transposition
8x8 output data
Eight FPGA−config. 1−D IDCT operations

Figure 4.5: **The computing scenario of 8×8 IDCT on the extended TriMedia.**

can be bypassed if the appropriate *zig-zag* scan ordering or its transposed version is used during the reconstruction process of the 8×8 matrices. Consequently, the performance evaluation of the 8×8 IDCT takes into consideration only eight 1-D IDCTs, a transposition stage, and again eight 1-D IDCTs.

As mentioned in Chapter 3, a hypothetical implementation of an FPGA-based computing unit having a recovery of 1 is not a realistic scenario, and a recovery of 2 or more is mandatory for the time being. Hereafter, we assume a recovery of 2 for 1-D IDCT, which translates into an FPGA cycle time to TriMedia cycle time ratio of 2. Considering a TriMedia–CPU64 running at 200 MHz, the pipelined implementation of 1-D IDCT will work with a clock frequency of 100 MHz. In the same time, the optimization task should also attempt to find the lowest possible number of 1-D IDCT pipeline stages. This is needed in order to reduce the pressure on the register file when the 2-D IDCT software pipeline is built. Subsequently, we describe in detail the 1-D IDCT implementation on an ACEX EP1K100 FPGA from Altera.

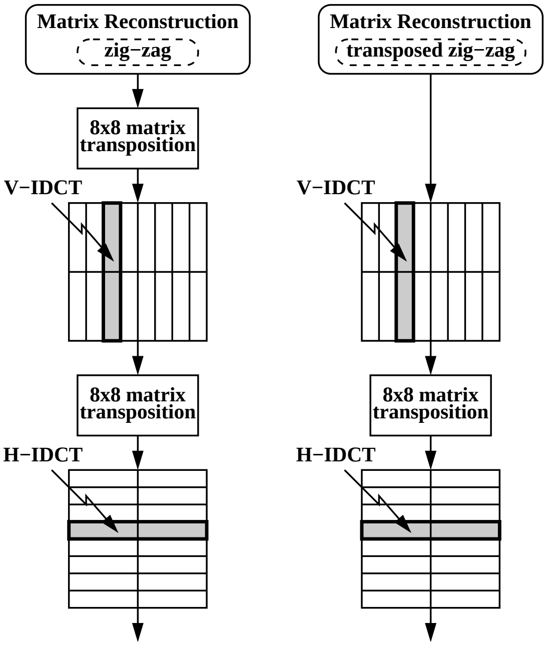


Figure 4.6: **Bypassing the first transposition.**

Implementation issues of the 1-D IDCT. All operations required to compute 1-D IDCT are implemented using 16-bit fixed-point arithmetic. Referring again to

Section 4.1, and to Figures 4.1, 4.3, and 4.4, since the computation of 1-D IDCT requires 14 multiplications, an efficient implementation of each multiplication is of crucial importance. For all multiplications, the multiplicand is a 16-bit signed number represented in 2's complement notation, while the multiplier is a positive constant of 15 bits or less. As proved in [112], these word lengths in connection with fixed-point arithmetic and proper rounding are sufficient to fulfill the IEEE numerical accuracy requirements for IDCT in MPEG applications.

A general multiplication scheme for which both multiplicand and multiplier are unknown at the implementation time exhibits the largest flexibility at the expenses of higher latency and larger area. If one of the operands is known at the implementation time, the flexibility of the general scheme becomes redundant, and a customized implementation leads to improved latency and area. A scheme which is optimized against one of the operands is referred to as *multiplication-by-constant*. Since such a scheme is more appropriate for our application, we will use it subsequently. We would like to emphasize that due to the FPGA reconfiguration capability, this customization can be re-iterated if the constant changes.

To implement the multiplication-by-constant scheme, we built a partial product matrix, where only the rows corresponding to a '1' in the multiplier are filled in. Then, reduction schemes which fit into a pipeline stage running at 100 MHz are sought. It should be emphasized that a reduction algorithm which is optimum on a particular FPGA family may not be appropriate for a different family.

In order to implement an IDCT at 100 MHz, reduction modules which can run at 100 MHz or more should be considered. Subsequently, we present the reduction steps for all multiplications. In order to implement 16-bit fixed-point arithmetic, both the multiplicand and multiplier have been properly scaled so that values remain representable with 16 bits and 15 bits, respectively, while preserving the highest possible precision [112]. Also, only the most significant 16 bits of the extended 31-bit product are to be stored. It is worth mentioning that only 30 of 31 bits of the product have to be computed. As depicted in Figure 4.7 – (a), the 31st (most significant) bit (labeled '30') is derived from the sign-bit of the multiplicand, while the carry from position '29' to '30' is discarded. Recursively, assuming that the multiplier magnitude is small enough so that the multiplier can be represented with only 14, 13, ... bits, then only 29, 28, ... of 31 bits, respectively, have to be computed. The most significant 2, 3, ... bits of the product are derived from the sign-bit of the multiplicand, while the carry from position '28' to '29', '27' to '28', ..., respectively, is discarded (Fig. 4.7 – (b)).

In addition to the solution we described in [95], we implemented a *Rounding-To-Nearest* (rtn) scheme [75] at the end of each multiplication. Assuming

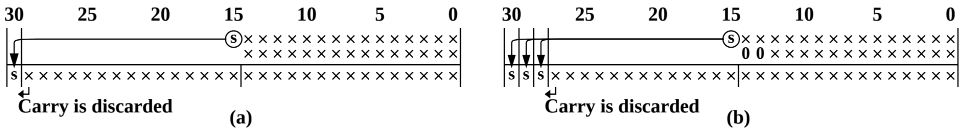


Figure 4.7: **Sign-extension for: (a) – 15-bit multiplier; (b) – 13-bit multiplier.**

that the extended 31-bit product is $p_{15}, \ldots, p_1, p_0, p_{-1}, p_{-2}, \ldots, p_{-15}$, where $p_{-1}, p_{-2}, \ldots, p_{-15}$ are the bits to be discarded, the *rounding-to-nearest* is performed by adding the bit p_{-1} to the 16-bit ($p_{15}, \ldots, p_1, p_0$) unrounded product, as it is depicted in Figure 4.8 ('S' represents the sign-bit, and 'R' specifies the rounding-bit). The generated rounding function is depicted in Figure 4.9.

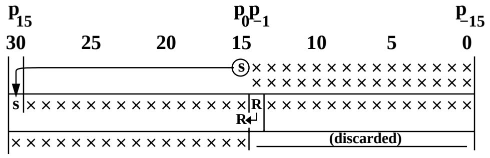


Figure 4.8: ***Rounding-To-Nearest* implementation.**

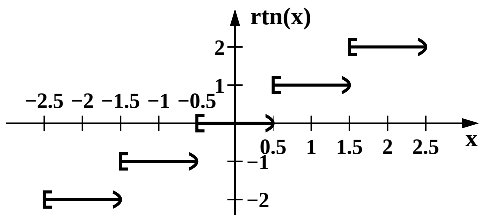


Figure 4.9: **Rounding a real value to the nearest integer.**

In connection to the rounding stage, several comments are worth to be provided. First, in order to obtain the correct result when rounding from the (31-bit) highest negative numbers, i.e., `7fff 8000` ÷ `7fff ffff`, to zero, carry propagation over all 16 bits of the rounded product is needed. That is, the sign bit is involved in computation during the rounding stage. This imposes a significant overhead as the final rounding is always on the critical path of the multiplication. Once again, 16-bit fixed-point arithmetic and *rounding-to-nearest* is sufficient to fulfill the IEEE numerical accuracy requirements for IDCT in MPEG applications. This was really confirmed by performing the IEEE accuracy validation [112]. This means that when rounding from the (31-bit) highest positive numbers, i.e., `3fff 8000` ÷ `3fff ffff`, an overflow will never been encountered. For this reason, saturating arithmetic is not needed and, therefore, has not been implemented. Second, following the procedure described in [75], we estimate that the normalized magnitude of the upward *bias* introduced by the *rounding-to-nearest* scheme is:

$$\text{bias} = \frac{0.5}{2^{15-\text{atzlsb}}}$$

where *atzlsb* (*all-time zero least significant bits*) is the number of least significant $p_{-1}, p_{-2}, \ldots, p_{-15}$ bits of the product which are zero all the time. In the most disadvantageous case, *atzlsb* = 3 (multiplication by S_1', as shown later). Thus,

$$\text{bias} = \frac{0.5}{2^{12}} = \frac{0.5}{4096} = 0.00012207$$

which means that $4096 \cdot 2 = 8192$ operations containing a rounding step, i.e., multiplications, are needed to affect the precision of the 16-bit rounded product by only 1 bit! Fortunately, the number of the operations containing a rounding step which are needed to reconstruct a pixel (both 2-D IDCT and motion compensation are considered) is far less that 8196 (is of the order of 20 or so). Consequently, the bias introduced by the *rounding to nearest* scheme will never affect the result and, therefore, is out of concern.

The partial product matrix and the selected reduction modules and steps for multiplication by the constant $C_0' = 5\text{a}82\,\text{h}$ are presented in Figure 4.10 (the Roman numerals indicate the reduction steps). First, the partial product matrix is built. Then, reductions on the modules specified by the shaded areas are carried out. The first stage generates four binary numbers of different lengths result, which are reduced to one row in the second stage. Therefore, a multiplication by the constant C_0' including rounding is performed in two pipeline stages.

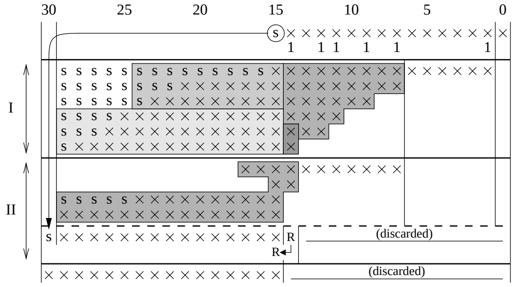


Figure 4.10: **The partial product matrix and the selected reduction steps for multiplication by the constant C_0'.**

The partial product matrix and the selected reduction modules and steps for multiplication by the constant $C_1' = 58\mathrm{c}5\,\mathrm{h}$ are presented in Figure 4.11. The reduction is performed in a horizontal way, two lines at a stage. Therefore, a multiplication by the constant C_1' is performed in three stages. The multiplication by the constant C_1' proved too difficult to be implemented in two stages only.

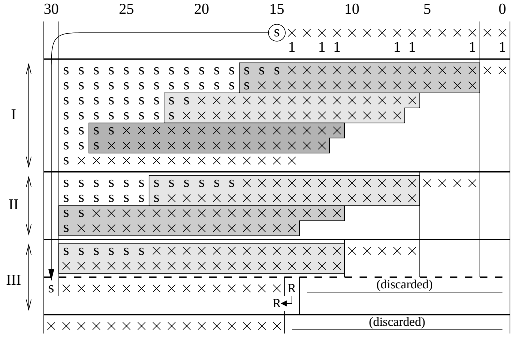


Figure 4.11: **The partial product matrix and the selected reduction steps for multiplication by the constant C_1'.**

The partial product matrix and the selected reduction modules and steps for multiplication by the constants $S_1' = 11\mathrm{a}8\,\mathrm{h}$, $C_3' = 4\mathrm{b}42\,\mathrm{h}$, $S_3' = 3249\,\mathrm{h}$, $C_6' = 22\mathrm{a}3\,\mathrm{h}$, and $S_6' = 539\mathrm{f}\,\mathrm{h}$ are presented in Figures 4.12, 4.13, 4.14, 4.15, and 4.16, respectively. Concerning the multiplication by constant S_6', some comments are worth to be provided. In order to reduce the number of '1' in the multiplier S_6' and, consequently, the number of rows in the corresponding partial product matrix, the Booth's recoding [75] has been applied. That is, the multiplier S_6' is rewritten as $S_6' = 5420\,\mathrm{h} - 0081\,\mathrm{h}$, and the rows in the partial product matrix corresponding to $0081\,\mathrm{h}$ are subtracted rather than added.

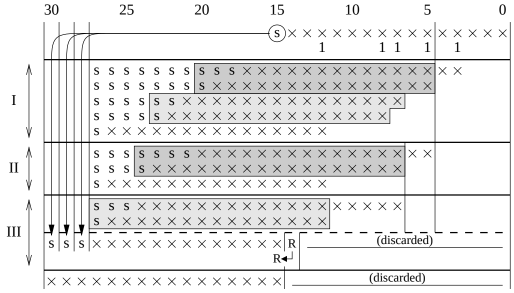


Figure 4.12: **The partial product matrix and the selected reduction steps for multiplication by the constant S_1'.**

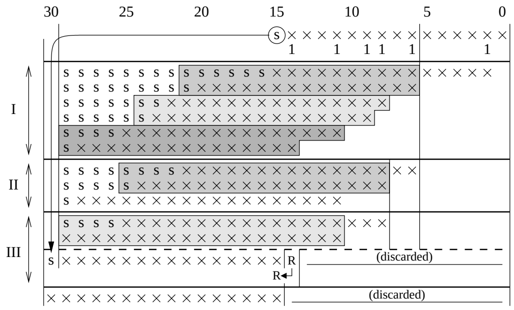


Figure 4.13: **The partial product matrix and the selected reduction steps for multiplication by the constant C'_3.**

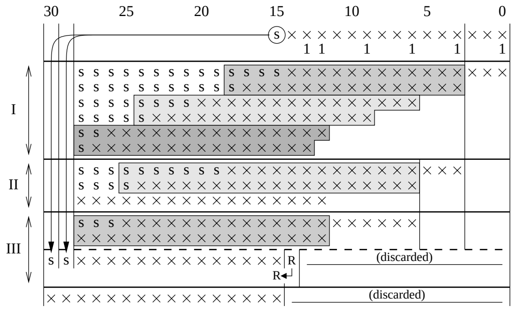


Figure 4.14: **The partial product matrix and the selected reduction steps for multiplication by the constant S'_3.**

We would like to note that the critical path of the 1-D IDCT is located on the lower half part of the modified ‘Loeffler’ algorithm (Fig. 4.1). Once the multiplication by constant C'_1 is performed in three stages, there is no gain in performance to implement the other three multiplications by constants S'_1, C'_3, S'_3 in less than three stages. Therefore, the multiplications by the constants S'_1, C'_3, S'_3 are implemented in three stages also, even though they may allow for an efficient (timing) implementation in two stages, too (however, at the expense of a slightly larger area). The same timing-relaxed implementation strategy is used for multiplications by the constants C'_6 and S'_6, since they both are not located on the critical path.

The sketch of the 1-D IDCT pipeline is depicted in Figure 4.17 (the roman numerals specify the pipeline stages). As mentioned, our effort was focused on the optimization of the entire IDCT compound. Considering the critical path, the latency of the 1-D IDCT is composed of:

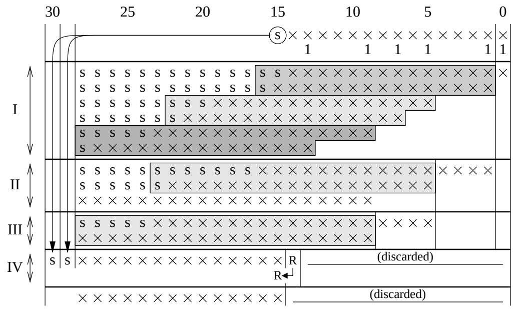


Figure 4.15: **The partial product matrix and the selected reduction steps for multiplication by the constant C_6'.**

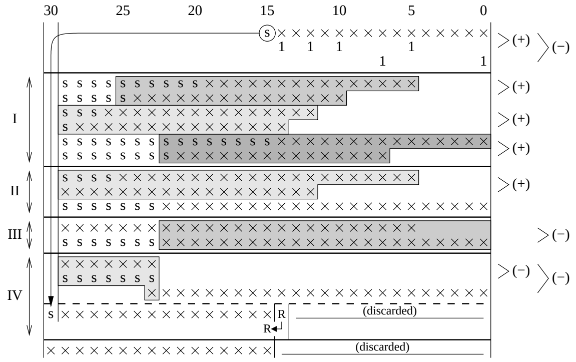


Figure 4.16: **The partial product matrix and the selected reduction steps for multiplication by the constant S_6'.**

- one TriMedia cycle for reading the input operands from the register file into the input flip-flops of the 1-D IDCT computing resource;
- two pipeline stages (that is, two FPGA cycles or four TriMedia cycles) for computing the multiplication by constant C_0';
- one pipeline stage (that is, two TriMedia cycles) for computing all additions/subtractions required by the butterflies that are located between the $\sqrt{1/2}$ multipliers and $\sqrt{2}C_1$ and $\sqrt{2}C_3$ rotators.
- three pipeline stages (six TriMedia cycles) for computing the multiplication by constant C_1';
- one pipeline stage (two TriMedia cycles) for computing all the remaining additions/subtractions (at the end of $\sqrt{2}C_1$ and $\sqrt{2}C_3$ rotators, and final butterflies);
- one TriMedia cycle for writing back the results from the output flip-flops of the 1-D IDCT computing resource into the register file.

Therefore, the latency of the 8-point 1-D IDCT operation is $1 + (2 + 1 + 3 + 1) \times 2 + 1 = 16$ TriMedia cycles. We determined that 1-D IDCT uses 45% of the logic elements and 257 I/O pins of an ACEX EP1K100 device.

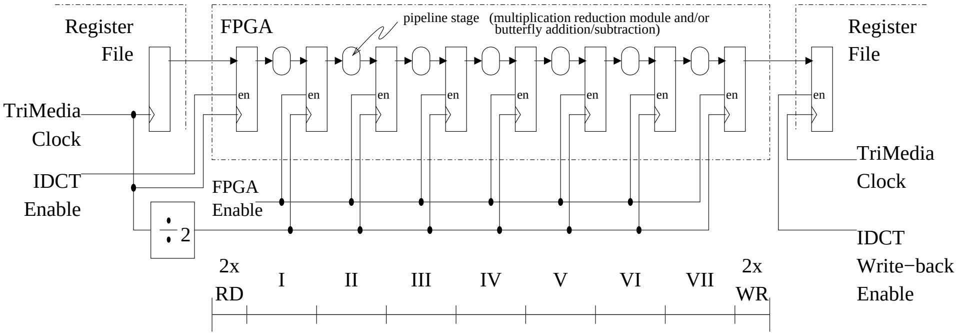


Figure 4.17: **The 1-D IDCT pipeline.**

4.4 2-D IDCT implementation on ρ–TriMedia

As mentioned, an 1-D IDCT with a latency of 16 and a recovery of 2 is configured on the RFU at application launch-time. We assigned the IDCT operation to the slot pair 1+2. After eight 1-D IDCTs, eight `TRANSPOSE` super-operations are scheduled on the slot pairs 1+2 or 3+4 to compute the transpose of the 8×8 matrix. Then, eight 1-D IDCTs complete the 2-D IDCT. Before and after each 2-D IDCT, `LOAD` and `STORE` operations fetch the input operands from main memory into register file, and store the results back into memory, respectively. The scheduled code and the performance figures are presented in Figure 4.18.

In order to keep the pipeline full, back-to-back 1-D IDCT operation is needed. That is, a new 1-D IDCT instruction has to be issued every two cycles. Since true dependencies forbid issuing the last eight 1-D IDCTs of a 2-D IDCT to fulfill back-to-back requirement, the 2-D IDCTs are processed in chunks of two, in an interleaved fashion. A number of $2 \times 16 = 32$ registers are needed for this processing pattern. This 2-D IDCT implementation exhibits a throughput of 1/32 IDCT/cycle and a latency of 84 cycles for two IDCTs (that is, an average of 42 cycles/IDCT). It is worth mentioning that the machine is well balanced, none of the 5-slot VLIW instructions being fully occupied with operations:

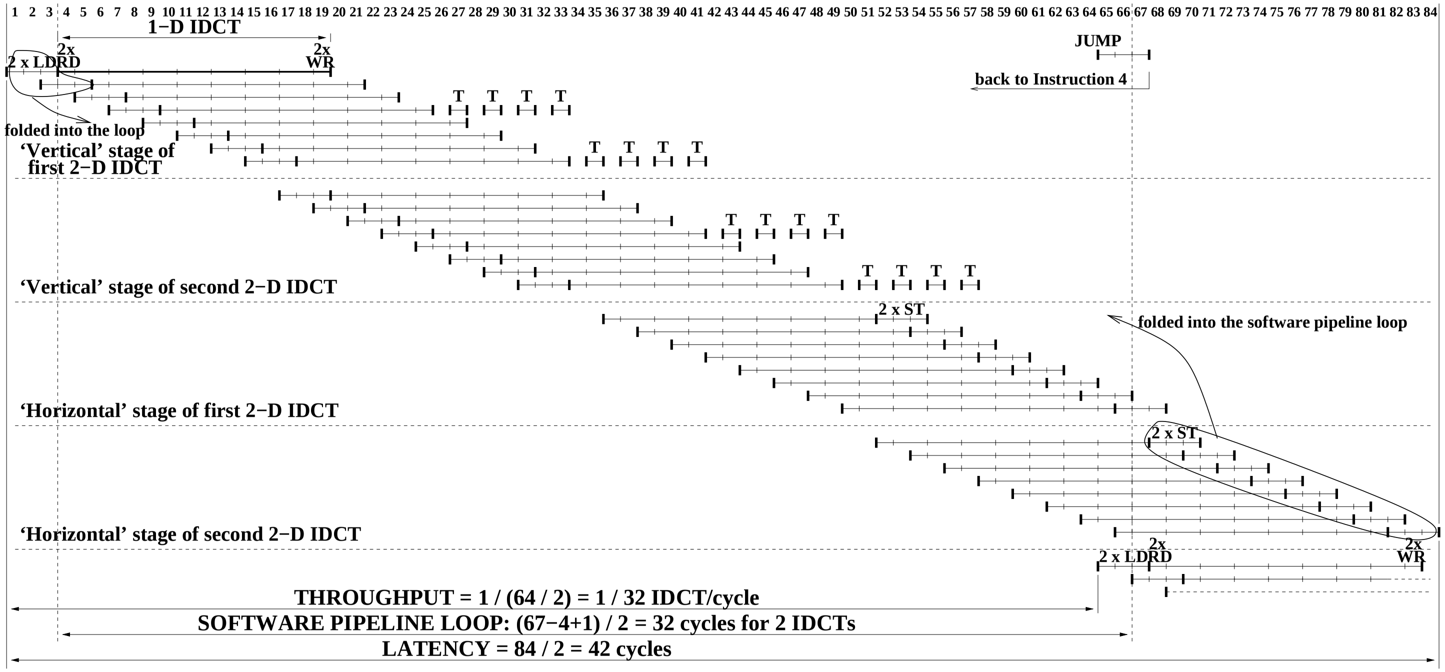


Figure 4.18: **Schedule result for a 1-D IDCT having the latency of** 16 **and recovery of** 2 **(LD stands for** `LOAD`**, RD for read, WR for write, ST for** `STORE`**, and T for** `TRANSPOSE`**).**

- Two `LOAD` or two `STORE` operations are issued every other clock cycle on slots 4 and 5; thus the slots 4 and 5 are only 50% occupied.
- `IDCT` super-operations are issued on slots $1 + 2$ every other clock cycle, which translates to a 50% usage of the slots 1 and 2.
- The *transpose* super-operations are also issued on every other clock cycle, and the issuing slots can be either $1 + 2$ or $3 + 4$. Since there are only eight transpositions per 2-D IDCT, the overall slot occupancy percentage does not increase significantly above 50%.

In this way, there are plenty of free slots which can be utilized for other purposes, e.g., for implementing the post-IDCT rounding and saturation required by MPEG standard [70], or even a 2-D IDCT in the standard instruction set. Consequently, the announced figures represent the lower bound of the performance that can be achieved on ρ–TriMedia.

In connection to the scheduled code presented in Figure 4.18, we would like to mention that cycling over Instructions $1 \div 84$ is needed to launch the computation of the next two 2-D IDCTs. The immediate effect is that there is an overhead associated to firing-up and flushing the reconfigurable-hardware (1-D IDCT) pipeline. Thus, the throughput of 1/32 IDCT/cycle corresponds to the ideal scenario of a loop which is unrolled an infinite number of times.

In order to have a realistic scenario, two techniques can be employed: (1) finite loop unrolling + grafting techniques, and (2) *software pipelining*. Both techniques are analysed subsequently and performance figures are provided. Concerning the second technique we have to mention that, for the time being, the TriMedia scheduler uses the decision tree as a scheduling unit [50]. Thus, all operations return the results in the same decision tree that they are launched, even though the TriMedia architecture does not forbid the contrary. This is the major limiting factor in generating deep software pipelined loops containing long-latency operations. However, the code containing RFU operations is very simple. Therefore, programming directly in assembly is indeed a feasible solution for such simple routines, despite of the fact that the host is a complex VLIW processor.

Figure 4.18 also presents the edges of the software pipeline loop (Instructions 4 and 67), as well as the corresponding `JUMP` operation which cycles over the loop. To employ loop pipelining, the first 4 `LOAD` operations and the last 16 `STORE` operations should be folded into the loop. Thus, the overhead associated to firing-up and flushing the software pipeline (i.e., the *prologue* and the *epilogue*) consists of these 4 `LOAD`, and 16 `STORE` operations, respectively, which have to be issued any other cycle. Thus, the total overhead is 20 cycles.

In order to assess the implications of the loop prologue and epilogue in a real case, we have focused on the average number of coded blocks per slice for a number of MPEG-conformance bit-strings (Table 4.1). If all the blocks in an MPEG slice are first reconstructed and only then transformed as a single batch, then the lowest average batch size is 38 blocks/slice (B frames in the *popplen* scene). This figure translates into the worst case penality associated to the prologue and epilogue of the software pipeline loop of $20/38 \approx 0.53$ cycles/block. Since this overhead represents less than 1.66% of the 32 cycle/block throughput in the most disadvantageous case, it can be neglected.

Table 4.1: **The average number of coded blocks per slice for a number of MPEG-conformance bit-strings.**

Scene		Coded blocks/slice		
	Frame type	I	P	B
bat_327_334		–	257	234
popplen		264	80	38
sarnoff2		270	171	61
tennis		264	167	71
ti1cheer		264	155	88

In Table 4.2, we present performance figures for two loop organizations (linear and software pipelined), several computing scenarios (FPGA-based 2-D IDCTs are processed in chunks of two, FPGA-based 2-D IDCTs are blended with a single 2-D IDCT computed in software, vertical 1-D IDCTs and transpositions are computed first for all matrices of the testbench, and only then all horizontal 1-D IDCTs are carried out), and several degrees of loop unrolling. Since the IDCT rounding and saturation as specified by MPEG standard [70] may be subject to optimization at a complete MPEG decoder level, we will also present the experimental figures for three cases: IDCT rounding and saturation is performed in FPGA as an additional (the eighth) hardware pipeline stage, in software in the standard TriMedia instruction set, or postponed for a subsequent stage of MPEG decoding process. We mention that when the IDCT rounding and saturation is carried out immediately after the 2-D IDCT completed, the square of the intrinsic Loeffler gain is also compensated out by right-shifting by three positions (i.e., integer division by 8).

Table 4.2: **Performance figures for 8×8 IDCT on (FPGA-augmented) TriMedia.**

Loop organization	Computing scenario	Unrolling degree	IDCT rounding and saturation	Effectiveness (issues/cycle)	Performance (cycles/8×8 matrix)	Comments
Linear	Two FPGA-IDCTs	none	none	1.94	41.5	
Linear	Two FPGA-IDCTs	none	in FPGA	1.87	43.0	requires two `RFU— OP— IDs`
Linear	Two FPGA-IDCTs	none	in SW	1.87	44.4	
Linear	Two FPGA-IDCTs	2×	none	2.18	36.8	
Linear	Two FPGA-IDCTs	2×	in FPGA	2.15	37.3	requires two `RFU— OP— IDs`
Linear	Two FPGA-IDCTs	2×	in SW	3.76	38.5	
Linear	Two FPGA-IDCTs	3×	none	2.30	61.6	**spilling encountered**
Linear	Two FPGA-IDCTs	3×	in FPGA	2.11	63.1	**spilling encountered** requires two `RFU— OP— IDs`
Linear	Two FPGA-IDCTs	3×	in SW	3.22	64.0	**spilling encountered**
Linear	Four FPGA-IDCTs + one SW-IDCT	none	none	3.10	35.6	
Linear	Four FPGA-IDCTs + one SW-IDCT	none	in FPGA/SW	3.24	38.2	requires two `RFU— OP— IDs`
Linear	16 Vertical 1-D IDCTs + Transposition	none	n/a	2.30	28.0	
	16 Horizontal 1-D IDCTs	none	none	1.90	25.5	
	Total for 2-D IDCT	–	–	–	53.5	
Linear	16 Vertical 1-D IDCTs + Transposition	none	n/a	2.30	28.0	requires two `RFU— OP— IDs`
	16 Horizontal 1-D IDCTs	none	in FPGA	1.83	26.5	
	Total for 2-D IDCT	–	–	–	54.5	
Linear	16 Vertical 1-D IDCTs + Transposition	2×	n/a	2.92	22.0	
	16 Horizontal 1-D IDCTs	2×	none	2.33	20.8	
	Total for 2-D IDCT	–	–	–	42.8	
Linear	16 Vertical 1-D IDCTs + Transposition	2×	n/a	2.92	22.0	requires two `RFU— OP— IDs`
	16 Horizontal 1-D IDCTs	2×	in FPGA	2.27	21.3	
	Total for 2-D IDCT	–	–	–	43.3	

Table 4.2: **Performance figures for 8 × 8 IDCT on (FPGA-augmented) TriMedia (contd.).**

Loop organization	Computing scenario	Unrolling degree	IDCT rounding and saturation	Effectiveness (issues/cycle)	Performance (cycles/8×8 matrix)	Comments
Linear	16 Vertical 1-D IDCTs + Transposition	3×	n/a	2.79	37.2	**spilling encountered** requires two `RFU— OP— IDs`
	16 Horizontal 1-D IDCTs	3×	none	2.03	35.5	
	Total for 2-D IDCT	–	–	–	72.7	
Linear	16 Vertical 1-D IDCTs + Transposition	3×	n/a	2.79	37.2	**spilling encountered** requires two `RFU— OP— IDs`
	16 Horizontal 1-D IDCTs	3×	in FPGA	1.98	36.8	
	Total for 2-D IDCT	–	–	–	74.0	
Software pipelined	Two FPGA-IDCTs	n/a	none	3.81	32.1	
Software pipelined	Two FPGA-IDCTs	n/a	in FPGA	3.69	33.1	requires two `RFU— OP— IDs`
Software pipelined	All Vertical 1-D IDCTs + Transposition	n/a	n/a	3.20	20.1	
	All Horizontal 1-D IDCTs	n/a	none	4.10	16.1	
	Total for 2-D IDCT	–	–	–	36.2	
Software pipelined	All Vertical 1-D IDCTs + Transposition	n/a	n/a	3.20	20.1	requires two `RFU— OP —IDs`
	All Horizontal 1-D IDCTs	n/a	in FPGA	4.10	16.2	
	Total for 2-D IDCT	–	–	–	36.3	

Note: **FPGA-IDCT** stands for an IDCT which benefits from reconfigurable hardware support
SW-IDCT stands for an IDCT carried out in software.

As expected, the best result is obtained for a software pipeline loop: 1/32 IDCT/cycle. However, a linear loop organization with two FPGA-based IDCTs and $2\times$ unrolling is not a bad choice either, since it achieves a throughput only 17% lower: 1/37.3 IDCT/cycle. Since generating software pipeline loops is not supported by the current TriMedia toolchain, the advantage of the later approach is an easier programming task. If IDCT rounding and saturation can be postponed for a different stage of MPEG decoding process, the best solution for a linear loop corresponds to a computing scenario with four FPGA-based and one software-based IDCTs in the loop. The throughput in this case is 1/35.6 IDCT/cycle. For the same linear loop organization and a computing scenario in which the vertical 1-D IDCTs are computed for all the matrices of the testbench in a first loop, and only then all the horizontal 1-D IDCTs are carried out in a separate loop, the double overhead associated to the prologues and epilogues of the two loops decreases the throughput to 1/42.8 IDCT/cycle (about 6% lower). Unrolling the loop three or more times generates register spilling; thus the performance degrades significantly.

It is worth noting that IDCT rounding and saturation carried out in software requires about 1.5 cycles/IDCT, and only $0.5 \div 1.0$ cycles/IDCT when carried out in FPGA. However, two `RFU`– `OP`– `ID`s are needed to embed IDCT rounding and saturation in FPGA: one `ID` for horizontal 1-D IDCT, and one `ID` for vertical 1-D IDCT. At this moment we would like to emphasize that due to the *MOLEN* concept which provides means to specify multiple RFU-based operations for the same `RFU` – `OP`, the need for two or more `ID`s never becomes a limitation.

In Table 4.3 and Figure 4.19 we compare the performances of several 2-D IDCT implementations: on standard TriMedia [113], on FPGA-augmented TriMedia, on FPGA alone [19], and on several 2-D convolution-oriented coarse-grain programmable architectures: REMARC [72], MorphoSys [101], M.*F.A.S.T.* [78], and ManArray [79]. Since the IDCT is applied on large batches of 8×8 blocks, the throughput is more important than latency. For this reason, our performance analysis is focused on the 2-D IDCT throughput. A special remark regarding the *distributed arithmetic*–based implementation on FPGA alone has to be made. Since the multiplications are computed by looking-up into on-chip small memories (the so called Block SelectRAM cells) [19], the pipeline cannot be made deeper, and 55.6 MHz is the upper bound of the frequency that can be achieved on the Virtex XCV600 device for such implementation. Thus, the number of cycles which corresponds to a throughput of 4.27 millions IDCT/sec at a clock frequency of 55.6 MHz is not relevant, and the comparison with the processor-based implementations has to be made in terms of absolute throughput expressed in IDCT/sec. With 6.25 millions IDCT/sec, the FPGA-augmented TriMedia provides an improvement of 46% in terms of throughput over FPGA alone.

Table 4.3: **2-D IDCT performance figures on state-of-the-art architectures.**

Computing engine	FPGA family	Throughput		Latency		FPGA usage
		IDCT/cycle	IDCT/s	cycles	ns	
TriMedia–CPU64 (200 MHz) [113]	n/a	1/56	3.57 M	56	280	n/a
FPGA-augmented TriMedia–CPU64	EP1K100 (Altera)	1/32	6.25 M	42	210	45 %
FPGA alone (55.6 MHz) [19]	XCV600 (Xilinx)	non relevant	4.27 M	non relevant	467.9	88 %
REMARC [72]	Coarse-grain	1/54	no info.	54	no info.	100 %
MorphoSys (100 MHz) [101]	Coarse-grain	1/37	2.70 M	37	370	100 %
M.*F.A.S.T.* (50 MHz) [78]	n/a	1/22	2.27 M	22	440	n/a
ManArray [79]	n/a	1/34	no info.	34	no info.	n/a

The 2-D IDCT implementation on standard TriMedia exhibits the lowest throughput (1/56 IDCT/cycle), while the highest throughput (1/22 IDCT/cycle) is achieved for the implementation on M.*F.A.S.T.*. The second highest throughput (1/32 IDCT/cycle) is achieved for the implementation on augmented TriMedia, which is an improvement of 75% over standard TriMedia (40% in terms of computing time). We would like to comment that the difference in performance between M.*F.A.S.T.* and FPGA-augmented TriMedia will diminish if additional computation is considered, e.g., the post-IDCT rounding and saturating required by MPEG standard [70]. While the throughput will decrease on M.*F.A.S.T.*, it will remain about the same on FPGA-augmented TriMedia, since the 1-D IDCT pipeline can be easily enlarged by an additional stage for computing post-IDCT rounding and saturating (see Table 4.2). Alternatively, post-IDCT rounding and saturating can be implemented within the standard instruction set of TriMedia, since there are still plenty of empty slots, as already mentioned.

Finally, we would like to mention that the second highest throughput is achieved with a fine-grain field-programmable custom computing machine, that is, FPGA-augmented TriMedia, which exhibits flexibility over a 2-D convolution-oriented architectures like M.*F.A.S.T.* [78] or ManArray [79] for implementing heterogenous tasks, e.g., variable-length decoding [70].

4.5 Conclusion

In this chapter we investigated the inverse discrete cosine transform, and proposed a reconfigurable 2-D IDCT design for ρ–TriMedia. Essentially, we implemented an

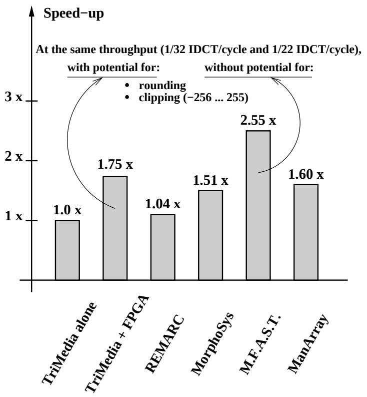


Figure 4.19: **The speed-up of various 2-D IDCT implementations on several state-of-the-art architectures relative to standard TriMedia–CPU64.**

1-D IDCT operation on the reconfigurable unit, and use the standard row-column separation strategy to compute the 2-D IDCT (the 1-D transform is carried out on each row and subsequently on each column of the 8×8 block). This way, the 1-D IDCT is computed in reconfigurable hardware, while the transposition of the 8×8 block is carried out in the standard TriMedia–CPU64 instruction set.

In particular, we also described the implementation of the 1-D IDCT computing unit on the reconfigurable hardware. This implementation is based on a modified Loeffler algorithm that ensures IEEE-compliancy for IDCT in MPEG applications. A significant effort has been made to implement each multiplier. By writing VHDL code, followed by placement and routing, we determined that the 1-D IDCT computing unit has a latency of 16 and a recovery of 2 TriMedia@200 MHz cycles, and occupies 45% of the logic elements of an ACEX EP1K100 FPGA from Altera.

By configuring the 1-D IDCT computing unit on the reconfigurable hardware, and by calling it inside a software pipeline loop together with transposition operations, 2-D IDCT can be computed with the throughput of 1/32 IDCT/cycle. This figure translates to an improvement of 40% in terms of computing time, that is, a speed-up of 75% is achieved on ρ–TriMedia over standard TriMedia–CPU64. It is also worth mentioning that there are plenty of empty slots in the implementation on ρ–TriMedia. Consequently, the announced figures represent the lower bound of the performance improvement that can be achieved on ρ–TriMedia. Given the fact that TriMedia–CPU64 is a 5-issue slot VLIW processor with 64-bit datapaths and a very rich multimedia instruction set, such an improvement within its target media processing domain indicates that ρ–TriMedia is a promising approach with respect to an IDCT task.

Chapter 5

Inverse Quantization

Quantization is basically a process for reducing the precision of the DCT coefficients. Precision reduction is extremely important, since lower precision almost always implies a lower bit rate in the compressed data stream. The quantization process involves division of the integer DCT coefficient values by integer quantizing values. The result is an integer and fraction, and the fractional part must be rounded according to the rules defined by MPEG. It is the quantized DCT values that is transmitted to the decoder. For reconstruction, the decoder must first *dequantize* the quantized DCT coefficients, to reproduce the DCT coefficients computed by the encoder. Essentially, the Inverse Quantization (IQ) algorithm scales every element by a unique quantized weight. Since some precision was lost in quantizing, the reconstructed DCT coefficients are necessarily approximations to the values before quantization.

Inverse Quantization is a computing-intensive stage of MPEG decoding. Traditionally, IQ has been captured directly in customized hardware in Application-Specific Instruction Processors, or carried out in software in media-domain processors. In this chapter, we describe a reconfigurable IQ design for ρ–TriMedia, and demonstrate that significant speed-up can be achieved over standard TriMedia–CPU64 for an IQ application.

The chapter is organized as follows. For background purpose, we present the most important issues related to IQ theory in Section 5.1. Several considerations regarding the IQ implementation on standard TriMedia-CPU64 are discussed in Section 5.2. Implementation issues of an IQ computing unit on FPGA are presented in Section 5.3. The execution scenario of the IQ on ρ–TriMedia, as well as experimental results are presented in Section 5.4. Section 5.5 completes the chapter with some conclusions and closing remarks.

5.1 Theoretical background

After entropy decoding, the two-dimensional array of coefficients, $QF[v][u]$, is inverse quantised to produce the reconstructed DCT coefficients, $F[v][u]$. In MPEG2, Inverse Quantisation (IQ) consists of three stages: Inverse Quantisation Arithmetic, Saturation, and Mismatch Control [53]. The inverse quantisation arithmetic produces $F''[v][u]$ coefficients. For DC coefficients in intra-coded blocks, Equation 5.1 is used:

$$F''[0][0] = \mathrm{intra\_dc\_mult} \times QF[0][0] \tag{5.1}$$

where the factor *intra_dc_mult* is derived from the data element *intra_dc_precision* according to Table 7-4 of the ITU-T Recommendation H.262 [53]. Basically, Equation 5.1 specifies a scaling-up by a factor of 8, 4, 2, or 1. For all other coefficients, the following equation should be used:

$$F''[v][u] = (2 \times QF[v][u] + k) \times W[w][v][u] \times \mathrm{quantizer\_scale}/32 \tag{5.2}$$

where

$$k = \begin{cases} 0 & \text{intra blocks} \\ \mathrm{sign}(QF[v][u]) & \text{non-intra blocks} \end{cases} \tag{5.3}$$

The factor *quantizer_scale* is an unsigned integer and is encoded as a 7-bit fixed-length code. Thus, it has values in the range $\{1, \ldots, 31\}$, inclusive (0 is not allowed). Each weighting coefficient, $W[w][v][u], w = 0 \ldots 3, v = 0 \ldots 7, u = 0 \ldots 7$, is represented on an 8-bit unsigned integer, and extracted during the parsing of the sequence header. The operator '/' represents the integer division with truncation of the result toward zero.

The coefficients resulting from the Inverse Quantisation Arithmetic are saturated to lie in the range $[-2048 \cdots + 2047]$. Finally, the mismatch control operation toggles the least significant bit of $F[7][7]$ if the double sum $\sum_{v=0}^{7} \sum_{u=0}^{7} F[v][u]$ of all DCT coefficients is even.

We would like to mention that MPEG defines rules for changing the quantization of the DCT coefficients from place to place in the image as follows. The factor *quantizer_scale* is derived from the data elements *quantizer_scale_code* and *quantizer_scale_type* according to Table 7-6 of the ITU-T Recommendation H.262

[53], and therefore can be changed per coded macroblock. However, the factor *intra_dc_mult* can be changed only per picture. Since we use only MP@ML MPEG conformance bit-strings in all subsequent experiments, only two weighting matrices (one for intra-coded blocks, and the other for non-intra-coded blocks) are used for inverse quantization. Thus, $w = \{0, 1\}$.

For the inverse quantization, all the mentioned values should be regarded as parameters. Consequently, the inverse quantization routine has to read in both the DCT coefficients to be dequantized and the following parameters: the weighting array W, the *quantizer_scale*, and an intra/non-intra flag.

5.2 IQ pure-software implementation

After variable-length decoding, each DCT coefficient is represented on a 16-bit signed integer. Thus, the 8×8 matrix can be thought as being stored in 16 four-element vectors. In this way, the IQ implementation can intensively use four-way SIMD operations.

In the pure-software solution, all 64 coefficients are first inverse quantised with the general Formula 5.2, and then saturated. In parallel, the intra DC coefficient is scaled-up according to Equation 5.1. Next, if the block is intra-coded, the top left-handed DCT coefficient of the 8×8 block is replaced with this DC coefficient. Finally, mismatch sum is computed and the least significant bit of $F[7][7]$ is updated accordingly. We would like to mention that a separate IQ routine has been designed for dequantizing each of the intra-coded and non-intra-coded information. The rationale behind this strategy is to bypass the computation of the **signum** function of $QF[7][7]$, and also the addition of the term $k \equiv 0$ for intra-coded blocks.

After developing C-level code that makes intensivelu use of TriMedia–CPU64 custom operations, compiling it, and running the executable on a cycle-accurate simulator, we determined that an 8×8 matrix can be dequantized in 39 cycles for intra-coded blocks, and 52 cycles for non-intra-coded blocks (`LOAD` and `STORE` operations are taken into account). 26 `NOP`s are inserted by the TriMedia scheduler in the 39-cycle routine for intra-coded blocks, which translates to an average utilization of 4.33 out of 5 operations per VLIW instruction. For non-intra-coded blocks, 30 `NOP`s are inserted in the 52-cycle routine, which means that 4.41 out of 5 operations are issued per instruction. Since the average utilization of the issue slots reaches such a large value, we can state that the TriMedia–CPU64 runs close to its full processing speed, and the pure-software IQ implementation on TriMedia–CPU64 constitutes a real challenge for an FPGA-based solution.

In inverse quantization of an 8×8 block, each and every pixel but the bottom right-hand one (which is subject to the mismatch operation) is dequantized independently of any other pixel in the block. Thus, IQ is mostly a feed-forward task that exhibits a large data-level parallelism. Consequently, the entire IQ computation can benefit from reconfigurable support if sufficient reconfigurable hardware is available. This way, the VLIW core will have only to load new data from and write the computed data back to main memory. In the next section, a number of details regarding IQ implementation on FPGA are outlined.

5.3 IQ implementation on FPGA

As mentioned, the number of pixels that can simultaneously be inverse quantized on FPGA is subject to the raw hardware logic capacity. On an ACEX EP1K100 FPGA, we succeeded to map an IQ-4 unit that can process four coefficients per call. This way, a burst of sixteen IQ-4 operations has to be launched in order to dequantize an entire 8×8 block. As depicted in Figure 5.1, the IQ-4 circuitry is structured as follows: the first part implements the IQ arithmetic (which is defined by Equations 5.1 and 5.2) and subsequent saturation, while the last part is a finite state machine implementing the mismatch control operation.

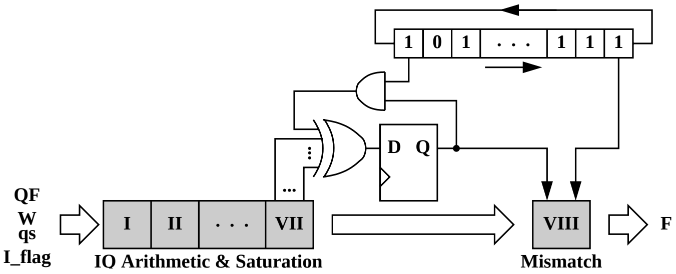


Figure 5.1: **The IQ-4 implementation on FPGA.**

The reduction modules corresponding to multiplications by $W[v][u]$ (8-bit unsigned integer) and $\mathrm{quantizer\_scale}$ (7-bit unsigned integer) have been splitted-up in order to fit into an 100 MHz pipeline. No special optimization technics to reduce the partial product matrices have been employed; instead, we rely on the FPGA mapping tools in detecting *carry-propagate* (which is fast on FPGA) primitive. The factors $\mathbf{intra\_dc\_mult}$ and $\mathbf{quantizer\_scale}$ are generated inside FPGA from the MPEG data elements $\mathbf{intra\_dc\_precision}$, respectively $\mathbf{quantizer\_scale\_code}$ and $\mathbf{q\_scale\_type}$.

In addition to the feed-forward circuitry for IQ arithmetic and saturation computation, the IQ unit also includes a finite state machine that controls the processing of the DC component in intra-coded blocks, as well as the mismatch operation as follows:

- During the first out of sixteen IQ-4 calls needed for processing an 8×8 block, the fourth element of the QF[3..0] vector (i.e., the DC component) is dequantized according to Equation 5.1 for intra-coded blocks, and Equation 5.2 for non-intra-coded blocks.
- The mismatch information is accumulated during sixteen successive IQ-4 calls, and updates the last DCT coefficient accordingly at the end of each 16^{th} call.

Thus, the IQ-4 unit we propose is a circuitry with state (non-re-entrant functional unit). In order to ensure a correct response, a block should be completely processed before a new one is being considered. Furthermore, the 64-bit word containing the DC component should be processed firstly, and the 64-bit word containing the highest spatial frequency component should be processed lastly.

By writing and synthesizing VHDL code, we determined that 8 pipeline stages are needed to implement the IQ-4 unit on an ACEX EP1K100 FPGA, which translates into a latency of $8 \times 2 + 1 + 1 = 18$ and a recovery of 2 TriMedia@200 MHz cycles. It worth to mention that IQ-4 unit occupies 43% of the logic cells, and 171 out of 333 I/O pins of the mentioned reconfigurable device.

We would also like to mention that, on the same device, we did not succeed to map an IQ-8 unit that processes eight coefficients per call. Although about 80% of the logic cells of the ACEX EP1K100 array would be occupied by the IQ-8 unit, the FPGA mapping tools did not succeed to map the circuitry mainly due to the large numbers of I/O pins that are needed. Indeed, 331 out of 333 I/O pins would be used by IQ-8. The pin limitation in FPGA-based circuitry is a known problem – see for example [8]. A way to overcome this limitation is to provide for a larger FPGA having more I/O pins (and, implicitly, more raw hardware). However, this solution is more expensive in terms of silicon area for the same number of I/O pins, since the logic capacity increases with the square root of the chip edge, while the number of I/O pins increases only linearly with the chip edge. A second solution is to emulate an IQ-8 unit processing two 8×8 blocks by two IQ-4 units each mapped on a smaller RFU, and each processing a separate 8×8 block.

In the next section, we present a number of routines that contain calls to FPGA-mapped IQ computing units, and compare the performance achieved on ρ–TriMedia over the standard TriMedia–CPU64.

5.4 IQ implementation on ρ–TriMedia

Since the FPGA-mapped IQ is a circuitry with state, two operations are needed to control the unit: one that resets the finite state machine, and the other that launches the proper IQ operation. Assuming an IQ-4 unit, the syntax of each operation is:

$$\textbf{EXECUTE} < \textbf{RESET-IQ-4} > \rightarrow$$

$$\textbf{EXECUTE} < \textbf{IQ-4} > \textbf{R\_QF, R\_W, R\_qs, R\_param} \rightarrow \textbf{R\_F}$$

The first operation has a latency of 3 cycles, while, as mentioned, the later (2-slot) operation has a latency of 18 cycles and a recovery of 2 cycles. For reasons that will become relevant later this dissertation, the inverse quantization is carried out at slice level. That is, during the decoding of the slice header, a pair of `ACTIVATE_CONTEXT` (which activates the IQ-4 computing unit) and $\texttt{EXECUTE} < \texttt{RESET} - \texttt{IQ} - 4 >$ is launched. Then, the entire slice is inverse quantized by means of $\texttt{EXECUTE} < \texttt{IQ} - 4 >$ instructions.

To inverse quantize an 8×8 block of coefficients, sixteen $\texttt{IQ} - 4$ operations are launched in a row. Before and after the RFU calls, `LOAD` and `STORE` operations fetch the input operands from main memory into register file, and store the results back into memory, respectively. Since the code is very simple and symmetrical, generating a tight software-pipeline loop by programming directly in assembly is indeed feasible, as depicted in Figure 5.2. As it can be observed, the loop is folded at Cycles 4 and 35, thus a throughput of 1/32 IQ/cycle is achieved. The first two `LOAD` operations that are executed during the previous loop iteration, and the last 9 `STORE` operations that are executed during the next loop iterations generate an overhead for firing-up and flushing the software pipeline of 24 cycles. In addition, loading the $W[w][v][u], w = 0 \ldots 1, v = 0 \ldots 7, u = 0 \ldots 7$ array from memory into register file needs 16 `LOAD` operations, that is, 8 cycles. Thus, the total overhead for firing-up and flushing the software pipeline is 32 cycles.

In order to assess the implications of the loop prologue and epilogue in a real case, we have focused on the average number of coded blocks per slice for a number of MPEG-conformance bit-strings (Table 5.1). If all the blocks in an MPEG slice are first reconstructed and only then transformed as a single batch, then the lowest average batch size is 38 blocks/slice (B frames in the *popplen* scene). This figure translates into the worst case penality associated to the prologue and epilogue of the software pipeline loop of $32/38 \approx 0.84$ cycles/block. Since this overhead represents about 2.5% of the 32 cycle/block throughput in the most disadvantageous case, it can be neglected.

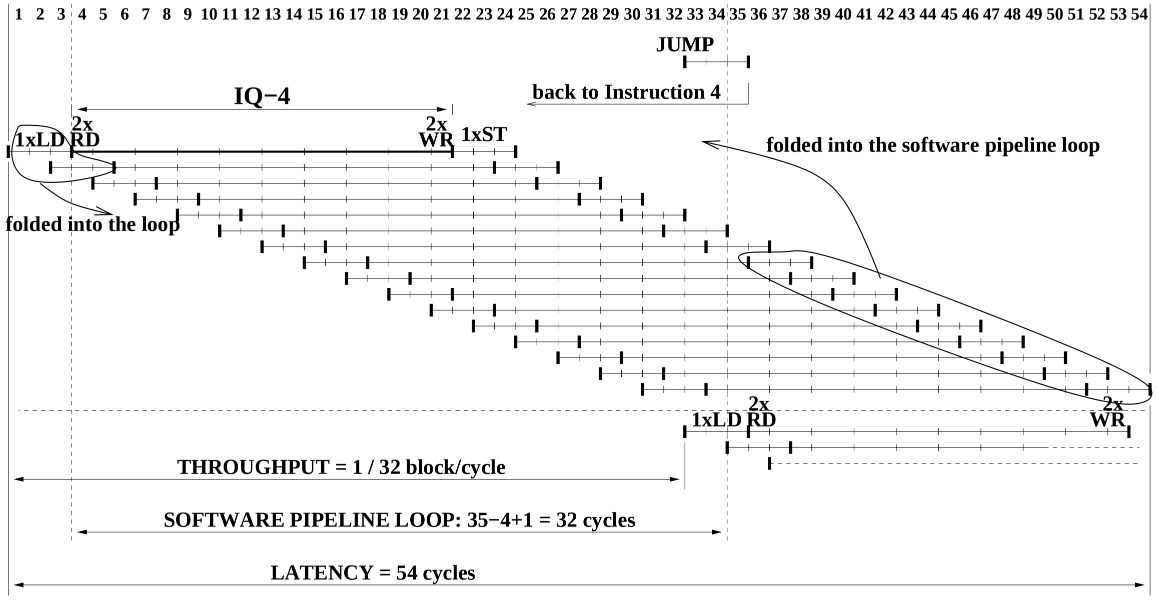


Figure 5.2: **Schedule result for an IQ-4 unit having the latency of 18 and recovery of 2 (LD stands for `LOAD`, RD for read, WR for write, and ST for `STORE`).**

Table 5.1: **The average number of coded blocks per slice – from Appendix A.**

Scene		Coded blocks/slice		
	Frame type	I	P	B
batman		–	257	234
popplen		264	80	38
sarnoff2		270	171	61
tennis		264	167	71
ti1cheer		264	155	88

Referring again to Figure 5.2, the issue-slot occupancy for a single IQ-4 unit having a recovery of 2 cycles is as follows:

- 16 double-slot IQ-4 operations (slot: 1+2) = 32 slots
- 16 single-slot `LOAD` operations (slot: 4 or 5) = 16 slots
- 16 single-slot `STORE` operations (slot: 4 or 5) = 16 slots
- 1 single-slot `JUMP` operations (slot: 3) = 1 slot
- cycling condition computation (slot: 3) = 3 slots
- *some pointer arithmetic* (slot: 3) = about 6 slots

Thus, the global occupancy figure is 74 out of $32 \times 5 = 160$, which means that 2.31 out of 5 issue-slots are filled in with operations. There are plenty of free slots that can be utilized for other purposes, e.g., implementing an additional pure-software IQ. Thus, the throughput figure of 1/32 blocks/cycle represents the lower bound of the performance improvement.

Assuming that enough FPGA I/O pins are available, an IQ-8 computing unit can be mapped on the reconfigurable array. The syntax of the new EXECUTE (3-slot) operation is:

EXECUTE <IQ-8> R_QF_l, R_QF_r, R_W, R_qs, R_param → R_F_l, R_F_r

In order to schedule all the `STORE` operations associated to the IQ-8 operation in the next iteration IQ-8 is launched, the loop should be folded at cycle 22, as depicted in Figure 5.3. This translates to a throughput of 1/18 block/cycle.

A better solution is to process the blocks in chuncks of two, which translates to a throughput of 2/32 = 1/16 block/cycle (Figure 5.4). However, an additional overhead of 16 cycles is encountered when the number of blocks to be processes is odd, since the last iteration of the software pipeline dequantize a dummy block. In the most disadvantageous case (38 blocks/slice for B frames in the *popplen* scene – Table 5.1), this translates to the additional penalty of 8/38 = 0.21 cycles/block, which is a good trade-off for the 2 cycles gained by this procedure.

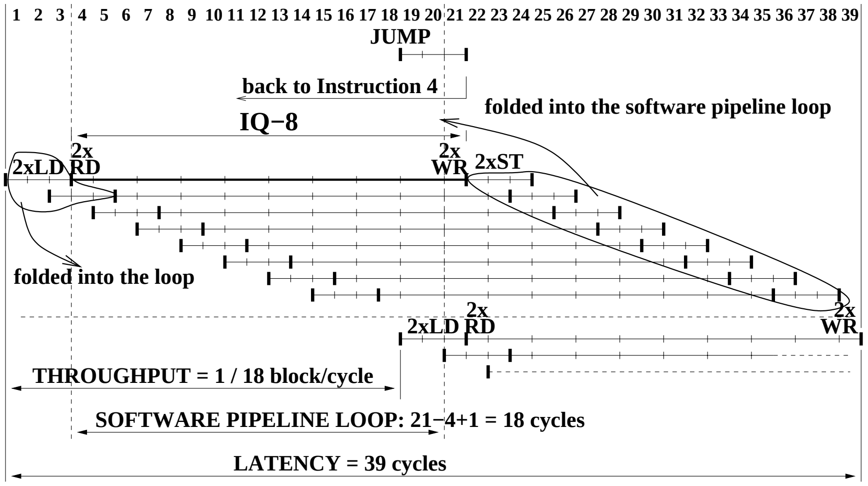


Figure 5.3: **Schedule result for an IQ-8 unit having the latency of 18 and recovery of 2 (LD stands for `LOAD`, RD for read, WR for write, and ST for `STORE`).**

Due to the FPGA pin limitation problem, providing more raw hardware simultaneously with a larger number of I/O pins can be achieved by augmenting the TriMedia core with multiple RFUs. Due to the need to issue instructions to both RFUs, a larger utilization of the issue slots will be encountered over a solution with a single FPGA having a large number of I/O pins.

Assuming that two RFUs are available, and each of such RFUs contains an EP1K100 FPGA, two IQ-4 units can be used: IQ-4-a, and IQ-4-b, each having a recovery of 2 cycles. This way, an IQ-4 unit with the recovery of 1 cycle is emulated. Since the IQ-4 is re-entrant, each of the IQ-4 units should process a separate block. That is, two blocks are processed in parallel, as depicted in Figure 5.5. In this case, the issue-slot occupancy for two IQ-4 units, each having a recovery of 2 cycles is as follows:

- 32 double-slot IQ-4 operations (slot: 1+2) = 64 slots
- 32 single-slot `LOAD` operations (slot: 4 or 5) = 32 slots
- 32 single-slot `STORE` operations (slot: 4 or 5) = 32 slots
- 1 single-slot `JUMP` operations (slot: 3) = 1 slot
- cycling condition computation (slot: 3) = 3 slots
- *some pointer arithmetic* (slot: 3) = about 12 slots

Thus, the global occupancy figure is 144 out of $32 \times 5 = 160$ slots, which means 4.50 out of 5 issue-slots are filled in with operations.

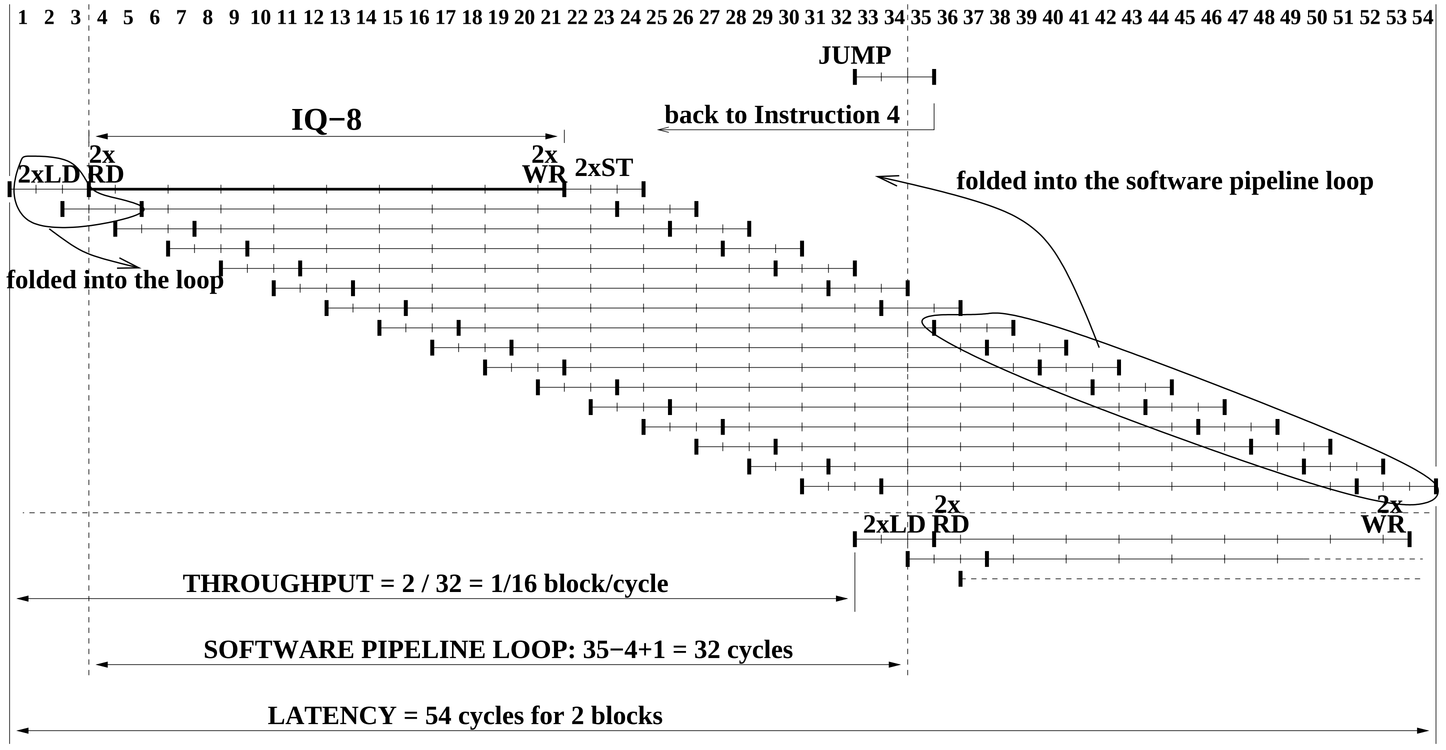


Figure 5.4: **Schedule result for an IQ-8 unit having the latency of 18 and recovery of 2. Two 8×8 blocks are processed per loop iteration (LD stands for `LOAD`, RD for read, WR for write, and ST for `STORE`).**

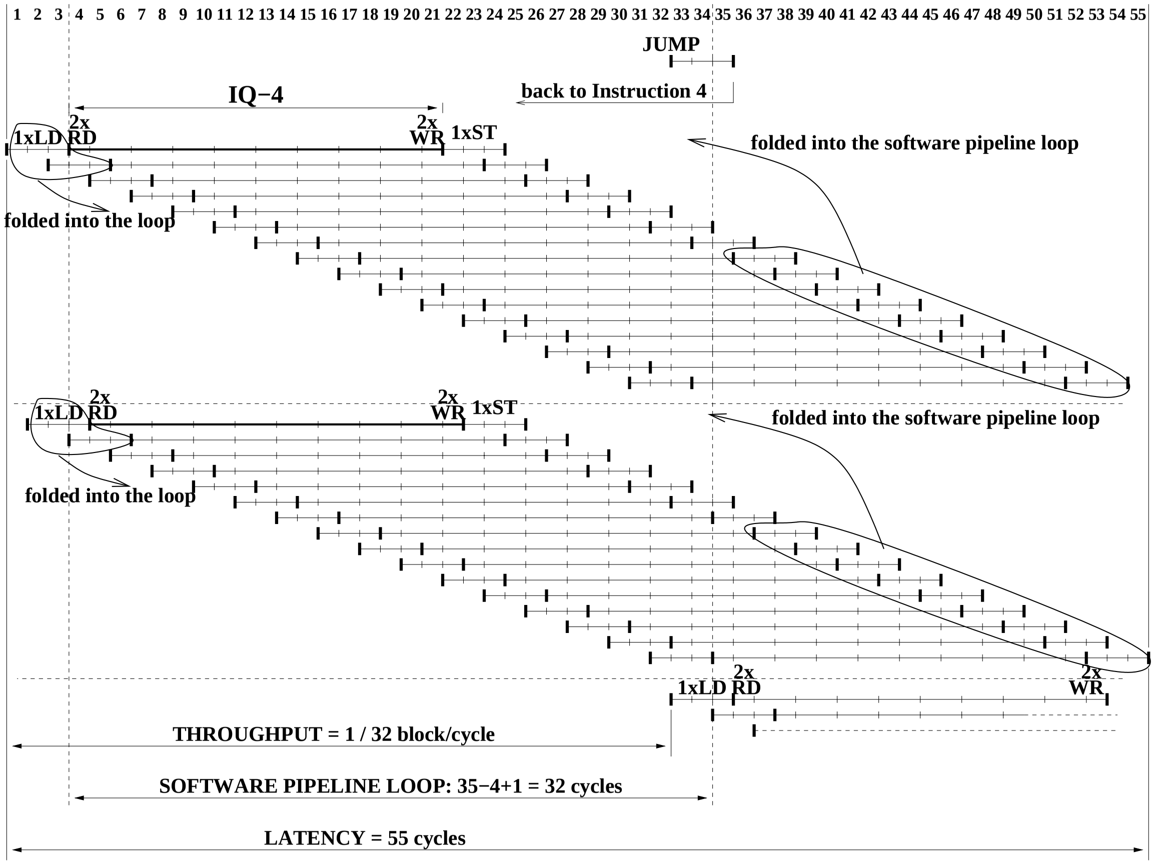


Figure 5.5: **Schedule result for two IQ-4 units, each having the latency of 18 and recovery of 2 (LD stands for `LOAD`, RD for read, WR for write, and ST for `STORE`).**

If a single block is to be dequantized per loop iteration, then non-re-entrant IQ computing units are needed, since the mismatch has to be carried out in software. However, assuming two units IQ-4-a, IQ-4-b, only IQ-4-a can process intra-coded DC coefficients. Since the mismatch is carried out in software, the last pipeline stage of the IQ computing unit is not needed any longer, and the latency of IQ-4 is 16 cycles instead of 18. However, the operation should return also mismatch information in order to allow the mismatch operation to be carried out in software. The syntax of the `EXECUTE` operations is:

EXECUTE <IQ-4-a> R_QF, R_W, R_qs, R_param → R_F, R_mismatch

EXECUTE <IQ-4-b> R_QF, R_W, R_qs, R_param → R_F, R_mismatch

The attempt to process a single block per iteration encounters the same problem of achieving a throughput of only 1/18 block/cycle, as it can be easily observed in Figure 5.6. In connection to this figure, we would like to mention the following issues:

- Folding the loop at Cycles 3 and 19 in order to have a throughput of 1/16 blocks/cycle is not an easy task. Since the first IQ-4 operation still needs 3 cycles to complete, and since 16 cycles are needed to store the dequantized information to memory, a total of 19 cycles should be folded into the 16-cycle software pipeline loop. This is possible only if the `STORE` operations are scheduled two loop iterations away from the moment of launching the corresponding IQ-4 operations. Although this approach is indeed possible, the higher complexity of the loop makes programming directly in assembly difficult.

- In order to keep the complexity of the software pipeline loop at a level where programming directly in assembly is feasible, we fold the loop at cycle 21, thus 1+16 = 17 cycles are folded into the 18-cycle software pipeline loop. This way, the throughput is 1/18 blocks/cycle.

- As a rule of thumb, in order to have a simple software pipeline and, therefore, being able to program directly in assembly language, the length of the loop should be greater than the latency of the FPGA-mapped operation. This constitutes the rationale for an optimization effort aimed to reduce the latency of the FPGA-mapped operation, as we already spent for the FPGA–based 1-D IDCT.

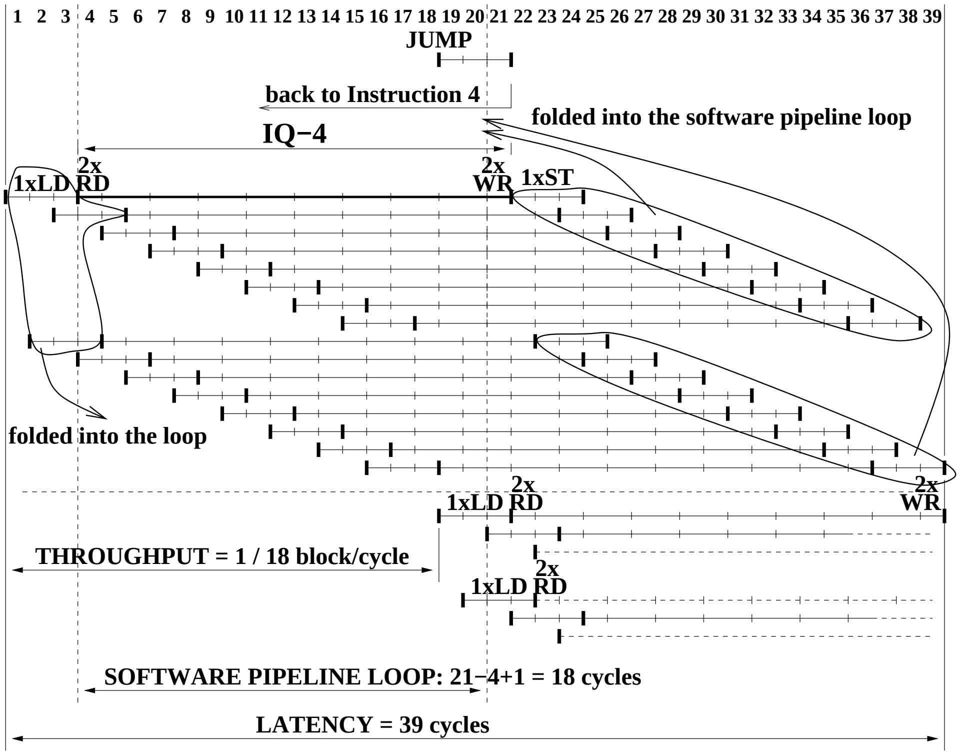


Figure 5.6: **Schedule result for two IQ-4 units, each having the latency of 18 and recovery of 2. A single 8×8 block is processed per loop iteration (LD stands for `LOAD`, RD for read, WR for write, and ST for `STORE`).**

The issue-slot occupancy for two IQ-4 units, each having a recovery of 2 is:

- 16 double-slot IQ-4 operations (slot: 1+2) = 32 slots
- 16 single-slot `LOAD` operations (slot: 4 or 5) = 16 slots
- 16 single-slot `STORE` operations (slot: 4 or 5) = 16 slots
- 1 single-slot `JUMP` operations (slot: 3) = 1 slot
- cycling condition computation (slot: 3) = 3 slots
- *some pointer arithmetic* (slot: 3) = about 6 slots

Thus, the global occupancy figure is 74 out of $18 \times 5 = 90$ slots, which means that 4.11 out of 5 issue-slots are filled in with operations.

In the last experiment we assume a hypothetical FPGA architecture on which only circuitry having the clock frequency smaller or at most equal to one-quarter than TriMedia frequency can be implemented. Thus, the recovery of any computing unit mapped on such FPGA is at least 4 cycles. In order to emulate an IQ-4 unit with recovery of 1 cycle, we also assume that four RFUs are available, on each RFU an IQ-4 unit being configured: IQ-4-a, IQ-4-b, IQ-4-c, IQ-4-d. These IQ operations are launched in four phases: a, b, c, d. An assembly code sample is presented in Algorithm 3.

Algorithm 3 IQ-4 with four phases

```
...
(* cycle 4 *)
    IF r(1) EXECUTE ->
    <IQ_4_a> ->
    nop,
    nop,
    IF r(1) ldd_h(32) r(5) -> r(9);
(* cycle 5 *)
    nop,
    IF r(1) std_h(8) r(5) -> r(7),
    IF r(1) ldd_h(16) r(5) -> r(8);
(* cycle 6 *)
    IF r(1) EXECUTE ->
    <IQ_4_c> ->
    nop,
    nop,
    IF r(1) ldd_h(8) r(5) -> r(7);
(* cycle 7 *)
    IF r(1) EXECUTE ->
    <IQ_4_d> ->
    nop,
    nop,
    IF r(1) ldd_h(8) r(5) -> r(7);
(* cycle 8 *)
    IF r(1) EXECUTE ->
    <IQ_4_a> ->
    nop,
    nop,
    IF r(1) ldd_h(8) r(5) -> r(7);
(* cycle 9 *)
    IF r(1) EXECUTE ->
    <IQ_4_b> ->
    nop,
    nop,
...
```

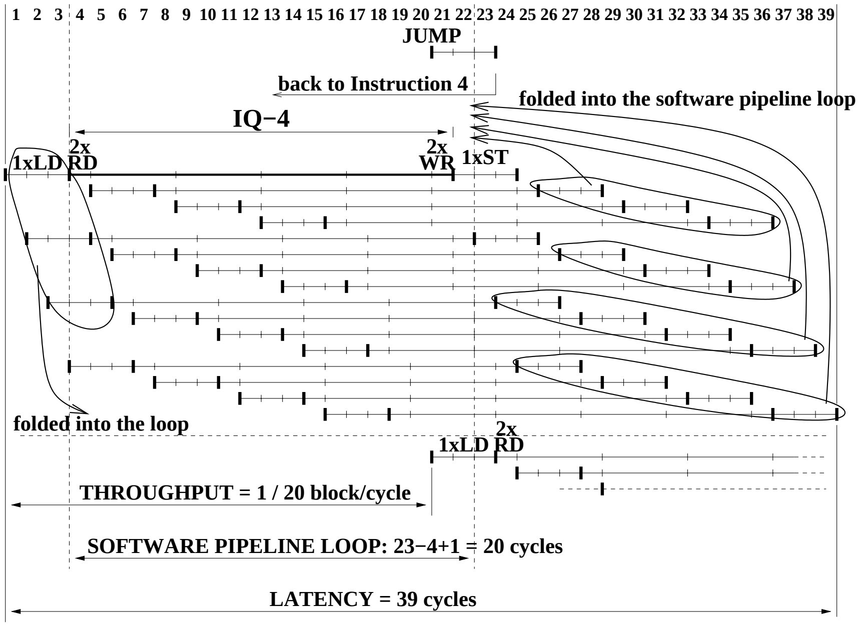


Figure 5.7: **Schedule result for four IQ-4 units, each having the latency of 18 and recovery of 2 (LD stands for `LOAD`, RD for read, WR for write, and ST for `STORE`).**

Figure 5.7 presents the scheduling of the IQ routine when four IQ-4 units, each having a recovery of 4 cycles, are mapped on reconfigurable array. Since the loop is folded at Cycles 4 and 23, a throughput of 1/20 blocks/cycle is achieved. The slot occupancy is as follows:

- 16 double-slot IQ-4 operations (slot: 1+2) = 32 slots
- 16 single-slot `LOAD` operations (slot: 4 or 5) = 16 slots
- 16 single-slot `STORE` operations (slot: 4 or 5) = 16 slots
- 1 single-slot `JUMP` operations (slot: 3) = 1 slot
- cycling condition computation (slot: 3) = 3 slots
- *some pointer arithmetic* (slot: 3) = about 6 slots

which translates to a global occupancy figure of 74 out of $20 \times 5 = 100$ slots, which means that 3.70 out of 5 issue-slots are filled in with operations.

Table 5.2: **Performance figures for IQ.**

Scene (*.m2v)	Block type	Workload (blocks)	**Pure software** (cycles)	1 RFU@100MHz (cycles)	2 RFUs@100MHz (cycles)	4 RFUs@50MHz (cycles)
batman	I	–	n/a	n/a	n/a	n/a
	P	36,943	1,921,036	1,185,632	595,584	742,316
	B	25,295	1,315,340	812,032	408,192	508,492
	Total	62,238	3,236,376	1,997,664	1,003,776	1,250,808
	Improvements	–	n/a	**38.3%**	69.0%	61.6%
popplen	I	3,960	154,440	127,080	63,720	79,560
	P	3,606	187,512	116,472	59,224	73,200
	B	1,145	59,540	37,360	19,376	23,620
	Total	8,711	401,492	280,912	142,320	176,380
	Improvements	–	n/a	**30.0%**	64.6%	56.1%
sarnoff2	I	8,100	315,900	259,920	130,320	162,720
	P	5,120	266,240	164,560	82,896	103,120
	B	3,645	189,540	118,080	60,192	74,340
	Total	16,865	771,680	542,560	273,408	340,180
	Improvements	–	n/a	**29.7%**	64.6%	55.9%
tennis	I	9,504	370,656	304,992	152,928	190,944
	P	17,992	935,584	578,336	291,424	362,432
	B	10,250	533,000	331,456	168,416	208,456
	Total	37,746	1,839,240	1,214,784	612,768	761,832
	Improvements	–	n/a	**34.0%**	66.7%	58.6%
ti1cheer	I	7,920	308,880	254,160	127,440	159,120
	P	4,481	233,012	144,088	72,600	90,316
	B	5,275	274,300	170,240	86,272	106,940
	Total	17,676	816,192	568,488	286,312	356,376
	Improvements	–	n/a	**30.3%**	64.9%	56.3%

In Table 5.2 we present the computing performance reported in cycles for a number of MPEG-conformance bit-strings (scenes). We also present the relative improvement achieved on ρ–TriMedia over standard TriMedia–CPU64 for each of the analysed scenarios. When a single RFU@100 MHz is available, the improvement is about 32.5% in terms of cycles, which translates to a speed-up of $1.5\times$. Assuming that two RFUs@100 MHz are available, the improvement increases to about 66.0% (speed-up of $2.9\times$), while for four RFUs@50 MHz an improvement slightly lower is achieved (57.7% or $2.4\times$ speed-up).

In all the experiments we have not counted the cache misses. The rationale behind this approach is as follows. Since the code is simple and symetrical, a routine which contains the entire information needed to perform an 8×8 block dequantization can be built. The benefic effect is that prefetch operations can be scheduled, and so most of the cache misses can be avoided [116].

We would like to note that, as opposed to IDCT, each and every pixel can processed independently in an IQ task. Thus, an IDCT-like true dependency is not encountered in an IQ application. The immediate effect is that the number of DCT coefficients that can be dequantized simultaneously is subject to FPGA size and the number of issue slots. Considering a single FPGA–mapped IQ-4 unit having a recovery of 2 cycles, we showed that only 3 out of 5 issue slots are filled in with operations in the ρ–TriMedia implementation. As mentioned, when enough raw hardware is available, two IQ-4 units can be configured on the reconfigurable core. Assuming both units have a recovery of 2 cycles, an IQ unit with recovery of 1 cycle can be emulated by calling the first IQ-4-a unit on even cycles, and the second IQ-4-b unit on odd cycles, as depicted in Figures 5.5 and 5.6. This way, the computing power is large enough to dequantize the coefficients at a rate close to the transfer capability with the memory (throughput of 16 cycles/8×8 block for 2 LOAD/STORE units). However, in this case the issue-slot occupancy increases from 3 to 4.5 out of 5 issue slots.

Finally, we would like to mention that any optimization which would make use of the small number of non-zero coefficients (which ranges from 5 to 10 per block), thus would reduce the number of IQ-related operations, is equally applicable to pure-software and FPGA-based solutions. Thus, the percentage improvement is likely to remain the same.

5.5 Conclusion

In this chapter we investigated inverse quantization, and proposed a reconfigurable IQ design for ρ–TriMedia. Essentially, we implemented an IQ-4 unit on reconfigurable hardware, which can dequantize four elements per call. Assuming that a single ACEX 1K device is available as raw hardware, by calling sixteen IQ-4 operations in a row, an 8×8 block is fully inverse quantized on ρ–TriMedia with the average improvement of 32.5% in terms of cycles (speed-up of $1.5\times$) over standard TriMedia–CPU64.

As an interesting remark, we note that only the pair issue-slot 1+2 is used to launch the FPGA–mapped IQ operations, while the slots 4 and 5 are occupied with `LOAD` and `STORE` operations. Thus, the Slot number 3 is available to carry out additional computation, e.g., pointer arithmetic, condition evaluation for the `JUMP` operation, etc. Moreover, since the issue slots are used for rather independent tasks (slot pair 1+2 for $\mathtt{IQ}-4$, slots 4 and 5 for `LOAD` and `STORE`, slot 3 for extra computation), there is an increasing interest to split the register file into two pieces, with benefic effects in timing and silicon area. We leave this issue as an open question for further work.

Chapter 6

Entropy Decoding

In Chapters 4 and 5 we showed that significant improvements had been achieved on ρ–TriMedia over standard TriMedia-CPU64 for tasks that exhibited data- and instruction-level parallelism. In this chapter, we assess the improvement when the parallelism is not available, in particular for an Entropy Decoding task.

Entropy decoding consists of Variable-Length Decoding (VLD) [103, 70] followed by a Run-Length Decoding (RLD), both VLD and RLD being sequential tasks. Due to data dependency, entropy decoding is an intricate function on TriMedia, since a VLIW architecture must benefit from instruction-level parallelism in order to be efficient. For this reason, such a function is an ideal candidate to benefit from reconfigurable hardware support.

In this chaper, we demonstrate that a significant improvement over a pure-software solution is possible if a VLD computing unit is configured on FPGA. In particular, we show that a VLD instruction that can return two symbols per call (VLD-2) leads to the most efficient entropy decoder in terms of instruction cycles. We also demonstrate that further improvements are still possible if part of the functionality related to RLD, e.g., computing the absolute position of the non-zero coefficients within the 8×8 matrix, is also mapped on FPGA.

The chapter is organized as follows. For background purpose, we briefly present the most important theoretical issues related to entropy decoding in Section 6.1. An entropy decoding pure-software solution is described in Section 6.2. Implementation issues of the VLD-2 computing unit on FPGA are presented in Section 6.3. The execution scenario of the Entropy Decoder on the FPGA-augmented TriMedia-CPU64, and experimental results are presented in Sections 6.4, and 6.5, respectively. The final section completes the chapter with some conclusions and closing remarks.

6.1 Theoretical background

As mentioned in Chapter 3, in MPEG the video data is essentially processed in 8×8-element blocks. For a lossless compression following the Discrete Cosine Transform (DCT) and Quantization stages, the block is first transformed into a vector by a zig-zag operation. This vector, which contains large series of zeros, is then sent to an Entropy Coder that consists of a Run-Length Coder (RLC) and a Variable-Length Coder (VLC). The RLC represents consecutive zeros by their *run* lengths, and generates composite words, referred to as *symbols*. Thus, such a symbol describes a *run-level* pair; the *run* value indicates the number of zeros by which a (non-zero) DCT coefficient is preceeded, while the *level* value represents the value of the DCT coefficient. When all the remaining coefficients in a vector are zero, they are all coded by the special symbol *end-of-block*. Variable-length coding is a mapping process between *run-level*/*end-of-block* symbols and *variable-length codewords*, which is carried out according to a set of tables defined by the standard. Not every *run-level* pair has a variable-length codeword to represent it, only the frequent used ones do. For those rare combinations, an *escape* code is given. After an *escape* code, the *run*- and *level*-value are coded using fixed-length codes.

In order to achieve maximum compression, the coded data does not contain specific guard bits separating consecutive codewords. As a result, the decoding procedure must recognize the symbol itself as well as its *code-length*. Before decoding the next symbol, the input data string has to be shifted by a number of bits equal to the decoded code-length. We would like to note that these are recursive operations that generate true-dependencies.

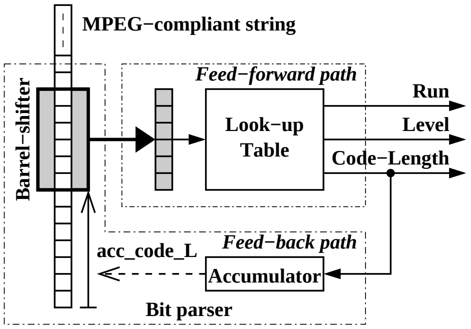


Figure 6.1: **Variable-length decoding principle.**

An **Entropy Decoder** consists of a **Variable-Length Decoder** (VLD) followed by a **Run-Length Decoder** (RLD). The input to the VLD is the incoming bit stream, and the output is the decoded symbols. As depicted in Figure 6.1, a VLD is a system with feedback, whose loop typically contains a *Look-Up Table* (LUT) on the feed-forward path and a *bit parser* on the feed-back path. The LUT receives the variable-length code itself as an address [63] and outputs the decoded symbol (*run-level* pair or *end_of_block*) and the codeword length, *code_length*. In order to determine the starting position of the next codeword, the *code_length* is fed back to an accumulator and added to the previous sum of codeword lengths, *accumulated code_length*, or `acc_code_L`. The bit parsing operation is completed by the *barrel-shifter* (or *funnel-shifter*) which shifts out the decoded bits.

With respect to the hardware complexity, we would like to note that the longest codeword excluding Escape has 17 bits. Therefore, the LUT size reaches $2^{17} = 128$ K words for a direct mapping of all possible codewords. Regarding the performance of a variable-length decoder, it is worth mentioning that the throughput of a VLD is bounded by a value inverse to the latency of the loop [65].

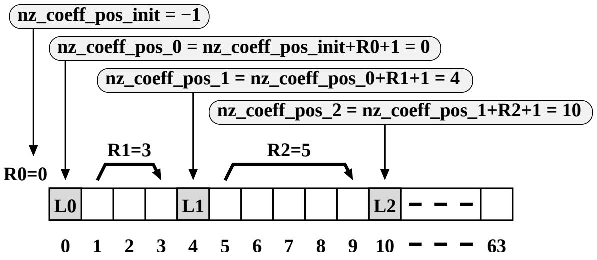

nz_coeff_pos_init = −1
nz_coeff_pos_0 = nz_coeff_pos_init+R0+1 = 0
nz_coeff_pos_1 = nz_coeff_pos_0+R1+1 = 4
nz_coeff_pos_2 = nz_coeff_pos_1+R2+1 = 10
R0=0
R1=3
R2=5
L0
L1
L2
0 1 2 3 4 5 6 7 8 9 10 63

Figure 6.2: **Run-length decoding principle.**

Conceptually, for each *run-level* pair returned by the VLD, the run-length decoder outputs the number of zeros specified by the *run* value and then pass the *level* through. In a programmable processor platform, a way to optimize this process is to fill in an empty vector with *level* values at positions defined by *run* values, as depicted in Figure 6.2: the non-zero coefficient position, `nz_coeff_pos`, is computed by adding the *run* value, `R`, and an ’`1`’ to the position of the previous non-zero coefficient. This common strategy has been used in previous work [74, 82, 96] and will be used subsequently, too.

MPEG2–compliant Variable-Length Decoding. As we mentioned in the previous chapters, we mainly focus on MPEG2 decoding in this dissertation. Thus, several elements particular to MPEG2 standard regarding variable-length decoding are worth to be provided.

MPEG-2 defines four Variable-Length Code (VLC) tables for encoding the DCT coefficients: B12, B13, B14, and B15 [74]. Which table is used depends on the type of image – intra (I) or non-intra (NI), luminance (Y) or chrominance (C) – and a bit-field, **`intra_vlc_format`**, in the macroblock header, as shown in Table 6.1. In general, this means that a single stream uses all tables, and the tables can be switched per macroblock and/or block.

`intra_vlc_format`			0	1
I	DC coefficient	Y	B12	B12
		C	B13	B13
	AC coefficient		B14	B15
NI	1^{st} & subsequent coefficient		B14	B14

Table 6.1: **Selection of VLC tables**

The process of reconstructing an 8×8 block starting from an MPEG-2 string is exemplified in Figure 6.3 for an intra-coded chrominance block and `intra_vlc_format` = 1. In this example, the *level* of the DC component is -2. Then, four AC components are coded, the *run-level* pair having the values 4/+1, 0/-3, 5/-5 (which is coded as an *Escape* sequence), and 7/+1, respectively. The last symbol of the block is the composite *end-of-block*.

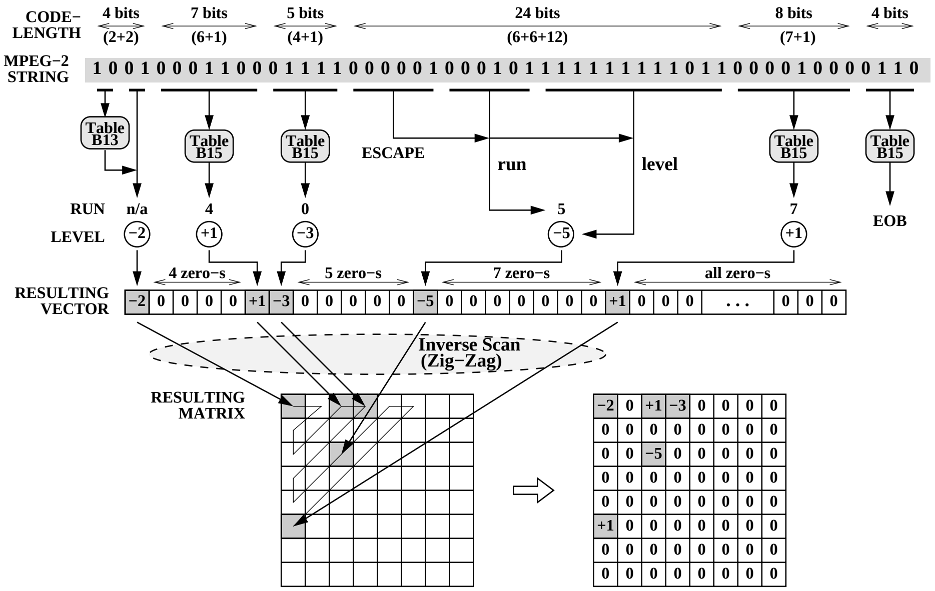


Figure 6.3: **MPEG2–compliant entropy decoding example.**

For most of the DCT coefficients, the decoding follows the “normal” rule. The maximum *code-length* is 16 bits plus a sign bit. The variable-length code determines the *run* and *level* values, while the sign bit indicates the sign of the *level* value. Apart from this rule, there are a few exceptional cases to be dealt with:

1. **The DC coefficient** for intra macroblocks, which is encoded through the B12/B13 tables, depending on the block type: luminance or chrominance.
2. The 6-bit **Escape** symbol, which is followed by the fixed-lengths 6-bit *run* and 12-bit signed *level*.
3. The **end-of-block** symbol, which is coded with 2 or 4 bits, depending on the `intra_vlc_format` bit.

In the next section we present an entropy decoder implementation on standard TriMedia–CPU64.

6.2 Entropy decoder pure-software implementation

According to the reference implementation [82], the VLD is implemented as a repeated table-lookup. Each lookup analyzes a fixed size chunk of bits (for example, `LOOKUP_ADDRESS_WIDTH` = 6 or 8) and determines if a valid code was encountered or some more bits need to be decoded. In any case, the number of consumed bits ranging from the smallest variable-length code to the chunk size is generated. In case of a valid decode, i.e., *hit*, a *run-level* pair is generated, or an *escape* or *end_of_block* flag is set. If a *miss* is detected, i.e., more bits are needed for a valid decode, an offset into the VLC table for a second- or third-level lookup, *table_offset*, is generated. This process of signaling an incomplete decode and generating a new offset may be repeated a number of times, depending on the largest variable-length code and chunk size. The following basic stages can be discerned in the reference implementation of the entropy decoder on TriMedia–CPU64:

1. **Initializations**.
2. **Barrel-shift the VLC string** according to the *accumulated code-length* value.
3. **Table look-up** (look-up address computation and the proper table look-up). The table look-up returns a 32-bit word containing all the fields mentioned at Stage 4.
4. **Field extraction**: *run*, *level*, *code_length*, *valid_decode*, *end_of_block*, *escape*, *table_offset*.
5. **Update (modulo-64) the accumulated code-length**:
   acc_code_length = *acc_code_length* + *code_length*
   If an overflow has been encountered, **advance the VLC string** by 64 bits.
6. **Exit** the loop if *end_of_block* has been encountered.
7. **Handle escape** if *escape* has been encountered.
8. **Run-length decoding**: inverse zig-zag process, followed by filling-in an empty 8×8 matrix.
9. **Go to** Stage 2.

The Stage 8 – **run-length decoding** – is folded into the loop, such that loop pipelining is employed [82]. That is, the run-length decoding for the previously decoded symbol is carried out in the same iteration with the variable-length decoding of the current symbol.

Updating the *acc_code_length* value is carried out modulo-64. The main idea is to match this process with the bus width of TriMedia–CPU64. That is, a new chunk of 64 bits of information to be decoded is read into register file on overflow. Also, we would like to note that the VLC-related information is stored into the lookup table in a packed format, as 32-bit unsigned integers, as depicted in Table 6.2. Thus, a sequence of masking and shifting operations is needed to extract these fields.

Table 6.2: **The original VLC table format.**

	end-of-block (stop)	escape	valid decode	run	level	table offset	code-length
No. of bits	1	1	1	5	8	12	4
Position	31	30	29	28-24	23-16	15-4	3-0

To make the presentation self consistent, the reference implementation of the entropy decoding routine is presented in Algorithm 4. All identifiers starting with a capital letter are regarded as constants. The routine consists of a first **for** loop (lines 3–41) cycling over all coded macroblocks in the MPEG string, a second **for** loop (lines 4–40) cycling over all coded blocks in a macroblock, and an inner (infinite) loop labeled **loop** (lines 10–39), cycling over all DCT coefficients in a block. This inner loop is left when an *end_of_block* is encountered (lines 27–29).

The initializations for block-level decoding are performed at lines 5–8. The table look-up, i.e., variable-length decoding, is carried out at lines 11–13. Lines 15–18 implement run-length decoding, which has been folded into the loop to employ loop pipelining. Field extraction is performed at line 20. The barrel-shifting (line 11) is done on an 128-bit field, by means of the TriMedia custom operation:

`bitfunshift Rsrc_1 Rsrc_2 Rsrc_3` $\rightarrow$ **`Rdest_1 Rdest_2`**

where `Rsrc_1` and `Rsrc_2` are the two 64-bit registers storing the leading 128 bits of the VLC string to be shifted, the `Rsrc_3` defines the shifting value, and `Rdest_1` and `Rdest_2` are the two 64-bit registers storing the 128-bit shifted string. Obviously, only the value stored into `Rdest_1` register will be used for the look-up procedure. It should be mentioned that since *acc_code_length* is updated modulo-64 (lines 30–32), at least 47 bits are available in `Rdest_1` for the next decoding iteration in the worst case (this can be easily verified by assuming that *acc_code_length* = 63 at line 34).

A particular optimization technique has been used in order to keep the most likely iteration (that is when new incoming bits from the MPEG string are not needed for continuing the decoding process, and none of the *escape*, *end_of_block*, and *error* conditions is raised), as short as possible. According to this technique, the *escape* flag is also set to '1' when any of the *escape*, *end_of_block*, or *error* conditions occurs. In this way, a jump to the beginning of the inner loop is taken when none of the above mentioned conditions is raised (lines 24–26). All the exceptional cases are managed after this jump: *end_of_block* at lines 27–29, modulo-64 updat-

Algorithm 4 Entropy decoder routine – reference implementation

```
 1: set-up the test-bench (store the VLC lookup table, read the VLC_string into memory, etc.)
 2:
 3: for $i = 1$ to NO_OF_MACROBLOCKS do
 4:   for $j = 1$ to NO_OF_BLOCKS_IN_MACROBLOCK do
 5:     table_offset ← FIRST_TABLE_OFFSET
 6:     nz_coeff_pos_ZZ ← 0
 7:     run ← 0
 8:     valid_decode ← 0
 9:
10:     loop
11:       barrel-shift the VLC_string with acc_code_length positions
12:       lookup_address ← the leading LOOKUP_ADDRESS_WIDTH bits from VLC_string
13:       lookup_address ← lookup_address + table_offset
14:       retrieved_32_bit_word ← VLC_table[lookup_address]
15:
16:       nz_coeff_pos_ZZ ← nz_coeff_pos_ZZ + run
17:       nz_coeff_pos ← invZZ_table[nz_coeff_pos_ZZ]
18:       $8 \times 8$_matrix[nz_coeff_pos] ← level
19:       nz_coeff_pos_ZZ ← nz_coeff_pos_ZZ + valid_decode
20:
21:       extract code_length, run, level, table_offset, escape, valid_decode, end_of_block from
          retrieved_32_bit_word
22:
23:       acc_code_length ← acc_code_length + code_length
24:       if acc_code_length $\leq 64$ and not(escape) then
25:         continue { ————————————-> go to loop}
26:       end if
27:       if end_of_block flag is raised then
28:         break { ——————————————-> initiate the next for iteration (block-level)}
29:       end if
30:       if acc_code_length $\geq 64$ then
31:         advance the VLC_string by 64 bits
32:         acc_code_length ← acc_code_length - 64
33:       end if
34:       if escape flag is raised then
35:         run ← next 6 bits from VLC_string
36:         level ← next 12 bits from VLC_string
37:         acc_code_length ← acc_code_length + 6 + 12
38:       end if
39:     end loop
40:   end for
41: end for
```

ing and advancing the VLC string at lines 30–33, and *escape* at lines 34–38. It should be mentioned that there is no flag to indicate an *error* condition. When an *error* is encountered, *end_of_block* = 1 and *valid_decode* = 0 simultaneously. Since the *end_of_block* flag is set the loop is left. However, it is still the responsibility of the entropy decoder calling routine to detect if a valid *end_of_block* has been encountered or an *error* has occured.

Regarding the efficiency of the reference implementation, we would like to mention that only variable-length decoding for the first DCT coefficient is carried out during the first iteration of the loop **loop** (lines 10–39 in Algorithm 4), while the code associated to run-length decoding performs a dummy action. That is, there is an overhead of one iteration to fire-up the software pipeline. Since the number of non-zero DCT coefficients in a block is small, ranging, for example, between 3.3 and 5.8 for non-intra macroblocks [82], the number of iterations per block is also small. Consequently, this overhead can be significantly large.

To improve the performance of the pure-software entropy decoder, we propose the following changes with respect to the reference implementation. For reasons that will become relevant later in the chapter, only the decoding of non-intra macroblocks is addressed. That is, the VLC Table B14 of the MPEG standard is assumed if we do not state otherwise.

- **The prologue of the pipelined loop** [57] **is exposed to the compiler**. Since the VLC table does not have "holes" in the region of short codewords (i.e., each and every entry in that region corresponds either to a short codeword that is decoded in a single iteration, or to a long codeword that is decoded in two or more iterations), there are no incoming bit combinations that do not have a meaning within the prologue. Therefore, an *error* condition is never raised within the prologue. Moreover, since an *end_of_block* symbol is not allowed for the first coefficient in a block, an *end_of_block* condition is never encountered within the prologue, too. Thus, testing the *end_of_block* flag (lines 27–29 in Algorithm 4) within the prologue becomes superfluous, and a simple code consisting of a first-level look-up, folowed by an extraction of the code_length, run, level, lookup_address_width, table_offset, escape, valid_decode (and, notable, no extraction of the *end_of_block* flag) can efficiently fire-up the software pipeline.

- **Barrel-shifting is carried out by means of an extended `bitfunshift` TriMedia specific operations**.

`bitfunshift_3 Rsrc_1 Rsrc_2 Rsrc_3 Rsrc_4` → **`Rdest_1 Rdest_2`**

The main idea is to gain flexibility over the modulo-64 operation by performing the barrel-shift operation on $3 \times 64 = 192$ bits instead of $2 \times 64 = 128$ bits. In this way, the modulo-64 operation can be postponed, since additional 64 bits are available for decoding over the standard implementation.

- **The lookup returns a 64-bit value instead of a 32-bit value**. The fields *code_length*, *run*, *level*, *lookup_address_width* (which defines the chunk size of the next look-up), *table_offset*, *escape*, *valid_decode* (which signals a hit), and *end-of-block* fields are each stored within the boundaries of a byte (that is, in an unpacked way instead of a packed one), as shown in Table 6.3. Since extracting a byte from a 64-bit value takes only 1 cycle on TriMedia–CPU64, our solution is faster than using a pair of masking and shifting operations required by the 32-bit approach. The cost of such approach is a double-size look-up table. It is still an open question which approach is better with respect to a particular TriMedia cache size, as the cache misses may become dominant when the evaluation is made for a complete MPEG decoder.

- **The chunk size is variable**, which leads to a more compact look-up table. According to our experiments, there are enough empty slots in the TriMedia VLIW instruction format for an entropy decoding task. Consequently, a variable chunk size does not introduce real dependencies.

Table 6.3: **The proposed VLC table format.**

	code length	run	level	table offset	lookup address width	escape	valid decode	EOB
No. of bits	8	8	8	8	8	8	8	8
Position	63-56	55-48	47-40	39-32	31-24	23-16	15-8	7-0

In connection to the Table 6.3, several comments should be provided. The VLC table is an one-dimensional array of vectors, where each vector contains eight unsigned bytes. In order to keep the number of instructions as low as possible, we propose to store the sign bit of each and every codeword into the lookup table. According to Table B14, the *level* value ranges between $-40 \cdots +40$. Thus, 7 bits (less than 1 byte) are sufficient to represent all the values. However, precautions have to be taken to convert *level* to a signed integer after extraction (Algorithm 5).

Algorithm 5 Converting the *level* from 8-bit unsigned to a 16-bit signed integer

```
#define LEVEL_FIELD 5

int16 level;

retrieved_vec64ub = VLC_table[ lookup_address];
level = (int16) ub_get( retrieved_vec64ub, LEVEL_FIELD);
level = (int16)((level ≪ 24) ≫ 24); /* 32-bit processing */
```

The least significant byte has been allocated for *end_of_block* (EOB) flag. Since the TriMedia C compiler recognizes expressions of the form $(E_1 \& 1)$, the least significant bit of this byte is set to '1' when an *end_of_block* condition is raised. This way, the condition for leaving the loop can be written as follows:

Algorithm 6 TriMedia-specific code for testing the *end-of-block* condition

```
#define END_OF_BLOCK_FIELD 0

uint8 end_of_block;

for (;;) {
  retrieved_vec64ub = VLC_table[ lookup_address];
  end_of_block = ub_get( retrieved_vec64ub, END_OF_BLOCK_FIELD);
  if ( end_of_block & 1)
    break;
}
```

The *table_offset* field defines the partitioning of the B14 into smaller lookup tables. The B14 table has been splitted in eight tables (*first*, *second*, *third*, *forth*, *fifth*, *sixth*, *seventh*, *eighth*) which are presented subsequently. In order to improve the readness, we preserved the order of the rows as in the MPEG standard.

All eight tables are stored into memory concatenated. The number of address bits for each table corresponds to the maximum code-length of the symbols that the table is able to decode. That is, Tables *first* and *second* have each 8 address bits, Table *sixth* has 7 address bits, Tables *third* and *forth* have each 4 address bits, and Tables *fifth*, *seventh*, and *eighth* have each 5 address bits. The total size is 768 (64-bit) words, which means 6 KB.

Table 6.4: **Number of address lines, size, and offset for each VLC table.**

Table	No. of address lines (*lookup_address_width*)	Size (64-bit words)	*table_offset*
first	8	2^8 = 256	`0`
second	8	2^8 = 256	`0x100`
third	4	2^4 = 16	`0x200`
forth	4	2^4 = 16	`0x210`
fifth	5	2^5 = 32	`0x220`
sixth	7	2^7 = 128	`0x240`
seventh	5	2^5 = 32	`0x2c0`
eighth	5	2^5 = 32	`0x2e0`

Table 6.5: **First VLC partition**

VL code	Run	Level
1s	0	1
011s	1	1
0100 s	0	2
0101 s	2	1
0010 1s	0	3
0011 1s	3	1
0011 0s	4	1
0001 10s	1	2
0001 11s	5	1
0001 01s	6	1
0001 00s	7	1
0000 110s	0	4
0000 100s	2	2
0000 111s	8	1
0000 101s	9	1
0000 01	Escape	

Table 6.6: **Second VLC partition**

VL code	Run	Level
10	End of Block	
11s	0	1
011s	1	1
0100 s	0	2
0101 s	2	1
0010 1s	0	3
0011 1s	3	1
0011 0s	4	1
0001 10s	1	2
0001 11s	5	1
0001 01s	6	1
0001 00s	7	1
0000 110s	0	4
0000 100s	2	2
0000 111s	8	1
0000 101s	9	1
0000 01	Escape	

Table 6.7: **Third VLC partition**

1st prefix	VL code	Run	Level
0010 0	110 s	0	5
	001 s	0	6
	101 s	1	3
	100 s	3	2
	111 s	10	1
	011 s	11	1
	010 s	12	1
	000 s	13	1

Table 6.8: **Forth VLC partition**

1st prefix	VL code	Run	Level
0000 001	0 10s	0	7
	1 00s	1	4
	0 11s	2	3
	1 11s	4	2
	0 01s	5	2
	1 10s	14	1
	1 01s	15	1
	0 00s	16	1

Table 6.9: **Fifth VLC partition**

1st prefix	VL code	Run	Level
0000 0001	1101 s	0	8
	1000 s	0	9
	0011 s	0	10
	0000 s	0	11
	1011 s	1	5
	0100 s	2	4
	1100 s	3	3
	0010 s	4	3
	1110 s	6	2
	0101 s	7	2
	0001 s	8	2
	1111 s	17	1
	1010 s	18	1
	1001 s	19	1
	0111 s	20	1
	0110 s	21	1

Table 6.10: **Sixth VLC partition**

1st prefix	VL code	Run	Level
0000 0000	1101 0s	0	12
	1100 1s	0	13
	1100 0s	0	14
	1011 1s	0	15
	1011 0s	1	6
	1010 1s	1	7
	1010 0s	2	5
	1001 1s	3	4
	1001 0s	5	3
	1000 1s	9	2
	1000 0s	10	2
	1111 1s	22	1
	1111 0s	23	1
	1110 1s	24	1
	1110 0s	25	1
	1101 1s	26	1
	0111 11s	0	16
	0111 10s	0	17
	0111 01s	0	18
	0111 00s	0	19
	0110 11s	0	20
	0110 10s	0	21
	0110 01s	0	22
	0110 00s	0	23
	0101 11s	0	24
	0101 10s	0	25
	0101 01s	0	26
	0101 00s	0	27
	0100 11s	0	28
	0100 10s	0	29
	0100 01s	0	30
	0100 00s	0	31

Table 6.11: **Seventh VLC partition**

1st prefix	2nd prefix	VL code	Run	Level
0000 0000	001	1 000s	0	32
		0 111s	0	33
		0 110s	0	34
		0 101s	0	35
		0 100s	0	36
		0 011s	0	37
		0 010s	0	38
		0 001s	0	39
		0 000s	0	40
		1 111s	1	8
		1 110s	1	9
		1 101s	1	10
		1 100s	1	11
		1 011s	1	12
		1 010s	1	13
		1 001s	1	14

Table 6.12: **Eighth VLC partition**

1st prefix	2nd prefix	VL code	Run	Level
0000 0000	0001	0011 s	1	15
		0010 s	1	16
		0001 s	1	17
		0000 s	1	18
		0100 s	6	3
		1010 s	11	2
		1001 s	12	2
		1000 s	13	2
		0111 s	14	2
		0110 s	15	2
		0101 s	16	2
		1111 s	27	1
		1110 s	28	1
		1101 s	29	1
		1100 s	30	1
		1011 s	31	1

The decoding procedure can be exemplified on Figure 6.4. Let us suppose that the following string is to be decoded: `100000000000011000110....` The *Table_Offset* is initialized to `0`, that is the *first* table is being pointed to. Also, *Lookup_Address_Width* is initialized to `8`, which means that the first 8 bits of the string, i.e., `10000000`, will be regarded as address into the *first* table. The following values are retrieved: *code_length* = 2, *run* = 0, *level* = 1, *table_offset* = `0x100`, and *lookup_address_width* = `8`. Which means that the *second* table will be accessed during the second iteration.

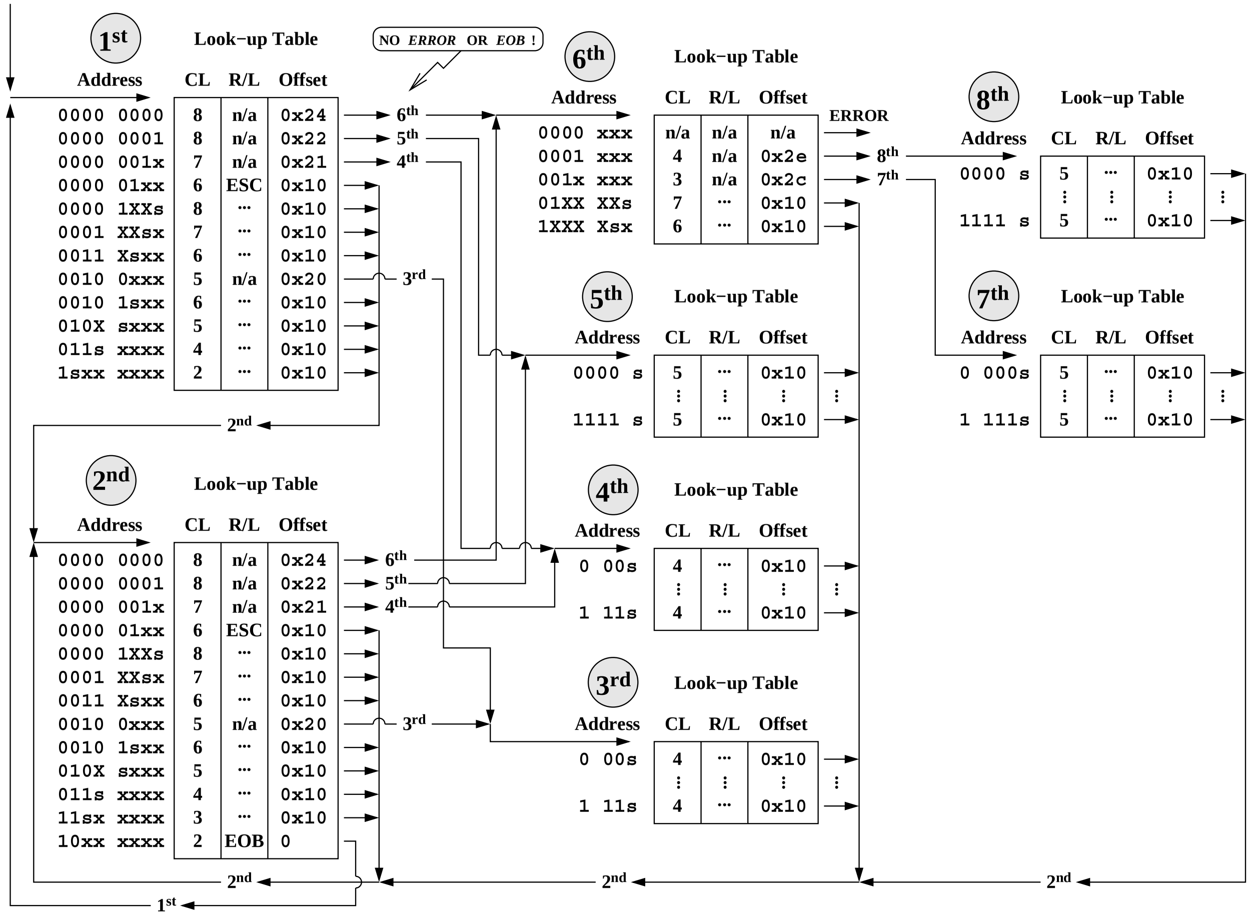
After shifting out the two bits decoded at the previous iteration, the leading eight bits, i.e., *00000000,* will be regarded as address, this time into the *second* table. By looking-up, *code_length* = `8`, *table_offset* = `0x240`, and *lookup_address_width* = `7`. That is, the *sixth* table will be accessed. No valid *run-level* pair has been detected.

At this moment the *accumulated_code_length* is 10. Therefore, the leading 10 bits have to be shifted out from the input string. Then, the next seven bits, i.e., *0011000,* are regarded as address into the *sixth* table. Again, no valid *run-level* pair is detected. The *code_length* = `3`, *table_offset* = `0x2c0`, *lookup_address_width* = `5`. That is, the *seventh* table will be accessed.

After incrementation, the *accumulated-code-length* = 13. After shifting out the leading 13 bits, the next five bits, i.e., *10001* are the address into the *seventh* table. The look-up procedure retrieves the following values: *code_length* = 5, *run* = `0`, *level* = `-32`, *lookup_address_width* = `8`, *table_offset* = `0x100` bypassing the *first* table. That is, all subsequent coefficients of the 8×8 block will use only the Tables *second - eighth*.

Finally, the accumulated-code-length is 18. The next eight bits to be sent as address to the *second* table are: *10xxxxxx.* An *end_of_block* symbol is detected, and the *table-offset = 0*; that is, the *first* table is to be accessed for decoding of a new block.

The entropy decoder implementation we propose is presented in Algorithm 7. As it can be observed, the prologue of the inner (infinite) loop (lines 17–47) is exposed to the compiler (lines 4–15). Since an *end_of_block* or *error* condition will never occur on first table lookup (line 7), testing of the *end_of_block* condition during the prologue becomes superfluous and, therefore, has been eliminated.



1st

Address	CL	R/L	Offset
0000 0000	8	n/a	0x24
0000 0001	8	n/a	0x22
0000 001x	7	n/a	0x21
0000 01xx	6	ESC	0x10
0000 1XXs	8	···	0x10
0001 XXsx	7	···	0x10
0011 Xsxx	6	···	0x10
0010 0xxx	5	n/a	0x20
0010 1sxx	6	···	0x10
010X sxxx	5	···	0x10
011s xxxx	4	···	0x10
1sxx xxxx	2	···	0x10

2nd

Address	CL	R/L	Offset
0000 0000	8	n/a	0x24
0000 0001	8	n/a	0x22
0000 001x	7	n/a	0x21
0000 01xx	6	ESC	0x10
0000 1XXs	8	···	0x10
0001 XXsx	7	···	0x10
0011 Xsxx	6	···	0x10
0010 0xxx	5	n/a	0x20
0010 1sxx	6	···	0x10
010X sxxx	5	···	0x10
011s xxxx	4	···	0x10
11sx xxxx	3	···	0x10
10xx xxxx	2	EOB	0

6th

Address	CL	R/L	Offset
0000 xxx	n/a	n/a	n/a
0001 xxx	4	n/a	0x2e
001x xxx	3	n/a	0x2c
01XX XXs	7	···	0x10
1XXX Xsx	6	···	0x10

5th

Address	CL	R/L	Offset
0000 s	5	···	0x10
	⋮	⋮	⋮
1111 s	5	···	0x10

4th

Address	CL	R/L	Offset
0 00s	4	···	0x10
	⋮	⋮	⋮
1 11s	4	···	0x10

3rd

Address	CL	R/L	Offset
0 00s	4	···	0x10
	⋮	⋮	⋮
1 11s	4	···	0x10

8th

Address	CL	R/L	Offset
0000 s	5	···	0x10
	⋮	⋮	⋮
1111 s	5	···	0x10

7th

Address	CL	R/L	Offset
0 000s	5	···	0x10
	⋮	⋮	⋮
1 111s	5	···	0x10

Figure 6.4: **The flowchart of the variable-length decoding procedure.**

Special considerations have to be provided with respect to modulo-64 operation. As me already mentioned, since the extended `bitfunshift` TriMedia-specific operation is used, more flexibility in postponing the modulo-64 operation is gained. Indeed, there is no such operation within the prologue. However, from the MPEG syntax point of view this is not entirely correct. Assuming that *acc_code_length* is 63 at line 36, it will become 81 at line 45. Considering that an *end_of_block* is encountered, then *acc_code_length* = 83. If this situation occurs during the decoding of the first block in a macroblock, and if the subsequent five coded blocks in the same macroblock include each an *escape* sequence followed by an *end_of_block*, then *acc_code_length* $= 83 + 5 \times 24 + 5 \times 2 = 213$, that is more than the limit of 192 bits. Fortunately, this case is not statistically relevant (we did verify it on all MPEG2–conformance strings mentioned in the subsequent section). Fortunately, this exceptional situation can be overcomed without much penalty by augmenting the *escape* handling code within the prologue (lines 11–15) with a modulo-64 operation.

The same strategy of exposing the prologue of the loop to the compiler can be applied for decoding of intra blocks, since an *end-of-block* can never occur during the decoding of an DC coefficient. However, special precautions have to be taken in order to deal with errors.

Finally, it should be mentioned that it is difficult to use standard optimization techniques such as *loop unrolling* or *grafting* [3], because awkward escape code and/or funnel-shifting processing would be introduced.

6.3 Variable-length decoding in FPGA

Due to data dependencies, both VLD and RLD are sequential tasks. Consequently, entropy decoding is an intricate function on TriMedia, since a VLIW processor must benefit from instruction-level parallelism in order to be efficient. For this reason, such function is an ideal candidate to benefit from reconfigurable hardware support. Since RLD is mostly a memory-dominant task, essentially only a VLD computing unit is mapped on FPGA.

Referring to the conceptual scheme of variable-length decoding presented in Figure 6.1, we would like to mention that arbitrary-size barrel-shifting can be achieved in one cycle by means of a matrix of transfer gates [18]. Since tri-state logic is not available on an ACEX 1K FPGA, the only possibility to implement a barrel-shifter on such FPGA is by means of cascaded multiplexers selecting fixed-size shifting by $1, 2, 4, 8, \ldots$, respectively. Therefore, such approach is likely to exhibit high latency and large reconfigurable hardware utilization. For example,

Algorithm 7 Entropy decoder routine with the prologue exposed to compiler

```
for $i = 1$ to NO_OF_MACROBLOCKS do
    for $j = 1$ to NO_OF_BLOCKS_IN_MACROBLOCK do
        nz_coeff_pos_ZZ ← $0$
        barrel-shifting the VLC_string
        lookup_address ← the leading FIRST_LOOKUP_ADDRESS_WIDTH bits from VLC_string
        lookup_address ← lookup_address + (FIRST_TABLE_OFFSET $\ll$ 4)
        retrieved_vec64ub ← VLC_table[lookup_address]

        extract code_length, run, level, table_offset, lookup_address_width, escape, valid_decode
        from retrieved_vec64ub {end_of_block field is not extracted!}
        acc_code_length ← acc_code_length + code_length
        if escape flag is raised then
            run ← next 6 bits from VLC_string
            level ← next 12 bits from VLC_string
            acc_code_length ← acc_code_length + 6 + 12
        end if

        loop
            barrel-shift the VLC_string
            lookup_address ← the leading lookup_address_width bits from VLC_string
            lookup_address ← lookup_address + table_offset
            retrieved_vec64ub ← VLC_table[lookup_address]

            nz_coeff_pos_ZZ ← nz_coeff_pos_ZZ + Run
            nz_coeff_pos ← invZZ_table[nz_coeff_pos_ZZ]
            $8 \times 8$_matrix[nz_coeff_pos] ← Level
            nz_coeff_pos_ZZ ← nz_coeff_pos_ZZ + valid_decode

            extract code_length, run, level, table_offset, lookup_address_width, escape,
            valid_decode, end_of_block from retrieved_vec64ub
            acc_code_length ← acc_code_length + code_length
            if acc_code_length $\leq 64$ and not(escape) then
                continue { ——————————————-> go to loop}
            end if
            if end_of_block flag is raised then
                break { ————————————————-> initiate the next for iteration (block-level)}
            end if
            if acc_code_length $\geq 64$ then
                advance the VLC_string by 64 bits
                acc_code_length ← acc_code_length - 64
            end if
            if escape flag is raised then
                run ← next 6 bits from VLC_string
                level ← next 12 bits from VLC_string
                acc_code_length ← acc_code_length + 6 + 12
            end if
        end loop
    end for
end for
```

considering that the longest *code-length* is 24 bit, we determined that a barrel-shift of 2:1 (longest) codewords, i.e., 48:24 bits, has a latency which ranges between 10.2 ÷ 11.9 ns, depending of the mapping and routing options. Similarly, the latency of a 3:1 (longest) codewords barrel-shift (72:24 bits) ranges between 11.9 ÷ 13.2 ns, and the latency of a 4:2 codewords barrel-shift (96:48 bits) ranges between 16.2 ÷ 15.0 ns. Considering a TriMedia@200 MHz, these figures translates to 2 ÷ 3 TriMedia cycles, while the latency of the hardwired barrel-shifter operation is only 1 TriMedia cycle.

For this reason, we implement in FPGA only the forward path of the VLD, i.e., the path that computes the *run-level* pair and the *code-length*, while the feedback path, i.e., accumulating the *code-length* and barrel-shifting will be implemented in standard TriMedia by means of hardwired operations. However, such architectural decision has to be reanalyzed if full-custom hardwired VLD is addressed.

Subsequently, we first present an MPEG2–compliant VLD which can return one symbol per call. Then, we will propose a strategy to break the data dependency between successive symbols, and propose VLD-2 and VLD-3 computing resources which can return 2, respectively 3 symbols per call.

VLD-1. VLD-1 is an FPGA–based VLD which can decode one symbol per execution. Since the latency of an RFU-configured computing resource should be known at compile time, only a constant-output-rate architecture [63, 104, 103] can be considered for VLD. An ACEX 1K FPGA–based VLD-1 which is MPEG2 compliant has been presented in [96, 97]. The main idea of this design is to compute the *run-level* pair by looking-up into the EABs of an ACEX 1K FPGA, while the *code-length* and control information are computed into FPGA logic cells. Since the EABs are essentialy random-access memories with 8 inputs, while the *code-length* ranges between 2 and 17 (excluding *escape* sequences), we partitioned the Tables B14 and B15 into Groups and Classes [98], such that by bypassing a so-called *header* common to all codewords in a group, the number of remaining bits to be decoded for each and every codeword in the group (and, therefore, the resulting *address length* for looking-up into EAB) is 8 or less.

In Table 6.13 we present the partitioning of VLC Tables B14 and B15. To overcome the slightly higher difficulty of having a large number of codewords starting with a ‘1’ in Table B15, the codewords starting with ”10” or ”01” have been allocated to Group 0, while the codewords starting with ”00” or ”11” have been allocated to Group 1. This way, each class in the Group 1 can be uniquely identified by the second most-significant bit. Thus, the first of the two most-significant bits can be bypassed as being the *header* of the Group 1.

Table 6.13: **Partitioning of the Tables B14 and B15 into groups and classes.**

Table B14

Group name	No. of symbols in the class	Class prefix	Code length	Group header	Effective EAB address length
Group 0 NI-1st coeff.	2	1	1 + s	–	n.a.
End-of-block	1	10	2	–	n.a.
Group 0 AC coeff.	2	11	2 + s	–	n.a.
Escape MPEG-2 (MPEG-1)	1	0000 01	6 + 18 6 + (14,22)	–	n.a.
Group 1 (implemented into EABs)	2	011	3 + s	0	3
	4	010	4 + s		4
	4	0011	5 + s		5
	2	0010 1	5 + s		5
	8	0001	6 + s		6
	8	0000 1	7 + s		7
	16	0010 0	8 + s		8
Group 2 (implemented into EABs)	16	0000 001	10 + s	0000 00	5
	32	0000 0001	12 + s		7
	32	0000 0000 1	13 + s		8
Group 3 (implemented into EABs)	32	0000 0000 01	14 + s	0000 0000 0	6
	32	0000 0000 001	15 + s		7
	32	0000 0000 0001	16 + s		8

Table B15

Group name	No. of symbols in the class	Class prefix	Code length	Group header	Effective EAB address length
End-of-block	1	0110	4	–	n.a.
Group 0	2	10	2 + s	–	n.a.
	2	010	3 + s	–	n.a.
	2	0111	4 + s	–	n.a.
Escape MPEG-2	1	0000 01	6 + 18	–	n.a.
Group 1 (implemented into EABs)	4	0011	5 + s	0	5
	2	0010 1	5 + s		5
	8	0001	6 + s		6
	8	0000 1	7 + s		7
	16	0010 0	8 + s		8
	2	110	3 + s	1	3
	4	1110	5 + s		5
	8	1111 0	7 + s		7
	2	1111 100	7 + s		7
	8	1111 11	8 + s		8
	4	1111 101	8 + s		8
Group 2 (implemented into EABs)	2	0000 0010	9 + s	0000 00	4
	4	0000 0011 1	9 + s		4
	4	0000 0011 0	10 + s		5
	20	0000 0001	12 + s		7
	24	0000 0000 1	13 + s		8
Group 3 (implemented into EABs)	32	0000 0000 01	14 + s	0000 0000 0	6
	32	0000 0000 001	15 + s		7
	32	0000 0000 0001	16 + s		8

In connection to the Table 6.13 we would like to mention that only the large groups, i.e., 1, 2, and 3, are implemented into EABs. The small groups, i.e.,, Group 0, escape, and end-of-block, as well Tables B12 and B13 are implemented into the FPGA logic cells. With such partitioning in groups, the *run* and *level* for each and every group were decoded in parallel, as the valid symbol would belong to that group. Then, a selection of the proper *run-level* pair is carried out according to the code-length, as depicted in Figure 6.5. For completeness, we also present the detailed implementation of the VLD-1 in Figure 6.6 (only `intra_vlc_format` = 0 is considered, however).

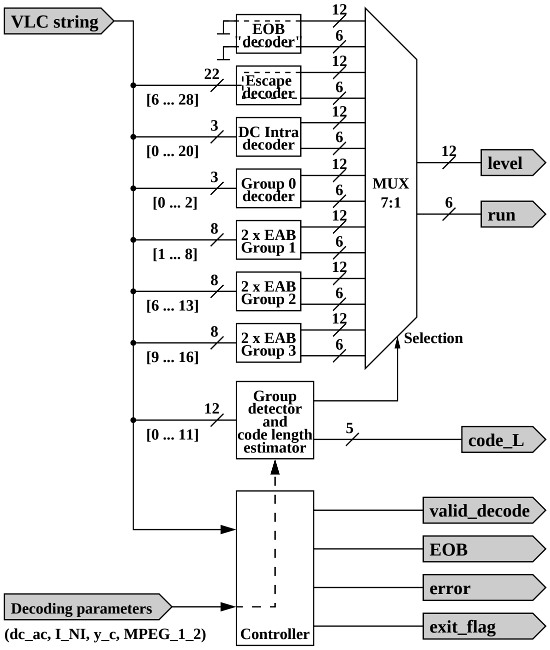


Figure 6.5: **The conceptual VLD-1 implementation on FPGA.**

Six EABs an EP1K100 device (the largest FPGA in the Altera's ACEX family) are needed to implement the Tables B14 and B15, an ACEX EP1K100 device have been used to implement the rest of the VLD-1. By simulation with Altera tools, we found that the VLD-1 latency is equal to 7 TriMedia cycles.

Since the next variable-length codeword can be decoded only after the current one has been decoded, VLD is a strictly sequential task. Subsequently, we will propose a strategy to increase the parallelism by breaking the explicit dependency between successive codewords, and will present a variable-length decoder which can decode 2 symbols per call.

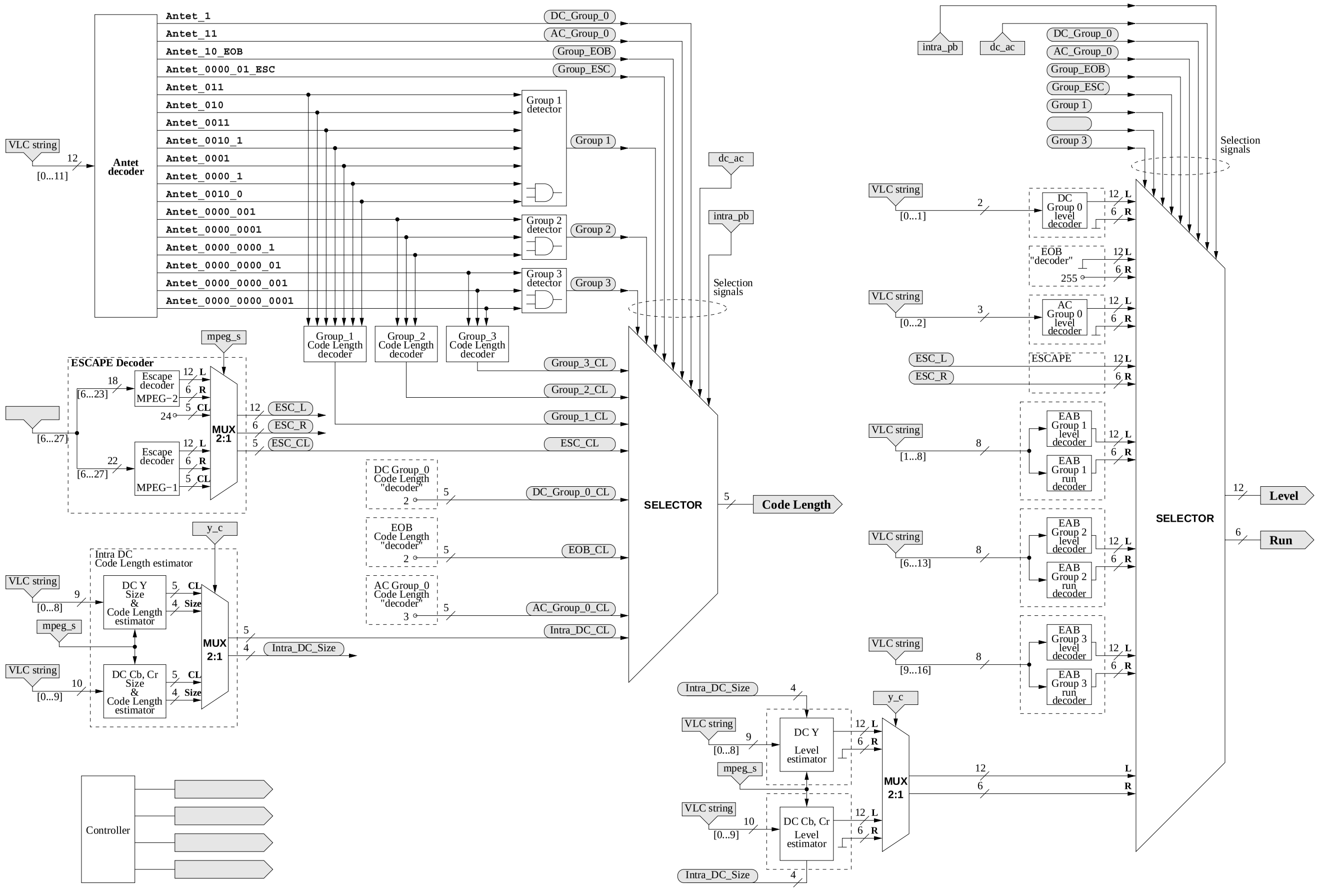


Figure 6.6: **The detailed VLD-1 implementation on FPGA.**

VLD-2. Since VLD is a strictly sequential algorithm, trully *two codewords at a time* can be achieved only if a huge look-up table of $28 + 28 = 56$ inputs (2^{56} word memory!) for MPEG-1, and $24 + 24 = 48$ inputs (2^{48} word memory!) for MPEG-2 is employed. Even by excluding the ESCAPE codes, a look-up memory with $17 + 17 = 34$ inputs, which means $2^{34} \approx 10$ Gwords is still needed! Several architectures have been proposed to overcome the need for such a large memory.

- The decoder proposed by Park [76] finds the *run*, *level*, and *code-length* of the first symbol, then barrel-shifts the VLC string according to the *code-length*, and finally finds the *run*, *level*, and *code-length* of the subsequent symbol. Since it is difficult to implemented a barrel-shifter on FPGA, the Park's architecture is not appropriate for our case.
- The decoders described in [66], [61] employ *advance computation* techniques. For the first bit of the VLC string, the symbol is fully decoded, i.e., *run*, *level*, and *code-length* are determined. In parallel, for all possible starting bit positions for the next symbol, *run*s, *level*s, *code-length*s are also generated. Finally, only a selection based on the *code-length* of the first decoded symbol is carried out. The major drawback of this approach is the huge complexity of the decoder, as decoding look-up tables have to be provided for the first symbol, and also for all possible second symbols.

In order to overcome the drawbacks of the above mentioned architectures, we propose to decode *run*, *level*, and *code-length* of a first symbol, and to determine only the *code-length* for the second symbol by means of advance computation techniques. The computation of the *run-level* pair of the second symbol is postponed to the next VLD call. In parallel, the *run-level* pair of the *previous* codeword is determined. In this way, with the exception of a firing-up call which decodes only a single symbol, trully two-symbol decoding is achieved for all subsequent VLD calls. The complexity of the VLD-2 remains resonable low since only a small number of look-up tables have to be provided. As mentioned, barrel-shifting will be carried out in software, in the standard TriMedia–CPU64 instruction set.

Several aspects regarding the terminology have to be discussed. The codeword corresponding to the first bit of the VLC string is referred to as *current* codeword. The second symbol is referred to as *next* codeword, and a codeword whose *code-length* was determined during the previous VLD-2 call is referred to as *previous* codeword. The acronyms related to the *current* codeword get the suffix `_c`, those related to *next* codeword get the suffix `_n`, and those related to the *previous* codeword get the suffix `_p`. Therefore, the acronyms `run_c`, `level_c`, `code_L_c`, `code_L_n`, `run_p`, `level_p`, etc, are valid.

The conceptual FPGA–based implementation of the VLD-2 core is presented in Figure 6.7. As it can be observed, decoding the *code-length* of the *next* codeword is carried out in parallel with decoding of the *run-level* pair of the *current* codeword (and also of the *previous* codeword). Therefore, the critical path of the VLD-2 is approximately the same with the critical path of the VLD-1. For VLD-2, the same methodology used in VLD_1 has been employed, i.e., *run* and *level* for all the Groups in the decoding tables are decoded in parallel, then only a selection of the proper result is carried out. In order to easily quit the entropy decoder calling routine once an *end-of-block* or an error has been detected for either of the *current* or *previous* codeword, a *global_exit* flag is provided.

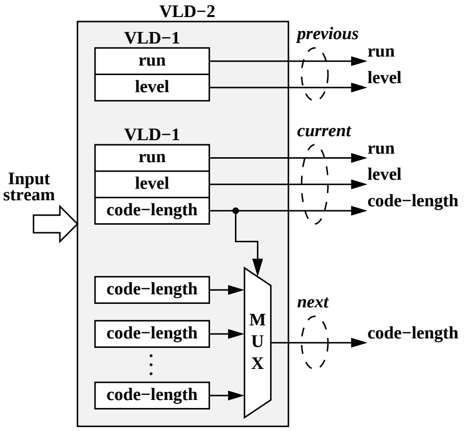


Figure 6.7: **The FPGA–based VLD-2.**

All 12 EABs and 51% of the logic cells of an ACEX EP1K100 device have been used to implement an MPEG-2–compliant VLD-2. By writing VHDL code, followed by compiling, mapping, and simulation with Altera development tools, we found that the VLD-2 latency ranges between 7-8 TriMedia cycles, depending on the computing of additional values which may prove useful at the entropy decoding routine level, e.g., `nz_coeff_pos`, `end_of_macro_block`, etc, as we will discuss in the next section.

VLD-3 The VLD-2 principle is scalable and can be extended to VLD-x ($x \geq 3$), subject to the FPGA size. In a VLD-3, two *next*/*previous* codewords are considered. Unfortunately, VLD-x ($x \geq 3$) seems not to be feasible. In VLD-2, the selection of the proper `code_L_n` can be completed in about the same time with *run-level* decoding for the *current* (and also *previous*) codeword(s). In VLD-3, for example, the computation of the *code-length*s for the *next* two codewords is on the critical path. Therefore, longer latencies are to be expected for VLD-x ($x \geq 3$), while VLD-2 has about the same latency with VLD-1. By simulation with Altera tools, we found that decoding the *code-length*s of the *current* and two *next* codewords can take up to 50 ns on an ACEX 1K FPGA, which is 10 TriMedia@200 MHz cycles. Considering the extra *read* and *write-back* cycles, the latency of the VLD-3 is 12 TriMedia cycles.

Another difficulty in mapping a VLD-3 is the need for two EP1K100 FPGAs. Moreover, there are also limitations connected with TriMedia–CPU64 super-operation format, which strongly discourages using VLD-x ($x \geq 3$), as we will describe later this chapter.

In entropy decoding on FPGA-augmented TriMedia, the VLD benefits from reconfigurable hardware support, while the processor still carries out the inverse zig-zag and matrix reconstruction, i.e., the RLD. In Section 6.4, we will describe three entropy decoders on FPGA-augmented TriMedia.

6.4 Entropy decoder implementation on ρ–TriMedia

Let us assume that the incoming bit-string is resident into main memory or Memory-Mapped Input/Output (MMIO) registers [2]. Since the TriMedia–CPU64 datapath is 64 bit wide, the bit-string will be downloaded into the VLIW core in chunks of 64 bits. Classical optimization techniques [77, 3] have been used for bit parsing operations. In particular, to avoid memory references, local copies of the input and output global variables are used. Since the compiler allocates the local copies to registers, the leading 64-bit chuncks are stored into register file.

As described in Section 6.1, the *accumulated code-length*, `acc_code_L`, stores the sum of the code-lengths of the previously decoded codewords. Before each VLD call, the input stream has to be shifted by `acc_code_L` positions in order to discard the decoded bits. As we established in Section 6.3, the shifting procedure will be performed in standard TriMedia, for which a dedicated *bitwise funnel-shift* double-slot operation is provided:

[if Rguard]
bitfunshift Rsrc_LDW, Rsrc_RDW, Rshift → Rdest_LDW, Rdest_RDW

where `Rsrc_LDW`, `Rsrc_RDW`, `Rdest_LDW`, `Rdest_RDW` are 64-bit unsigned integers defining the Left and Right Double Words (LDW, RDW) of the input, respectively the output. `Rshift` is considered to be a 32-bit unsigned integers defining the shifting value, and `Rguard` is the guarding register. The (original) funnel-shift operation on standard TriMedia is presented as shaded area in Figure 6.8.

The bit parsing should be done with caution, since enough bits for variable-length decoding have to be available after barrel-shifting. A strategy which guarantees that at least $1 + 64 = 65$ bits are available for the subsequent VLD calls is as follows [82]. The `acc_code_L` is incremented *modulo-64* with the *code-length* of each newly decoded codeword. On overflow, a new chunk of 64 bits from the

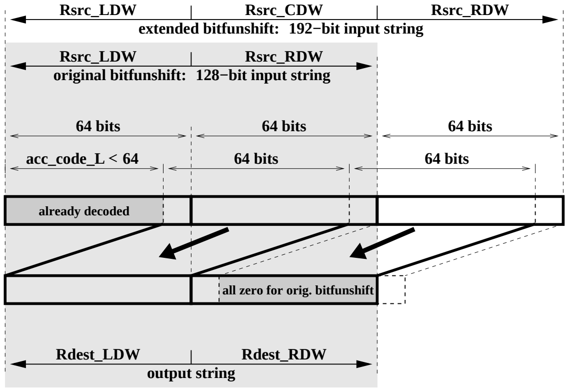


Figure 6.8: **The 'original bitfunshift' and 'extended bitfunshift' operations.**

MPEG string is downloaded from main memory into TriMedia core, as depicted in Figure 6.9. This way, the actual input bitstream is accessed only when the number of bits required is greater than the number of bits left in the register file.

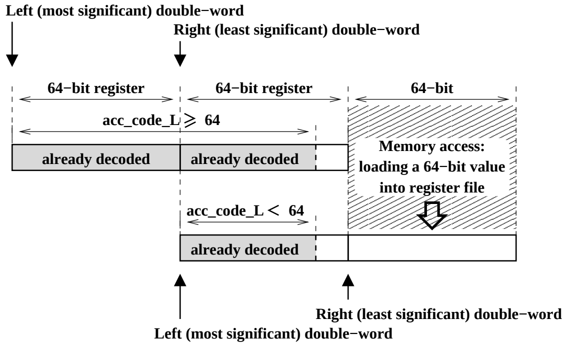


Figure 6.9: **Updating the 'register' copy of the leading (most significant) 64-bit chunks of the VLC bit-string.**

We would like to mention that the incrementation *modulo-64* is emulated by plain incrementation followed by a conditional jump which bypasses a subtraction with 64 if `acc_code_L` < 64. Since the longest (MPEG-1) *code-length* is 28 bit, which means that two codewords can be at most 56 bit long, those 65 bits guaranteed by the strategy presented in Figure 6.9 allows for a slight optimization: a plain incrementation can be used instead of an incrementation *modulo-64* any other single symbol variable-length decoding, provided that `acc_code_L` < 64 at the beginning of the algorithm. In this way, one of two conditional jumps is eliminated, with significant benefits in TriMedia-based entropy decoder implementations.

Since the double-slot operation format accepts four input registers, further improvements are still possible. We propose to extend the original funnel-shifting operation such that the barrel-shifting is done over three 64-bit chunks instead of two. The *extended funnel-shift* operations is depicted in the same Figure 6.9. The sintax of the new operation is:

[if Rguard]
bitfunshift_3 Rsrc_LDW, Rsrc_CDW, Rsrc_RDW, Rshift →
Rdest_LDW, Rdest_RDW

where `Rsrc_CDW` is the Central Double Word of the input string, all the other registers having the same meaning as in the original operation. After `acc_code_L` is incremented *modulo-64*, the extended funnel-shift operation guarantees that at least $1 + 128 = 129$ bits are available for subsequent VLD calls, provided that `acc_code_L` < 64 at the beginning of the algorithm. Consequently, at least five MPEG-2 codewords can be decoded before the `acc_code_L` is updated *modulo-64*. This approach may have a lot of benefits in optimizing the entropy decoder routine, especially when multiple-symbol VLDs are employed. We will come back to this issue later this chapter.

Finally, we would like to note that the extended funnel-shift operation having $64 \leq$ `acc_code_L` < 128 is identic with standard funnel-shift operation having `acc_code_L` ≥ 0. Also, the Figure 6.9 showing the updating process of the VLC string should be augmented in order to include three 64-bit words: `Rsrc_LDW`, `Rsrc_CDW`, `Rsrc_RDW`. However, this is not presented here.

Subsequently, we will focus on three entropy decoder classes: VLD-1–based, VLD-2–based and VLD-3–based. We will describe their implementation concepts and will present the optimizations thatv are likely to be the most effective in terms of instruction cycles.

VLD-1–based entropy decoder. The main idea of an FPGA-based solution is to replace the awkward table look-ups and *escape* handling code used in the pure software decoder [82, 99], by a single RFU call. In this paragraph, we assume that a VLD-1 computing resource which can decode one symbol per call is configured on FPGA. Thus, the entropy decoder routine includes calls to VLD-1. We would like to remind that a VLD-1 call is actually performed by launching an `EXECUTE` super-operation having a pointer to VLD-1 as $\mathtt{RFU} - \mathtt{OP} - \mathtt{ID}$, as described in Chapter 3. The following stages can be discerned in the implementation of the VLD-1–based entropy decoder:

1. **Initializations** (SET VLD-1 computing resource, Initialize VLD-1, update `acc_code_L`, etc.).
2. **VLD-1 call** ($\equiv$ EXECUTE call)
3. **Field extraction**: `run`, `level`, `code_L`, `exit_flag`.
4. **Updating the accumulated code-length**:

   `acc_code_L` $+=$ `code_L` (modulo 64)
5. **Updating the *register* copy of the leading (most significant) 64-bit chunks of the VLC string**.
6. **Exit** if an exit condition (*end-of-block*, error, etc.) has been encountered.
7. **Run-length decoding** (calculating `nz_coeff_pos` followed up by filling-in the non-zero coefficient into the 8×8 matrix) **and additional computations associated with MPEG** (de-zig-zag, inverse quantization, etc.).
8. **Barrel-shifting the VLC string** in order to discard the already decoded bits.
9. **Looping: GOTO step 2**.

Stages 7 (**run-length decoding**) and 8 (**barrel-shifting**) can be folded into the loop, such that loop pipelining is employed. In this way, the run-length decoding for the *previous* VL decoded symbol is carried out simultaneously with variable-length decoding of the *current* symbol. The penalty of such approach is that only variable-length decoding for the first DCT coefficient will be performed during the first iteration, since the software pipeline has to be fired-up. To reduce this overhead, we propose to expose the *prologue* of the loop [57] to the compiler. The dataflow of the entropy decoder routine with the prologue exposed to the compiler is presented in Figure 6.10–a.

In connection to the bit parsing operation, we would like to comment that, in order to increase the instruction level parallelism, software pipelining techniques are also employed. That is, *barrel-shift* operation is scheduled *before* both the *modulus* operation and VLC-string updating. As we established at the beginning of this section, this strategy will *never* rise problems even when *standard funnel-shift* is used, since even in the worst case there are enough bits left to decode an additional. Moreover, if *extended funnel-shift* is used, the conditional jump associated to *modulus* operation is no longer needed for variable-length decoding of any other single symbol. Therefore, the code can be replicated following a standard *grafting* optimization technique [3], as depicted in Figure 6.10–b. Based on the same observation, only a plain incrementation instead of a modulo-64 one during the *prologue* suffices.

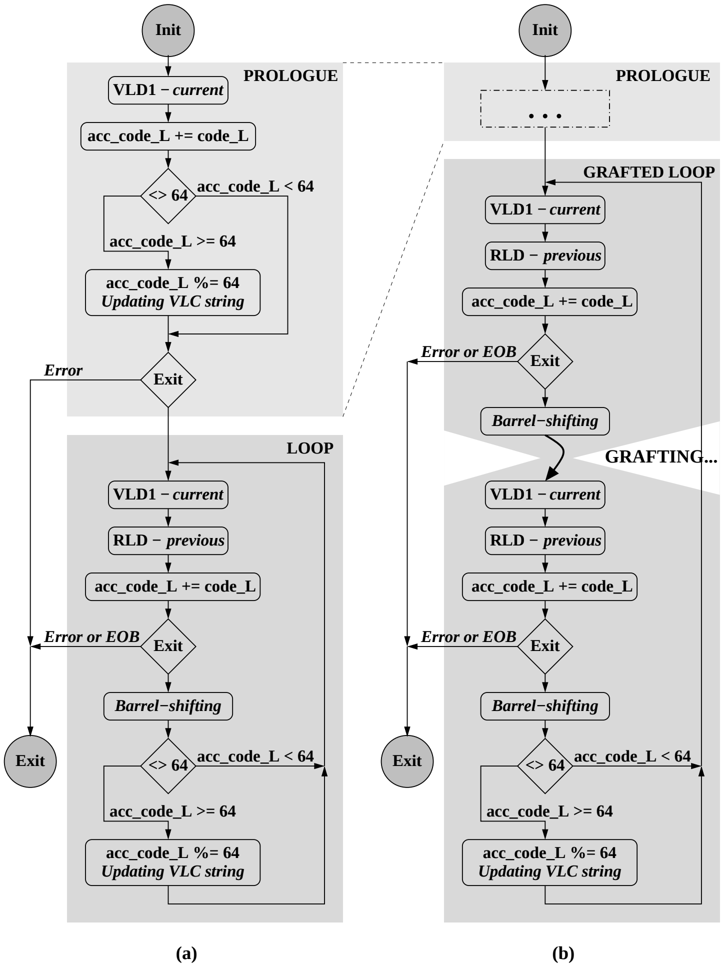


Figure 6.10: **The dataflow of the VLD-1–based entropy decoder: (a) – without grafting; (b) – with grafting.**

At the end of this paragraph, we would like to comment a little about the *Exit* test during the *prologue*. Since an *end-of-block* can never be encountered during the decoding of a DC/1st DCT coefficient, the *Exit* branch in *prologue* is taken only if an *error* is encountered. Since for the time being is hard to figure out whether dealing with error concealment may or may not be subject for further optimizations at a complete MPEG decoder level, the best we can do is to be conservative and consider that an *error* is processed immediately after it is encountered. Therefore, the penalty of the *prologue Exit* branch is taken into consideration in the subsequent experiments.

VLD-2–based entropy decoder. The VLD-2–based entropy decoder is a direct extension of the VLD-1–based one. The routine includes calls to VLD-2. The following stages can be discerned in the implementation of the VLD-2–based entropy decoder:

1. **Initializations** (SET VLD-2 computing resource, Initialize VLD-2, update `acc_code_L`, etc.).
2. **VLD-2 call** ($\equiv$ EXECUTE call)
3. **Field extraction**: `run_p`, `level_p`, `code_L_n`, `run_c`, `level_c`, `code_L_c`, `exit_flag`.
4. **Updating the accumulated code-length**:
   `acc_code_L` $+=$ `code_L_c` (modulo 64)
5. **Barrel-shifting the VLC string** in order to compute *VLC string - previous.*
6. **Updating the accumulated code-length**:
   `acc_code_L` $+=$ `code_L_n` (modulo 64)
7. **Updating the *register* copy of the leading (most significant) 64-bit chunks of the VLC string**.
8. **Exit** if an exit condition (*end-of-block*, error, etc.) has been encountered.
9. **Run-length decoding** for *previous* symbol (updating `nz_coeff_pos_p` followed up by filling-in the non-zero coefficient into the 8×8 matrix) **and additional computations associated with MPEG** (de-zig-zag, inverse quantization, etc.).
10. **Run-length decoding** for *current* symbol (updating `nz_coeff_pos_c` followed up by filling-in the non-zero coefficient into the 8×8 matrix) **and additional computations associated with MPEG** (de-zig-zag, inverse quantization, etc.).
11. **Barrel-shifting the VLC string** in order to compute *VLC string - current, next.*
12. **Looping: GOTO step 2**.

Following the same strategy used for VLD-1–based entropy decoder, either the Stage 10 or both Stages 10 and 9 can be folded into the loop. As it can be easily observed, the complexity of the entropy decoding loop is significantly higher than that of its VLD-1–based counterpart. For this reason, the overhead associated with firing-up the loop may become significant and even cancel the efficiency provided by VLD-2. The same technique of exposing the *prologue* can be applied. Since the VLD-2–based entropy decoder routine is actually *double-pipelined* (once at the VLD level and once at the entropy decoder itself level), two entropy decoder iterations (the *prologue* and the 1$^{\text{st}}$ one) are needed to completely fire-up the pipeline. This is shown in Figure 6.11.

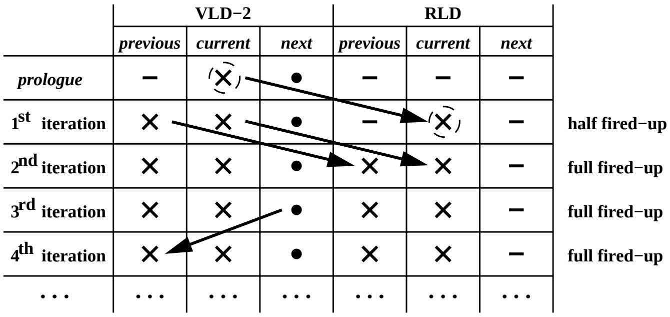


Figure 6.11: **The firing-up process of the entropy decoder software pipeline.**

In VLD-2–based entropy decoder, *grafting* should be used with caution, since not even the *extended funnel-shift* operation can provide enough bits to be decoded in the *worst* case. Fortunately, this *worst* case is extremely very unlikely to be encountered. By running the grafted VLD-2–based entropy decoder routine over MPEG-conformance scenes, we determined that the maximum value of `acc_code_L` which has been encountered before the adjustment modulo-64 is 137 from a total of 192, provided that *extended funnel-shift* operation is used. Thus, additional precautions regarding variable-length decoding are worthless.

VLD-3–based entropy decoder. VLD-2–based entropy decoder can be extended to an VLD-x–based ($x \geq 3$) entropy decoder. Unfortunately, three major issues limit the utilization of a VLD-x ($x \geq 3$) computing resource:

- The overhead associated with firing-up the loop becomes larger and larger, which turns into a highly inefficient VLD-x–based ($x \geq 3$) entropy decoder.
- The *grafting* technique is no longer as efficient as in the VLD-2–based entropy decoder. Since even the *extended funnel-shift* operation is not able to provide enough bits for variable-length decoding within an entropy decoding loop iteration, the conditional jump associated with the adjustment *modulo-64* cannot be eliminated at the *grafting* point any more.
- There are also limitations related with TriMedia super-operation format.

We will come back to these issues and present more details as well as experimental figures regarding VLD-3–based entropy decoder in Section 6.5.

With VLD-1, VLD-2, and VLD-3, different tests have been carried out. We will present them subsequently along with experimental results.

6.5 Experimental results

For all experiments described subsequently, the MPEG-compliant bit string is assumed to be entirely resident into the main memory. In this way, side effects associated with bit string acquisition such as asynchronous interrupts, trashing routines, or other operating system related tasks, do not have to be counted. Moreover, saving the reconstructed 8×8 matrices into memory, as well as zeroing these matrices in order to initialize a new entropy decoding task are equally not considered. Since both procedures can be considered parts of adjacent tasks, such as IDCT or motion compensation, they are subject to further optimizations at the complete MPEG decoder level. Thus, in our experiments, the run-length decoder will overwrite the same 8×8 matrices again and again. With these assumptions, the only relevant metric is the number of instruction cycles required to perform strictly entropy decoding. Therefore, the main goal was to minimize this number.

Two experiment classes have been considered: End-of-Block (EOB) and End-of-Macro-Block (EOMB). In the first (EOB) class, VLD-1, VLD-2 and VLD-3 generate an *exit* condition if an *end-of-block* has been encountered. This way, the entropy decoding loop is left after the 8×8 block has been fully reconstructed. As mentioned in Section 6.3, only a single codeword is decoded during the first VLD-x ($x \geq 2$) call. Since the average number of codewords per block is quite small, ranging between 4.3 and 6.8 for non-intra macroblocks (Table 6.14), the inefficiency of the first VLD-x ($x \geq 2$) is quite large. For example, 3 VLD-2 calls instead of the ideal 2.5 are needed to decode 5 coefficients (20% "overhead"), while 4 VLD-2 calls instead of the ideal 3 are needed to decode 6 coefficients (25% "overhead").

In the second (EOMB) experiment class, VLD-2 and VLD-3 generate an *exit* condition if an *end-of-macro-block* has been encountered, and the entropy decoding loop is left only after the entire macroblock has been reconstructed. There are at least 13.5 codewords per block for non-intra macroblocks, as evidenced in Table 6.14. Considering VLD-2, 8 instead of the ideal 7.5 VLD-2 calls are needed to decode a macroblock with 15 coefficients. Thus, the first iteration "overhead" is much lower, decreasing to only 7%. We have to note that there is no *end-of-macro-block* symbol. An *end-of-macro-block* condition is raised when the *end-of-block* is encountered during the decoding of the last block in the current macroblock.

For both experiment classes, the performance of the entropy decoders have been evaluated according to two scenarios. In the **first scenario**, the VLD unit returns the `run` value as defined by the MPEG standard, while the position of the non-zero coefficient, `nz_coeff_pos`, (see Section 6.1) is carried out in software.

Table 6.14: **Intrinsic MPEG-2 stream statistics – excerpt from Appendix A**.

	Intra (B14 + B15)				
Scene	coefficients	blocks	macroblocks	coeff. / block	coeff. / macroblock
batman	172,745	23,298	3,883	7.4	44.5
popplen	47,003	4,572	762	10.3	61.7
sarnoff2	80,563	8,418	1,403	9.6	57.4
tennis	133,099	12,222	2,037	10.9	65.3
ti1cheer	80,818	8,244	1,374	9.8	58.8

	Non-Intra				
Scene	coefficients	blocks	macroblocks	coeff. / block	coeff. / macroblock
batman	266,485	38,940	7,437	6.8	35.8
popplen	28,069	4,139	1,206	6.8	23.3
sarnoff2	36,408	8,447	2,673	4.3	13.6
tennis	137,756	25,524	8,454	5.4	16.3
ti1cheer	51,680	9,432	2,717	5.5	19.0

In addition, based on the *coded_block_pattern* value defined by MPEG standards [51, 53], the index of the block being decoded within the current macroblock, `block_index`, is also returned by the VLD unit. In the **second scenario**, the `nz_coeff_pos` within a macroblock is computed inside FPGA and returned by the VLD call. In this later case, the `block_index` information is redundant.

In connection to the FPGA–based computation of the `nz_coeff_pos`, several comments are worth to be provided. As depicted in Figure 6.12, the format of the *nz_coeff_pos*-relative-to-a-macroblock integer consists of two fields: (1) the *table_index*, and (2) *nz_coeff_pos*-relative-to-a-block. When accumulating the *run* values in order to generate the *nz_coeff_pos*-relative-to-a-block (as in Figure 6.2), a carry over the sixth position will never occur. This assertion is true for there are 64 elements in a 8×8 block. In addition, since the *table_index* is uniquely determined from *coded_block_pattern*, it can be computed in parallel with *nz_coeff_pos*. Without going into further details, we would like to mention that computing the *nz_coeff_pos* relative to a macroblock does not significantly increase the length of the VLD critical path. The latency of the VLD-2 unit in the second scenario remains 8 TriMedia@200 MHz cycles.

No. of bits	**3**	**6**
Field name	**table_index**	**nz_coeff_pos relative to a block**

Figure 6.12: **The format of the *nz_coeff_pos* relative to a macroblock.**

The VLD-x instructions are presented subsequently. For all VLD-1, VLD-2, VLD-3 computing units, an initialization stage is managed by the instruction:

EXECUTE_2 <INIT> Rinit → (void)

where the Register Rinit contains the initial values that are required by the decoding procedure at (macro)block level: *coded_block_pattern*, *intra_non-intra*, *luminance_chrominance*, and *intra_vlc_format*. Regarding the proper variable-length decoding task, the syntax of the VLD-1 and VLD-2 instructions are quite similar:

EXECUTE_2 <VLD_1> Ry → Rz, Rw

EXECUTE_2 <VLD_2> Ry, Ryy → Rz, Rw

The registers Ry and Ryy contain the incoming coded string which has been aligned to start with the *current*, respectively *previous* codeword. Their common format is disclosed in Table 6.15. The *run* (or alternatively, *nz_coeff_pos*), and the *level* for both *current* and *previous* codeword are each represented on a 16-bit signed integer, and stored together as a four 16-bit signed integer vector in the Rz register, as presented in Table 6.17. Even though *run* (or *nz_coeff_pos*) is always a positive number that can be represented on 6 bits (10 bits for *nz_coeff_pos*), our solution is more effective since it avoids type casting for splitting the Rz vector into its components. Indeed, a single TriMedia–CPU64 cycle is needed to extract an element from a vector. The register Rw is an eight 8-bit unsigned integer vector and contains the *code-length*s of the *current* and *next* codewords, *block index*-es associated to *current* and *previous* codewords, as well as control information for the *previous* and *current* codewords (Table 6.18). We decided to provide for redundant control information such as *error*, *valid_decode*, and *EOB* flags, in order to help the entropy decoder's calling routine to deal with error concealment [53, 70, 44]. A `global_exit` flag which is set up when any exit condition is encountered is also provided.

Table 6.15: **VLD-2 – The format of the first argument register – Ry (`uint64`).**

Field name	Acronym	Width (bit)	Position (bit)	Type (TriMedia)	Range	Description
VLC string	–	64	63 . . . 0	`uint64`	n.a.	The VLC string for the *current* and *next* codewords

Table 6.16: **VLD-2 – The format of the second argument register – Ryy (`uint64`).**

Field name	Acronym	Width (bit)	Position (bit)	Type (TriMedia)	Range	Description
VLC string	–	64	63 . . . 0	`uint64`	n.a.	The VLC string for the *previous* codeword

Table 6.17: **VLD-2 – The format of the rezult register Rz (`vec64sh`).**

Field name	Acronym	Width (bit)	Position (bits)	Type (TriMedia)	Range	Description
level of the previous codeword	`level_p`	16	63 . . . 48	`int16`	...	
run or *nz_coeff_pos* of the previous codeword	`run_p` `nz_coeff_pos_p`	16	47 . . . 32	`int16`	0 . . . 63 0 . . . 383	Negative values, e.g., -65,536, can be used for signaling special conditions
level of the current codeword	`level_c`	16	31 . . . 16	`int16`		
run or *nz_coeff_pos* of the current codeword	`run_c` `nz_coeff_pos_c`	16	47 . . . 32	`int16`	0 . . . 63 0 . . . 383	Negative values, e.g., -65,536, can be used for signaling special conditions

Table 6.18: **VLD-2 – The format of the rezult register Rw (`vec64ub`).**

Field name	Acronym	Width (bit)	Position (bit)	Type (TriMedia)	Range	Description
code_length of the current codeword	`code_L_c`	8	63 . . . 56	`uint8`	n.a.	High values (255, 254, ...) could be used for signaling special conditions
code_length of the next codeword	`code_L_n`	8	55 . . . 48	`uint8`	n.a.	High values (255, 254, ...) could be used for signaling special conditions
table_index of the previous codeword	`table_index_p`	8	47 . . . 40	`uint8`	n.a.	High values (255, 254, ...) could be used for signaling special conditions
table_index of the current codeword	`table_index_c`	8	39 . . . 32	`uint8`	n.a.	High values (255, 254, ...) could be used for signaling special conditions
Not used	–	8	31 . . . 24	`uint8`	n.a.	
`Exit` controls c	–	8	23 . . . 16	`uint8`		
valid_decode	`valid_decode_c`	1	19	`bit`	$\{0, 1\}$	Valid decode for the *current* codeword
error	`error_c`	1	18	`bit`	$\{0, 1\}$	Error for the *current* codeword
EOB	`EOB_c`	1	17	`bit`	$\{0, 1\}$	End-of-block for the *current* codeword
exit_flag	`exit_flag_c`	1	16	`bit`	$\{0, 1\}$	Exit flag for the *current* codeword
`Exit` controls p	–	8	15 . . . 8	`uint8`		
valid_decode	`valid_decode_p`	1	11	`bit`	$\{0, 1\}$	Valid decode for the *previous* codeword
error	`error_p`	1	10	`bit`	$\{0, 1\}$	Error for the *previous* codeword
EOB	`EOB_p`	1	9	`bit`	$\{0, 1\}$	End-of-block for the *previous* codeword
exit_flag	`exit_flag_p1`	1	8	`bit`	$\{0, 1\}$	Exit flag for the *previous* codeword
`Global Exit` flag	–	8	7 . . . 0	`uint8`	$\{0, 1\}$	Signals exit condition. Its least-significant position allows it to be used as guarding
global_exit_flag	`global_exit`	1	1	`bit`	$\{0, 1\}$	

The same strategy to pack *run*/*nz_coeff_pos* and *level* values into a four 16-bit signed integer vector is no longer possible in the VLD-3. Since there are too many values which have to be returned by the VLD-3 call (three *run*s/*nz_coeff_pos*, three *level*s, three *code-length*s, three *block_index*es for the EOMB class, as well as the control information), the only possible solution to pack them in a 64-bit word is to cross the boundaries between bytes. Thus, field extraction will be performed by a sequence of mask, shift and OR operations. Consequently, at least three TriMedia–CPU64 cycles instead of a single one will be needed.

As mentioned, the reference for evaluating the performance of FPGA–based VLDs is a pure software entropy decoder [99]. In order to decrease the number of instruction cycles needed for entropy decoding, quite large look-up tables are used for this reference implementation. For example, a 2^{11}-entry table is used for decoding an Intra DC coefficient; that is, an Intra DC coefficient is decoded in a single memory access. However, the performance evaluation of the pure software entropy decoder has been done assuming that, despide of the large look-up tables which are stored into memory, the standard TriMedia–CPU64 will never cope with a cache miss. In other words, we compare the FPGA-augmented TriMedia with an "ideal-cache" standard TriMedia–CPU64. It is worth mentioning here that only an *end-of-block* exit condition can be generated by looking-up into memory. Thus, this reference implementation will generate figures only for the EOB class.

By running our pure software entropy decoder on a TriMedia–CPU64 cycle accurate simulator over a set of MPEG-conformance bit-streams, we determined that 16.9 cycles/coefficient instead of 21.3 cycles/coefficient claimed by Pol [82] are needed to decode a symbol. Moreover, 4 out of 5 issue slots are filled in with operations (by comparison, we mention that 2.9 out of 5 issue slots are filled in with operations in Pol's implementation). This result that updates our initial assumption about the efficiency of the pure software implementation [96] is indeed a challenging reference for TriMedia–CPU64+FPGA hybrid.

The testing database for our entropy decoder consists of a number of pre-processed MPEG–conformance strings, or *scenes*, from which all the data not representing DCT coefficients have been removed. Therefore, such strings include only *run-level* and *end-of-block* symbols. All pure software, VLD-1–, VLD-2–, and VLD-3–based entropy decoders were run on the TriMedia–CPU64 cycle accurate simulator over each of the modified MPEG string. The best results for each entropy decoder with respect to all experimental classes and strategies are presented in Table 6.19. The figures indicate the number of instruction cycles needed to decode the pre-processed MPEG string. The last column of the table specifies the relative improvement of the most performant FPGA–based entropy decoder, i.e., VLD-2–based entropy decoder with grafting, over its pure software counterpart.

Table 6.19: **Entropy decoding experimental results.**

Scene (*.m2v)	Block type	Workload (coef.)	**Pure software** (the reference) (cycles)	VLD-1–based w/o grafting (cycles)	VLD-2–based w/o grafting (cycles)	**VLD-2–based with grafting** (cycles)	VLD-3–based w/o grafting (cycles)	Absolute performance (cycle/coeff.)	**Improvement** versus pure software
batman	I (B15)	172,745	**2,843,376**	2,050,693	1,618,656	**1,382,217**	1,799,032	8.0	**52.2 %**
	NI	266,485	**4,592,358**	3,112,249	2,534,072	**2,171,245**	2,768,638	8.1	
popplen	I (B15)	47,003	**777,553**	546,243	435,114	**368,813**	474,281	7.8	**51.3 %**
	NI	28,069	**475,326**	379,466	275,753	**240,801**	301,552	8.6	
sarnoff2	I (B14)	80,563	**1,387,489**	946,538	748,844	**635,646**	822,253	7.9	**50.5 %**
	NI	36,408	**577,388**	485,585	375,558	**337,393**	412,659	9.3	
tennis	I (B14)	12,345	**210,011**	149,366	118,943	**103,052**	131,284	8.3	**51.1 %**
	I (B15)	120,754	**1,937,808**	1,421,498	1,109,466	**937,625**	1,248,815	7.8	
	NI	137,756	**2,527,395**	1,795,628	1,416,234	**1,247,301**	1,597,229	9.1	
ti1cheer	I (B15)	80,818	**1,311,687**	970,904	749,386	**636,661**	823,350	7.9	**49.3 %**
	NI	51,680	**836,082**	667,417	512,389	**452,267**	573,799	8.8	

The main conclusion that one can draw from Table 6.19 is that a VLD-2 computing resource leads to the most efficient entropy decoding in terms of instruction cycles, the relative improvement being greater than 50%. According to our simulations, a more complex VLD, specifically VLD-3, does not lead to further improvements, and actually it diminishes the overall efficiency of the entropy decoder. Another interesting conclusion is that grafting is a very effective optimization technique, accounting for about 8 out of the total 50 percent of the improvement. Other implementation tricks, such as excluding the accumulator from the critical path of the VLD feedback as proposed by Sun [102], provides only a small improvement (less than 0.5%); therefore, such techniques will not be considered any more.

Comparing the figures of the EOB experiment class with the figures of EOMB class is a little unfair from the EOMB point of view, for more functionality is considered in the later class. Since the entropy decoder delivers the entire macroblock on completion in the EOMB class, and only an 8×8 block in EOB class, macroblock reconstruction is not carried out in EOB experiment class. Since this extra functionality for managing macroblock reconstruction is subject to optimizations at a complete MPEG decoder level, this is the best we can do for the time being. Therefore, we proceed to a conservative evaluation, accept this unfair comparison, and claim that the FPGA-augmented TriMedia–CPU64 can perform entropy decoding 50% faster than the standard TriMedia–CPU64. Given the fact that TriMedia–CPU64 is a 5 issue-slot VLIW processor with 64-bit datapaths and a very rich multimedia instruction set, such an improvement within the target media processing domain indicates that the hybrid TriMedia–CPU64 + FPGA is a feasible approach for entropy decoding.

It is also worth mentioning that the absolute performance of the VLD-2–based entropy decoder ranges between $7.8 \div 9.1$ cycles/coefficient. This is a very good result with respect to 33 cycles/coefficient needed for variable-length decoding on a Pentium processor with MultiMedia eXtension (MMX) claimed by Ishii *et. al* [55], and 26.0 cycles/coefficient achieved on a TMS320C80 media video processor by Bonomini *et. al* [12].

In the last experiment, we assume that the VLD-2 computing unit is implemented on dedicated hardware instead of FPGA. Since the delay on the FPGA interconnection network does not exist in an hardwired circuitry, we make a conservative assumption that the latency of the hardwired VLD-2 is half of the FPGA–based VLD-2. That is, $6/2 = 3$ TriMedia@200 MHz cycles is the VLD-2 delay on dedicated hardware. By adding one cycle for each of the read-in and write-back stages mentioned in Chapter 3, the hardwired VLD-2 latency ranges between 4 and 5 TriMedia@200 MHz cycles.

Table 6.20: **Relative efficiency of a hardwired VLD-2.**

Scene	Block type	Workload	**FPGA** VLD-2–based	**Hardwired** VLD-2–based	Absolute performance	Improvement versus FPGA	**Hardwired** VLD-2–based	Absolute performance	Improvement versus FPGA
			(latency = 8)	(latency = 5)			(latency = 4)		
(*.m2v)		(coef.)	(cycles)	(cycles)	(cycle/coeff.)		(cycles)	(cycle/coeff.)	
batman	I (B15)	172,745	1,382,217	1,108,573	6.4	**19.7** %	1,019,338	5.9	**26.1 %**
	NI	266,485	2,171,245	1,743,670	6.5		1,604,958	6.0	
popplen	I (B15)	47,003	368,813	295,654	6.3	**19.7** %	271,607	5.8	**26.2 %**
	NI	28,069	240,801	193,689	6.9		178,495	6.4	
sarnoff2	I (B14)	80,563	635,646	509,551	6.3	**19.7** %	468,266	5.8	**26.0 %**
	NI	36,408	337,393	272,228	7.5		251,758	6.9	
tennis	I (B14)	12,345	103,052	82,903	6.7	**19.5** %	76,395	6.2	**25.8 %**
	I (B15)	120,754	937,625	750,873	6.2		689,374	5.7	
	NI	137,756	1,247,301	931,287	6.8		1,597,229	9.1	
ti1cheer	I (B15)	80,818	636,661	510,225	6.3	**19.7** %	468,782	5.8	**26.1 %**
	NI	51,680	452,267	364,087	7.0		336,009	6.5	

The experimental results for the hardwired VLD-2–based entropy decoder are presented in Table 6.20. As it can be observed, the improvement of the hardwired VLD-2–based solution over FPGA–based solution is rather small, being about 19.6% for a VLD-2 latency of 5, and 26.0% for a VLD-2 latency of 4 TriMedia@200 MHz cycles. We have to note that such a small improvement is not a good trade-off for loosing the flexibility provided by the reconfigurable hardware.

6.6 Conclusion

In this chapter we investigated Entropy Decoding, and proposed a reconfigurable design for this function on ρ–TriMedia. Essentially, we proposed a strategy to partially break the data dependency related to variable-length decoding, and showed that a VLD-2 operation that returns two variable-length symbols per call leads to the most efficient entropy decoder in terms of instruction cycles.

In particular, we described an implementation of the VLD-2 computing unit on the reconfigurable hardware. We showed that, with the exception of a firing-up call, trully two-symbol decoding can be achieved for the subsequent calls, while the VLD-2 complexity remains resonable low. By writing VHDL code, followed by placement and routing, we determined that the VLD-2 computing unit has a latency of 8 TriMedia@200 MHz cycles, and occupies all electronic-array blocks and 51% of the logic cells of an ACEX EP1K100 FPGA.

A significant effort has also been made to improve the pure-software entropy decoder. The result of 16.9 cycles/symbol is, to the best of our knowledge, is better than those claimed in the literature. By configuring the VLD-2 computing unit on the reconfigurable hardware, and by unrolling the software pipeline loop calling VLD-2, the performance of the reconfigurable entropy decoder ranges from 7.8 to 9.1 cycles/symbol, which translates to a 50% improvement in terms of instruction cycles. That is, a $2\times$ speed-up with respect to the pure-software solution is achieved on ρ–TriMedia.

As a final remark we would like to emphasize that our results are particularly important since both VLD and RLD are sequential tasks. Due to data dependency, entropy decoding is an intricate function on TriMedia–CPU64, since a VLIW architecture must benefit from instruction-level paralellism in order to be efficient.

In the chapter to follow we will bring together the IDCT, IQ, and Entropy Decoder reconfigurable designs, and establish the gains in performance for a pel reconstruction application.

Chapter 7

Pel Reconstruction

P**el** reconstruction represents the joined task of header parsing, entropy decoding, inverse quantization, and IDCT. In this chapter, we show that significant improvement over a pure-software solution is possible on ρ–TriMedia for this application. Essentially, we assume that each of the reconfigurable designs proposed in the previous three chapters is configured on a different FPGA context, and the contexts are activated during computation as needed. With such reconfigurable hardware support, we establish the gain in performance when performing MPEG2-compliant pel reconstruction.

The chapter is organized as follows. For background purpose, we present the most important issues related to pel reconstruction in Section 7.1. Section 7.2 describes the execution scenario of the pel reconstruction task, which is the most appropriate on a TriMedia–based computing platform. Experimental results are presented in Section 7.3. Section 7.4 completes the chapter with some conclusions and closing remarks.

7.1 Pel reconstruction theoretical background

As mentioned, we focus on MPEG decoding in this dissertation. In particular, we aim to speed-up parts of the MPEG decoding task that are computing intensive. Several details of the MPEG standard that are important in the economy of this chapter are provided subsequently.

From the syntax point of view, an MPEG video sequence is an ordered stream of bits, which is structured hierarchically on layers. Each layer provides a wrapper around the encompassed layer, which is defined within a bit-field referred to as *header*. A *video sequence* includes a series of *Groups of Pictures* (GOP's). A GOP is divided into a series of *pictures* (frames), which always begins with an Intra-coded picture (I-picture) followed by an arrangement of Forward Predictive-coded pictures (P-pictures), and Bidirectionally Predicted pictures (B-pictures). A picture is further subdivided into *slices*, which is the smallest MPEG unit that can be independently decoded. A slice is composed of a series of *macroblocks*, and a macroblock is composed of 6 or fewer *blocks* (4 for luminance and 2 for chrominance[1]) and possibly motion vectors.

From a semantics point of view, five major stages can be distinguished in the MPEG decoding process: header parsing, Entropy Decoding (which is composed of Variable-Length Decoding, inverse zig-zag, and Run-Length Decoding), Inverse Quantization, IDCT, and motion compensation. In essence, header parsing searches for particular bit patterns in the incoming MPEG string, and then extracts different fields of the MPEG strings according to these bit patterns. Thus, header parsing is a control-dominant task, and is to be executed within the standard TriMedia instruction set. On the other hand, motion compensation is basically a memory-dominant task, the required arithmetic being a simple 16-bit signed addition per pixel. Consequently, it is also likely not to be subject for acceleration by means of reconfigurable logic. Thus, all the above mentioned stages but motion compensation are considered during the subsequent experiment. The joined task of these stages is generally referred to as *Pel Reconstruction* [70], which is outlined subsequently.

The pel reconstruction process is depicted in Figure 7.1. First, the headers at video sequence layer downto macroblock layer are decoded and various symbols are extracted: *decoding parameters* (e.g., *macroblock_address_increment*, *quantizer_scale_code*, *intra_dc_precision*), and *motion values*. The motion values are used by the motion compensation process which is not considered here. However, since these values are decoded during header parsing, the overhead associated with the decoding of the motion values will be taken into consideration in the subsequent experiment. After header parsing, the MPEG string still contains *composite symbols* (*run/level* pairs and *end_of_block*), which are decoded by the Variable-Length Decoder (VLD). Then, the Run-Length Decoder (RLD) recreates the 8×8 matrices that include DCT quantized coefficients. Next, using a quantization table and a *quantizer_scale*, an inverse quantization (IQ) is performed on each DCT coefficients. Finally, after the DC prediction unit reconstructs the DC coefficient in intra-coded macroblocks, an IDCT is carried out.

---

[1] Luminance is the monochrome reprezentation of the signal, while chrominance provides the color information for the video.

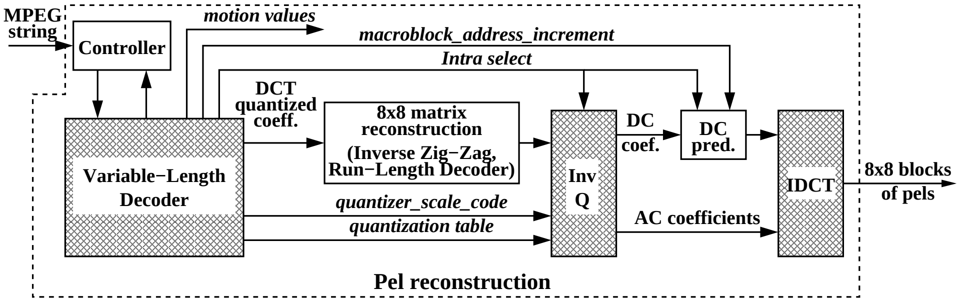


Figure 7.1: **Pel reconstruction conceptual diagram – adapted from [70].**

In connection with Figure 7.1 and the subsequent experiment, we would like to mention that the variable-length decoder, inverse quantizer, and IDCT (which are marked with gray in the figure) benefit from reconfigurable hardware support, each being configured on an FPGA context. In the next section we will analyse the pel reconstruction execution scenario.

7.2 Pel reconstruction implementation on ρ–TriMedia

In order to determine the potential impact on performance provided by the multiple-context reconfigurable core, we will consider the MPEG2 Pel Reconstruction as benchmark. As mentioned in Section 7.1, it consists of Entropy Decoding, Inverse Quantization, 1-D IDCT, and some extra tasts (header parsing, decoding of motion vectors, etc.). Our experiment includes two approaches: *pure software* and *FPGA-based*. Regarding the first approach, we would like to remind that a DCT coefficient can be decoded in 16.9 cycles [99]. Also, a pure software implementation of the 2-D IDCT can be scheduled in 56 cycles [113]. Inverse quantization takes 39 cycles per intra block and 52 cycles per non-intra block.

In the FPGA-based approach, the VLD, IQ, and IDCT functions benefit from reconfigurable hardware support. As depicted in Figure 7.2, each and every of the mentioned functions is replaced by a group of three instructions: `SET_CONTEXT`, `ACTIVATE_CONTEXT`, and `EXECUTE`. Due to the large off-chip reconfiguration penality, all the contexts of the RFU are configured at application load time, i.e., a number of `SET_CONTEXT` instructions are scheduled on the top of the program code. During the program execution, the VLD-2, IQ-4, and 1-D IDCT computing units are activated by `ACTIVATE_CONTEXT` instructions as needed. As mentioned in Chapter 3, the context switching penality is 100 cycles.

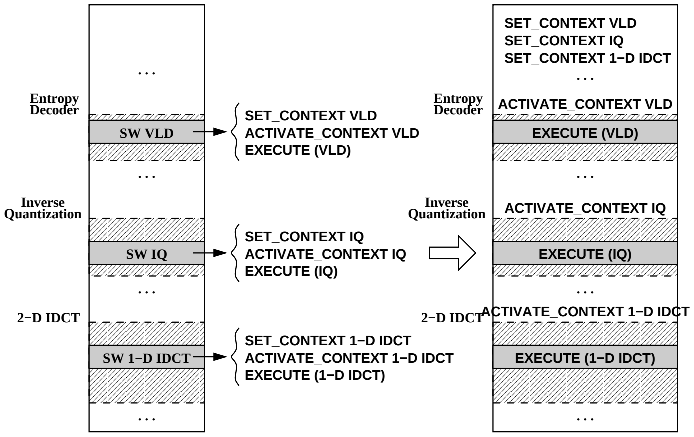


Figure 7.2: **The pieces of code that benefit from FPGA support.**

An MPEG-compliant program has been written in C, and compiled with TriMedia development tools. The performance evaluation has been done assuming that, despite of the large VLC tables that are stored into memory, the standard TriMedia will never encounter a cache miss when accessing these tables. In other words, we compare a multiple-context FPGA-augmented TriMedia with an “ideal-cache” standard TriMedia (considered from the VLC look-up perspective, since compulsory and trashing cache misses are still counted for both systems).

The testing database for our entropy decoder consists of a number of five Main Profile - Main Level (MP@ML) MPEG-2 conformance bit strings. For all experiments, the incoming string is assumed to be entirely resident into the main memory. In this way, side effects associated with string acquisition (such as asynchronous interrupts, trashing routines, or other operating system related tasks) do not have to be counted. With this assumption, the only relevant metric is the number of the instruction cycles required to perform MPEG decoding plus the overhead that will be discussed subsequently. Thus, the main goal is to minimize this number.

In the previous section, we outlined a reconfigurable Entropy Decoder that can decode a DCT coefficient in $8 \div 9$ cycles, a reconfigurable Inverse Quantizer and a 2-D IDCT, each of them having a throughput of one 8×8 block every 32 cycles. With these reconfigurable designs, different implementations of the pel reconstruction task can be envisioned. The efficiency of each implementation has to be evaluated against the overhead of firing-up and flushing the IQ and IDCT pipelines, the compulsory cache misses, and the FPGA context-switching penalty. Processing large batches of blocks translates into a low pipeline firing-up/flushing overhead and low context-switching penalty but high cache-miss penalty. A way to vary the batch size while the MPEG decoding syncronization is ensured is to carry out the decoding process at different levels: **macroblock**, **slice**, and **picture/frame**. Thus, to assess the performance for different batch sizes, the pel reconstruction task has been analysed according to three computing scenarios, as depicted in Figure 7.3:

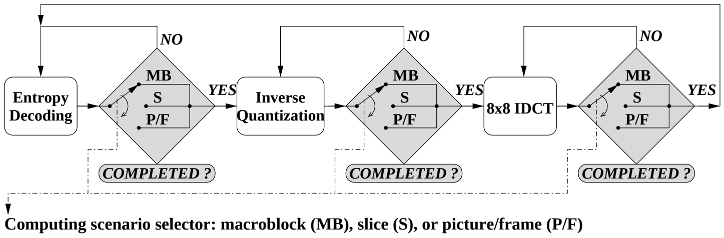


Figure 7.3: **Three possible computing scenarios of pel reconstruction.**

That is, the variable-length decoding is performed till an entire macroblock/slice/picture is fully extracted, and only then the IQ and IDCT are carried out for all blocks in the macroblock/slice/picture, respectively.

Assuming that pel reconstruction is carried out at macroblock level, the average number of blocks per macroblock is 3.1 for B-type pictures, 4.3 for P-type pictures, and 6.0 for I-type pictures (Table 7.1). For example, given the fact that the overhead to fire-up the 2-D IDCT software pipeline is 20 cycles, as we presented in Chapter 4), $84 + 2 \times 64 = 212$ cycles are needed to compute 2-D IDCT for an entire intra-coded macroblock, which translates to an average of $212/6 = 35.5$ cycles/block (11% performance degradation versus the ideal 32 cycles/block). The performance degrades even further for non-intra-coded macroblocks. When the number of blocks is odd, the last 2-D IDCT completes in 45 cycles instead of 32, giving a total of $84+45 = 129$ cycles for the average of 3 blocks in a B-coded macroblock. This translates to 43 cycles/block (34% performance degradation versus the ideal 32 cycles/block).

On the other side, processing a very large number of blocks to minimize the overhead associated to firing-up the pipelines implies that *trashing* cache misses are encountered when the blocks are read from and written back to the main memory. Even for the smallest average number of blocks per frame, which is equal to 573 in the *popplen* scene (Table 7.1), a data cache of $573 \times 64 \times 2 = 72$ KB is needed to avoid trashing cache misses after entropy decoding. Such a data cache is much larger than the current TriMedia cache (16 KB or 32 KB). Thus, assuming a medium penalty of 11 cycles per cache miss, an overhead of $16\,\text{words} \times 11\,\text{cycles} = 176$ cycles for reading a block is generated, and, therefore, all the performance improvement provided by the reconfigurable design is lost.

Table 7.1: **MPEG-2 statistics for several conformance bit-strings – excerpt from Table A.2.**

Scene	Blocks/macroblock						Blocks/slice						Blocks/{picture,frame}						Macroblocks with blocks/slice						Coded macroblocks/slice						Slices/{picture,frame}		
	I		P		B		I		P		B		I		P		B		I		P		B		I		P		B		I	P	B
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ	μ	μ	μ
batman	–	–	5.7	0.2	5.2	0.3	–	–	257	9	234	14	–	–	9236	236	8432	258	–	–	45	0	45	0	–	–	45	0	45	0	–	36	36
popplen	6.0	0	3.7	0.9	3.1	1.4	264	0	80	33	38	27	3960	0	1202	198	573	93	44	0	22	8	11	5	44	0	44	0	42	5	15	15	15
sarnoff2	6.0	0	4.1	0.5	2.4	0.4	270	0	171	26	61	23	8100	0	5120	0	1823	26	45	0	42	3	24	7	45	0	44	1	45	1	30	30	30
tennis	6.0	0	4.0	0.4	2.2	0.5	264	0	167	27	71	32	9504	0	5997	221	2563	718	44	0	41	5	31	10	44	0	43	3	44	1	36	36	36
ti1cheer	6.0	0	4.1	0.7	2.8	0.7	264	0	155	56	88	45	7920	0	4481	0	1324	62	44	0	36	11	29	11	44	0	38	10	34	11	30	30	15
Average	6.0	–	4.3	–	3.1	–	266	–	166	–	98	–	n/a	–	n/a	–	n/a	–	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 7.2: **Total number of blocks, macroblocks, slices, and pictures/frames for several MPEG-2 conformance scenes – excerpt from Table A.1.**

Scene	Blocks				Macroblocks with blocks				Coded macroblocks				Slices	{Pictures,Frames}
	I	P	B	Total	I	P	B	Total	I	P	B	Total	IPB	IPB
batman	0	36,943	25,295	62,238	0	6,473	4,847	11,320	0	6,480	4,860	11,340	252	7 (7+0)
popplen	3,960	3,606	1,145	8,711	660	970	338	1,968	660	1,980	1,247	3,887	90	3 (0+6)
sarnoff2	8,100	5,120	3,645	16,865	1,350	1,258	1,468	4,076	1,350	1,332	2,671	5,353	120	4 (4+0)
tennis	9,504	17,992	10,250	37,746	1,584	4,457	4,450	10,491	1,584	4,625	6,305	12,514	288	8 (8+0)
ti1cheer	7,920	4,481	5,275	17,676	1,320	1,049	1,722	4,091	1,320	1,106	2,045	4,471	120	4 (2+4)

Table 7.3: **Cache miss penalty, FPGA context switching overhead, and pipeline fire-up + flushing overhead for different computing scenarios.**

Scene	Macroblock level				Slice level – 16KB Data cache (D$)				Slice level – 32KB Data cache (D$)				Picture/Frame level			
	D$ misses (cycles)	FPGA (cycles)	pipeline (cycles)	**Total** (cycles)	D$ misses (cycles)	FPGA (cycles)	pipeline (cycles)	**Total** (cycles)	D$ misses (cycles)	FPGA (cycles)	pipeline (cycles)	**Total** (cycles)	D$ misses (cycles)	FPGA (cycles)	pipeline (cycles)	**Total** (cycles)
batman	1,369,236	3,396,000	412,624	**5,177,860**	2,688,444	75,600	8,880	**2,772,924**	1,395,020	75,600	8,880	**1,479,500**	4,107,708	2,100	268	**4,110,076**
popplen	191,642	590,400	58,406	**840,448**	285,054	27,000	3,368	**315,422**	196,922	27,000	3,368	**227,290**	574,926	1,800	216	**576,942**
sarnoff2	371,030	1,222,800	104,336	**1,698,166**	617,166	36,000	3,776	**656,942**	389,510	36,000	3,776	**429,286**	1,113,090	1,200	112	**1,114,402**
tennis	830,412	3,147,300	238,300	**3,385,600**	1,238,864	86,400	9,600	**1,334,864**	843,084	86,400	9,600	**939,084**	2,491,236	2,400	288	**2,493,924**
ti1cheer	388,872	1,227,300	129,260	**1,356,560**	625,988	36,000	3,600	**665,588**	399,432	36,000	3,600	**439,032**	1,166,616	1,800	184	**389,236**

Table 7.4: **Pel reconstruction experimental results for the winning computing scenario.**

Routine	Extra tasks: header parsing, motion vectors decoding, ...	2-D IDCT (1-D IDCT on FPGA)		Entropy Decoding (VLD-2 on FPGA)		Inverse Quantization (IQ on FPGA)		Penalties		**Pel reconstruction**	
								cache	FPGA		
Scene	TM & TM+FPGA (cycles)	TM (cycles)	TM+FPGA (cycles)	TM (cycles)	TM+FPGA (cycles)	TM (cycles)	TM+FPGA (cycles)	TM & TM+FPGA (cycles)	TM+FPGA (cycles)	TM (cycles)	TM+FPGA (cycles)
batman	7,122,569	3,485,328	2,000,496	7,435,734	3,553,462	3,236,376	1,997,664	1,395,020	84,480	22,675,027	16,153,691
popplen	1,328,094	487,816	282,120	1,252,879	609,614	401,492	280,912	196,922	30,368	3,667,203	2,728,030
sarnoff2	2,153,508	944,440	543,456	1,964,877	973,039	771,680	542,560	389,510	39,776	6,224,015	4,641,849
tennis	4,495,546	2,113,776	1,217,472	4,675,214	2,287,978	1,839,240	1,214,784	843,084	96,000	13,966,860	10,154,864
ti1cheer	1,708,082	989,856	569,292	2,147,769	1,088,928	816,192	568,488	399,432	39,600	6,061,331	4,373,822

For each iteration that reconstructs a number of pixels, the FPGA context is switched three times to succesively activate VLD-2, IQ-4, and 1-D IDCT. As mentioned, each switching takes 100 cycles. Based on the total number of macroblocks, slices, and $\{$pictures,frames$\}$ in an MPEG string (Table 7.2), we can determine the penalty associated to FPGA context-switching in all three considered computing scenarios. For example, there are 288 slices in the *tennis* scene; when the decoding is carried out at slice level, there are 3 context switches per slice, which translates to $288 \times 3 \times 10 = 8,640$ cycles. A complete list of penalties is presented in Table 7.3.

Two data cache sizes have been considered: 16 KB and 32 KB. As it can be observed, the lowest cumulated penalty is achieved for decoding at slice level. This winning computing scenario is presented *in extenso* in Figure 7.4. For each slice, the variable-length decoding of all macroblocks (headers and DCT coefficients extraction) is first performed. By software pipelining, run-length decoding is carried out in parallel to recreate the 8×8 matrices. Then, all the blocks in the slice are inverse quantized, and DC coefficient prediction for intra-coded macroblocks is carried out. Finally, a burst of 2-D IDCTs is launched in order to complete the reconstruction of the initial matrices of pels.

In connection with the cache miss penalty that is encountered at slice-level decoding, we would like to mention that the number of trashing cache misses approaches to zero if a 32 KB data cache is available. The remaining penalty, e.g., 1,395,020 cycles for the *batman* scene, is mostly due to the compulsory cache misses that are encountered during entropy decoding. Assuming a data cache of only 16 KB, a possible strategy to keep the number of trashing cache misses at low level is to split the slice-level processing into sub-parts. This way, the decoding will be carried out at *sub-slice level*. Since this issue is somehow beyond the paper scope, we will not go into further details.

7.3 Experimental results

The experimental results for the winning computing scenario are presented in Table 7.4. The figures indicate the number of instruction cycles needed to process the MPEG string and the associated overhead. It has to be mentioned that the overhead for setting-up three FPGA contexts at application load time is $3 \times 1,671,360 = 5,014,080$ cycles, which corresponds to about 25 miliseconds on a TriMedia@200MHz. Since long MPEG strings (minutes, hours) are to be decoded on FPGA-augmented TriMedia, this overhead has not been taken into consideration.

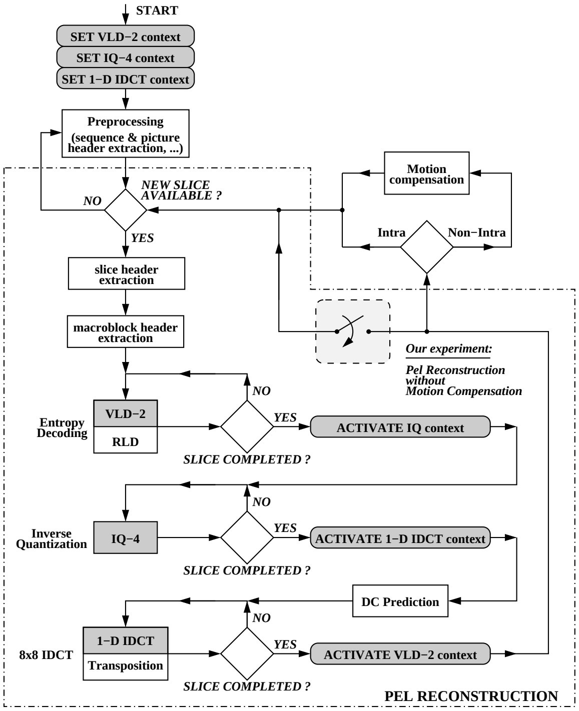


Figure 7.4: **The winning computing scenario of pel reconstruction.**

In Table 7.5, the relative improvement of the FPGA-augmented TriMedia versus standard TriMedia with respect to the number of cycles is shown. For each function that benefit from FPGA support, i.e., entropy decoding, inverse quantization, and IDCT, only the number of instruction cycles needed to perform strictly that particular function are considered (which is similar to assume zero overhead). For the entire pel reconstruction task, all the overhead assuming a 32 KB data cache is included. As it can be observed, the FPGA-augmented TriMedia can perform MPEG2-compliant pel reconstruction with the average improvement of 27% in terms of cycles over the standard TriMedia.

The speed-up figures achieved on FPGA-augmented TriMedia are presented in Figure 7.5. When VLD-2, IQ-4, and 1-D IDCT are each configured on a different FPGA context, *pel reconstruction* can be computed with the average speed-up of $1.4\times$. Assuming that only a single-context FPGA is available, then VLD-2, IQ-4, exclusive-or 1-D IDCT can benefit from reconfigurable hardware support. The speed-up decreases to $1.1\times$ when either IQ-4 or 1-D IDCT are configured on the RFU, and to $1.2\times$ when only VLD-2 is configured on the RFU.

Table 7.5: **Pel reconstruction relative improvement.**

Scene	batman	popplen	sarnoff2	tennis	ti1cheer
Entropy Decoding (VLD-2 on FPGA)	52.2%	51.3%	50.5%	51.1%	49.3%
Inverse Quantization (IQ-4 on FPGA)	38.3%	30.0%	29.7%	34.0%	30.3%
2-D IDCT (1-D IDCT on FPGA)	42.6%	42.2%	42.5%	42.4%	42.5%
Pel reconstruction (VLD-2, IQ-4, 1-D IDCT on FPGA)	**28.8%**	**25.6%**	**25.4%**	**27.3%**	**27.8%**

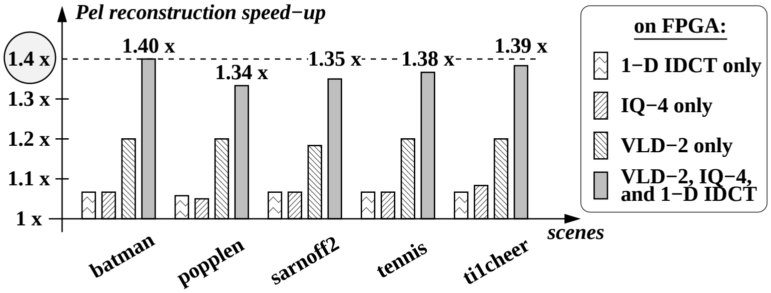


Figure 7.5: ***ρ*–TriMedia versus standard TriMedia speed-up.**

At the end, we would like to emphasize that TriMedia is a 5-issue slot VLIW processor with a 64-bit datapath and a very rich multimedia-oriented instruction set [113], and such improvements have been obtained within TriMedia target media processing domain [88]. For each function that benefited from FPGA support, a high-performance pure-software design constitutes the reference implementation: 4 out of 5 issue slots are filled in with operations in the pure-software entropy decoder [99], and more that 4.5 out of 5 issue slots are occupied in the pure-software IQ and 2-D IDCT. Given the fact that additional operations per cycle cannot be issued, providing more hardwired functional units (however, with the exception of a hardwired 1-D IDCT) is not likely to lead to computational improvement.

Another possibility would be to eliminate existing hardwired functional units from the processor to make extra room for the RFU. Since the host processor has to provide for fast pointer arithmetic, a complete removal of the hardwired functional units is not likely to be a viable solution. Consequently, a 32-bit adder, for example, will exist almost for sure in any future processor implementation. Therefore, a SIMD adder is also likely to exist, since a 16-bit (2-way) SIMD addition, for example, can be implemented on a 32-bit adder by simply breaking the carry path in the middle. However, this is still an open question and, therefore, subject to future work.

7.4 Conclusion

In this chapter we addressed pel reconstruction and proposed a reconfigurable pel reconstruction design for ρ–TriMedia. In particular, we analyzed pel reconstruction according to three computing scenarios: each of the Entropy Decoding, IQ, and IDCT routines returns only after an entire (1) macroblock, (2) slice, and (3) picture/frame is fully processed. The experimental results carried out on a TriMedia–CPU64 cycle-accurate simulator indicated that the second scenario, i.e., processing at slice level, is the winner. By configuring each of the VLD-2, IQ-4, and 1-D IDCT computing units on a different FPGA context, and by activating each context once per slice, pel reconstruction can be performed on ρ–TriMedia with a speed-up of 40% over the TriMedia-CPU64.

Since motion compensation is a memory-dominant task, the required arithmetic being a simple addition per pixel, it is likely not to be subject to reconfigurable hardware support. However, this leaves the following general open question:

1. Can a memory-dominant task benefit from reconfigurable hardware support?
2. Can a control-dominant task benefit from reconfigurable hardware support?

As a final remark we would like to emphasize that our results are important since, due to the removal of the redundancy in the signal during coding, a decoding process is essentially a sequential task. For this reason, a decoding process is typically an intricate function on TriMedia–CPU64, since a VLIW architecture must benefit from instruction-level paralellism in order to be efficient.

In the next chapter we will address color space conversion. We would like to mention that this function is not part of the MPEG decoding process. However, it is usually carried out at the end of MPEG decoding if the video information is to be displayed on a monitor.

Chapter 8

YCC-to-RGB Color Space Conversion

The back-end stage of MPEG decoding consists of a color space conversion task, which transforms the video data representation from a luminance (Y) and two chrominance (C) components to three color components (RGB) usually supported by any video display. Traditionally, color space conversion has been implemented in hardware, as a coprocessor next to a general-purpose computing engine, or in software in media-domain processors. In this chapter, we describe a reconfigurable YCC-to-RGB design for ρ–TriMedia, and demonstrate that significant speed-up for $Y'CbCr$-to-$R'G'B'$ color space conversion can be achieved on FPGA-augmented TriMedia–CPU64 over standard TriMedia–CPU64.

Essentially, $Y'CbCr$-to-$R'G'B'$ color space conversion is a linear transform from $Y'CbCr$ color space to $R'G'B'$ color space. As we describ subsequently, this transform exhibits large data and instruction-level parallelisms, and thus it can be implemented on TriMedia–CPU64 with a very high efficiency. Obtaining improvements for a task having a computational pattern which TriMedia–CPU64 has been optimised for, is a challenging task.

The chapter is organized as follows. For background purpose, we present the most important issues related to color space conversion in Section 8.1. A pure-software implementation of the color space conversion task is outlined in Section 8.2. An FPGA-mapped computing unit that can perform color space conversion for four pixels per call is presented in Section 8.3. The execution scenario of the $Y'CbCr$-to-$R'G'B'$ conversion on ρ–TriMedia, as well as experimental results are presented in Section 8.4. Section 8.5 completes the chapter with some conclusions and closing remarks.

8.1 Theoretical background

According to the Trichromatic Theory, it is possible to match all of the colors in the visible spectrum by appropriate mixing of three primary colors. Which primary colors are used is not important as long as mixing two of them does not produce the third. For display systems that emit light, the Red-Green-Blue (RGB) primary system is used.

A color space is a mathematical representation of a set of colors. In this section, we present two color spaces: $R'G'B'$ and $Y'CbCr$.

$R'G'B'$ color space. Film, video, and computer-generated imagery all start with red, green, and blue intensity components. In video and computer graphics, the nonlinearity of the CRT monitor is compensated by applying a nonlinear transfer function to RGB intensities to form *Gamma–Corrected Red*, *Green*, and *Blue* ($R'G'B'$). The gamma-corrected red, green, blue are defined on a relative scale from 0 to 1.0, chosen such that shades of gray are produced when $E'_R = E'_G = E'_B$, where $E'_.$ denotes the analog gamma–pre-corrected signal associated with the primary X color.

In digital video, the analog signal is uniformly-quantized on 8 bits, so that 256 equally spaced quantization levels are specified. Coding range in computing has a *de facto* standard excursion, 0 to 255. Studio video provides footroom below the black code, and headroom above the white code; its range is standardized from 16 to 235. However, values less than 16 and greater than 235 are allowed in order to accomodate the transients that result from filtering.

$Y'CbCr$ color space. The data capacity related to color information in a video signal can be reduced as follows. First, $R'G'B'$ is transformed into luminance-related quantity called *luma* (Y'), and two color difference components called *chroma* (Cb, Cr) [83]. Since the human visual system has poor color acuity, the color detail can then be reduced by subsampling (lowpass filtering) without the viewer noticing.

The $Y'CbCr$ color space was developed as part of ITU-R Recommendation BT.601 [52]. All components are represented as 8-bit unsigned integers. Y' is defined to have a nominal range of 16 to 235; Cb and Cr are defined to have a range of 16 to 240, with 128 equal to zero. It is Y', Cb, Cr values that is coded inside an MPEG string. It is worth mentioning that during the MPEG decoding process, Y', Cb, Cr are each represented on a 16-bit signed integer after motion compensation.

$Y'CbCr$-to-$R'G'B'$ conversion. If the gamma-corrected RGB data has a range of 0 to 255, as is commonly found in computer systems, the following equations describe the $R'G'B'$-to-$Y'CbCr$ conversion:

$$\begin{cases} R' = 1.164(Y' - 16) + 1.596(Cr - 128) \\ G' = 1.164(Y' - 16) - 0.813(Cr - 128) \\ \qquad\qquad\qquad\qquad\quad - 0.391(Cb - 128) \\ B' = 1.164(Y' - 16) + 2.018(Cb - 128) \end{cases} \tag{8.1}$$

Even though Y' is defined to have a range from 16 to 235, while Cb and Cr have a range of 16 to 240, sample values outside the above mentioned ranges may occasionally occur at the output of the MPEG-2 decoding process according to ITU-T Recommendation H.262 [53]. Thus, $R'G'B'$ values must be saturated at the 0 and 255 levels after conversion to prevent any overflow and underflow errors.

With connection to the subsequent experiment, we would like to mention that the mapping defined by Equation set 8.1 will benefit from configurable hardware support.

$Y'CbCr$ sampling format conversion. As mentioned, since the eye is less sensitive to color information than brightness, the chroma channels can have a lower sampling rate that the luma channel without a dramatic degradation of the perceptual quality. In MPEG, 2:1 horizontal downsampling with 2:1 vertical downsampling is employed. That is, the Cb and Cr pixels lie between the Y' pixels on every other pixel on both the horizontal and vertical lines. Thus, a two-dimensional 2-fold upsampling has to be carried out before the proper color space conversion.

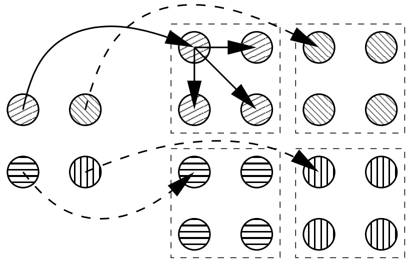


Figure 8.1: **Two–dimensional 2–fold upsampling by replication.**

The simplest upsampling method employs a *zero-order hold*. That is, each and every pixel is replicated to East, South-East, and South, as depicted in Figure 8.1. Consequently, no additional processing is needed for upsampling at the expense of a poor frequency response characteristic. Indeed, since the zero-order hold does not possess a sharp cutoff–frequency response characteristic, it is a poor *anti-image* filter, and it passes undesirable image components [111].

The filtering process is beyond the chapter scope. Thus, we will consider that a zero-order hold is used, although this solution does usually not provide satisfactory image quality. Synthesizing, the following stages have to be performed in the process of color space conversion carried out at the end of MPEG decoding: (1) *chroma* upsampling, and (2) $Y'CbCr$-to-$R'G'B'$ linear transform.

In the next section, we will present several considerations about performing color space conversion in software. We will mainly focus on the routine organization emphasizing the optimization techniques that expose to the compiler the available parallelism inside the color space conversion task.

8.2 YCC-to-RGB pure-software implementation

As evidenced in Algorithm 8, the pure-software color space converter consists of three imbricated loops. The first loop iterates over the number of frames in an MPEG scene. Since the outer loop can include additional MPEG-related code, e.g., motion compensation, it is subject for further optimization when an entire MPEG decoding process is considered. Since such optimization is beyond the scope of the chapter, it is not considered any longer.

Algorithm 8 Color space conversion – pure software solution.

```
for $i = 1$ to NO_OF_FRAMES do
    for $j = 1$ to IMAGE_VERTICAL_SIZE step 2 do
        for $k = 1$ to IMAGE_HORIZONTAL_SIZE step 2 do
            READ vec64sh_Y_top_left, vec64sh_Y_top_right, vec64sh_Y_bottom_left,
                    vec64sh_Y_bottom_right, vec64sh_Cb, vec64sh_Cr

            Color space conversion (4-way SIMD style) according to Equations 8.1

            WRITE vec64ub_R_top, vec64ub_R_bottom, vec64ub_G_top,
                    vec64ub_G_bottom, vec64ub_B_top, vec64ub_B_bottom
        end for
    end for
end for
```

The inner loops iterate over the vertical, and horizontal size of the image, respectively. Since the loop iterating over the vertical size include the loop iterating over the horizontal size, while the Contor $\boldsymbol{j}$ is incremented by two each iteration, two image rows are processed at a time. In addition, the Contor $\boldsymbol{k}$ of the inner loop is also incremented by two each iteration. The main idea of this approach is to avoid any interlocks across the iteration boundaries together with minimizing the number of read-in operations with respect to the number of pixels to be converted.

Four luminance and two chrominance vectors, each containing four 16-bit signed integers, are read in from memory at the beginning of the inner iteration. With these values, two vectors for Red, two for Green, and two for Blue color, each of them containing eight 8-bit unsigned integers, can be uniquely computed. That is, from $4 \times 4 + 4 + 4 = 24$ 16-bit signed integers corresponding to 16 pixels of the image represented in the $Y'CbCr$ format, $2 \times 8 + 2 \times 8 + 2 \times 8 = 48$ 8-bit unsigned integers corresponding to the same 16 pixels represented in the $R'G'B'$ format are generated. This process is summarized in Figure 8.2.

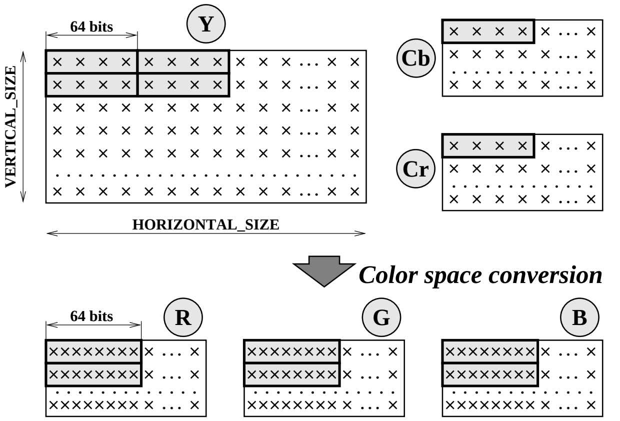


Figure 8.2: **Color space conversion strategy for the inner loop.**

In addition, four pixels are processed simultaneously by means of 4-way SIMD-style operations. With this strategy, 4.8 out of 5 issue slots are filled in with operations. That is, the TriMedia–CPU64 runs close to its full processing power, making from the pure-software implementation a challenging reference for the reconfigurable design.

8.3 YCC-to-RGB implementation on FPGA

Since three values (red, green, and blue) are to be computed for each pixel, we propose to provide configurable-hardware support for a 3-slot CSC operation which reads the $Y'CbCr$ triplet and returns the $R'G'B'$ triplet:

$$\textbf{CSC} \quad \boldsymbol{Y', Cb, Cr \longrightarrow R', G', B'}$$

where Y', Cb, Cr, R', G', B' are all 64-bit registers. Subject of the FPGA logic capacity and the number of FPGA I/O pins, a different number of pixels can be processed in parallel. Given the fact that the *luma* and *chroma* are represented as 16-bit signed integers, and gamma-corrected red, green, and blue are represented as 8-bit unsigned integers, at most four pixels can be processed in parallel. Indeed, the CSC is a 4-way SIMD operation which transforms three 16-bit signed integer vectors (Y', Cb, Cr) into three 8-bit unsigned integer vectors (R', G', B'). This translates to a number of $3 \times 4 \times 16 + 3 \times 4 \times 8 = 288$ I/O pins, which is acceptable for most FPGAs in general, and ACEX EP1K100 device in particular.

For the TriMedia–CPU64 simulator does not support super-operations on more than 2 slots for the time being, our 3-slot CSC operation has to be emulated by sequences of 1- and/or 2-slot operations. Thus, we define two 2-slot CSC instructions: CSC_R, which performs the proper color space conversion and returns only the *red* information, and CSC_GB, which returns the *green* and *blue* information:

$$\begin{aligned} &\textbf{CSC\_R} \quad \boldsymbol{Y', Cb, Cr} \longrightarrow \boldsymbol{R'} \\ &\textbf{CSC\_GB} \qquad \longrightarrow \boldsymbol{G', B'} \end{aligned}$$

We have to emphasize that this approach is carried out only for experimental purpose. Fortunately, our choice does not generate overhead, since it is easier to schedule a single 3-slot instruction than multiple 1- and/or 2-slot instructions.

The FPGA–based CSC unit implementing the Equation set 8.1, is presented in Figure 8.3 (the Roman numerals indicate the pipeline stages). By writing RTL-level VHDL code, we succeeded to identify a four-stage pipelined implementation which can run at 100 MHz on ACEX EP1K100 device. Adding the penalty of the extra read and write back cycles for an RFU–based operation, the CSC unit has a latency of 10 and recovery of 2 cycles if a ρ–TriMedia instance with a core TriMedia@200MHz and an FPGA@100MHz (i.e., an FPGA which circuits running at a frequency of maximum 100MHz can be mapped on) is considered. For an FPGA@50MHz, two pipelines stages can be merged into one, which translates into a CSC having the latency of 10 and recovery of 4.

8.4 YCC-to-RGB implementation on ρ–TriMedia

Summarizing the previous section, for the first ρ–TriMedia instance, a CSC computing unit having the latency of 10 and recovery of 2 cycles is configured on the FPGA@100MHz, while two CSC computing units with the latency of 10 and recovery of 4 cycles is configured on the FPGA@50MHz in the second ρ–TriMedia instance. That is, a lower pipeline frequency at the expenses of a double size FPGA is the trade-off of the second instance.

To perform color space conversion for an image, calls to CSC are issued within a software loop. The scheduled code when the FPGA@100MHz is considered

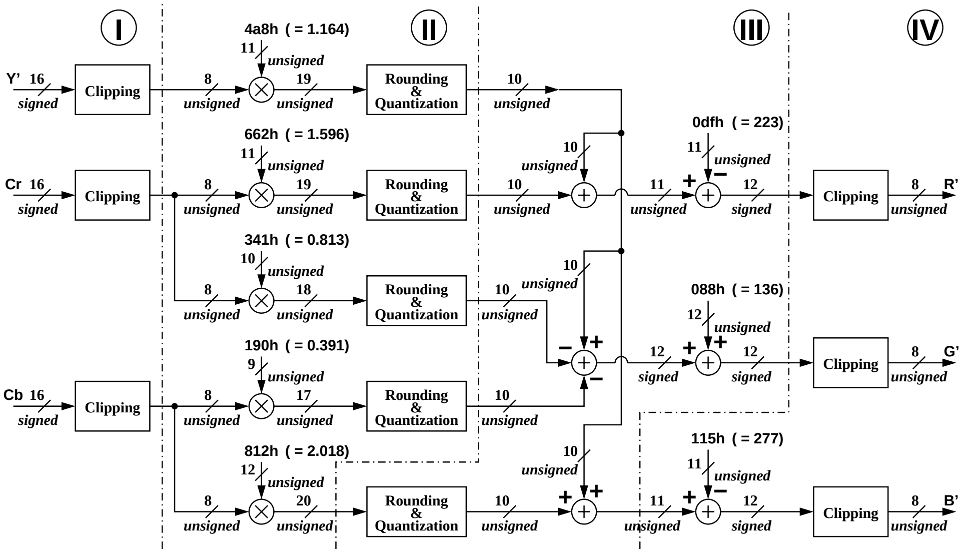


Figure 8.3: **The color space converter implementation on FPGA.**

is presented in Figure 8.4. First, `LOAD` operations are issued to fetch the pixels in $Y'CbCr$ format from memory. Then, pairs of `CSC_R` + `CSC_GB` operations are launched to perform color space conversion, four pixels per call. After eight pixels have been converted, `PACK` operations reorganize the $R'G'B'$ information in 8-bit unsigned integer vectors. Finally, `STORE` operations send the results to a display FIFO.

According to Figure 8.4, 16 pixels can be processed with the latency 25 cycles. In order to keep the pipeline full, back-to-back CSC_R operation is needed. That is, a new CSC_R instruction has to be issued every two cycles (or, every four cycles in the FPGA@50MHz–based ρ–TriMedia instance). This way, color space conversion can be performed with a throughput of 2 pixels/cycle. Unfortunately, this figure corresponds to the ideal case of infinite loop unrolling, which can never be achieved in practice. For a finite loop unrolling, the overhead associated to firing-up and flushing the CSC pipeline has to be taken into consideration. As a rule of thumb, the throughput drops to $N/(\mathit{latency}/16 + (N-1)/\mathit{ideal\ throughput})$, where N is the number of times which the loop is unrolled. For example, the ideal throughput drops to 1.3 pixels/cycle for $4\times$ loop unrolling, and to 0.64 pixels/cycle for a loop which is fully rolled. The same judgement can be carried out for the FPGA@50MHz–based ρ–TriMedia instance. Since the results are pretty much the same, we will not go into details.

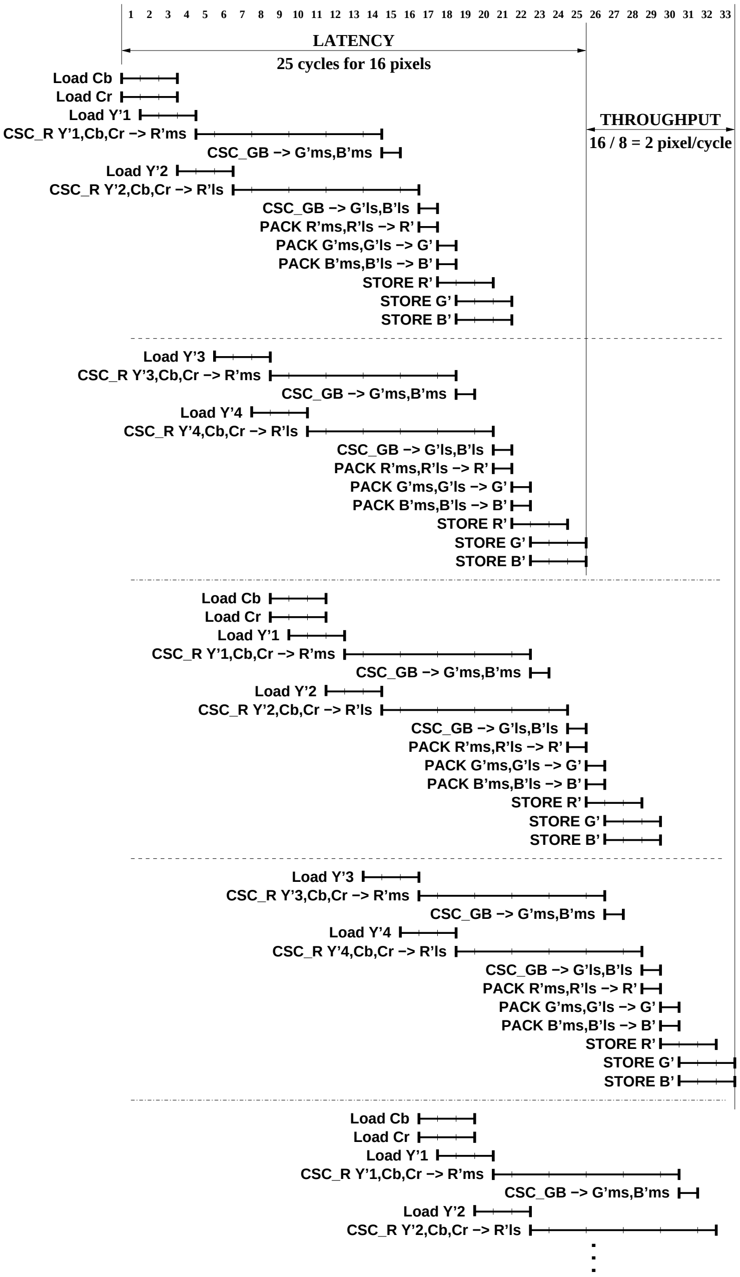


Figure 8.4: **Scheduling result for a CSC unit having the latency of 10 and recovery of 2.**

The testing database for both pure-software and FPGA-based color space converters consists of a stream of two 256×256 pixel images, for which the Y', Cb, Cr components are stored in separate tables in the main memory. The data is organized as 16-bit signed integer vectors as resulted from motion compensation. The $Y'CbCr$-to-$R'G'B'$ conversion is done in an SIMD fashion, by sequentially processing four triplets of Y', Cb, Cr values at a time. As mentioned, *chroma* 2-fold upsampling is performed by means of a zero-order hold; this way, no additional processing is needed. Then, the linear mapping defined by Equation set 8.1 is carried out, and the result is sent to a display FIFO.

Since all the input data is stored into memory, the very first access to it (which is actually the single access in our application) will generate so called *compulsory* read cache misses. The sum of the data cache stalls and the instruction cycles needed to perform the proper color space conversion translates to a *worst case* scenario, which provides the lower bound of the performance improvements. We will also present the results according to the *best case* scenario, in which all cache misses are counted as part of the motion compensation process. That is, improvements in terms of instruction cycles required to perform strictly color space conversion will be reported.

The reference for evaluating the performance of the color space conversion carried out on FPGA-augmented TriMedia is a pure-software implementation on standard TriMedia [94]. The reference color space converter is implemented as a loop, where each iteration processes 16 pixels. Since this pure-software implementation is beyond the chapter scope, we will not go into further details. However, we still mention that by running our pure-software color space converter on a TriMedia cycle accurate simulator, we determined that an iteration which processes 16 pixels can be scheduled into 24 cycles, which translates into 0.66 pixels/cycle. It is also worth mentioning that 4.8 of 5 issue slots are filled in with operations in the pure-software implementation. This result is indeed a challenging reference for the TriMedia+FPGA hybrid.

Therefore, our experiment includes two approaches: *pure software* and *FPGA-based*. As mentioned, 0.66 pixels/cycle are decoded in the pure software approach, while 2 pixels/cycle can be decoded in the FPGA-based approach if the loop is unrolled an infinite number of times. The configuration of the RFU is carried out at application load time.

The $Y'CbCr$-to-$R'G'B'$ performance evaluation has been carried out considering the two mentioned FPGA-augmented TriMedia instances: TriMedia + FPGA@100MHz and TriMedia + (double-size) FPGA@50MHz. A program has been written in C, and further compiled and scheduled with TriMedia development tools. To overcome the penalty associated to firing-up and flushing the pipeline, two techniques can be employed: (1) *loop unrolling*, and (2) *software pipelining*. Since the TriMedia scheduler uses the decision tree as a scheduling unit [50], all operations return their results in the same decision tree that they are issued, even though the TriMedia architecture does not forbid the contrary. This is the major limiting factor in generating deep software pipelined loops which contains long-latency operations starting from a C-level description. Since programming in assembly seems to be too complex for color space converision, only *loop unrolling* technique is considered subsequently.

The loop calling the CSC instruction has been manually unrolled a different number of times. The best results which corresponds to $4\times$ unrolling are presented in Table 8.1. We would like to mention that a $2\times$ unrolling does not suffice, since the firing-up and flushing overhead is still too large. At the same time, an $8\times$ unrolling generates long decision trees, which in turn translates into reduced performance due to register spilling. Also, unrolling does not provide a significant improvement in the pure-software approach, since 4.8 out of 5 issue slots are filled in with operations in this case. As it can be easily observed, about the same performance figures are obtained for each FPGA-augmented TriMedia instances. The speed-up is $\frac{0.66-0.47}{0.47} \times 100 \approx 40\%$ according to the *worst-case* scenario, and $\frac{1.22-0.66}{0.66} \times 100 \approx 85\%$ according to the *best-case* scenario.

Table 8.1: **Performance figures for $Y'CbCr$-to-$R'G'B'$ conversion.**

Experiment		1 CSC on FPGA@100MHz		2 CSCs on FPGA @50MHz	
	pure SW (rolled)	FPGA-CSC4 rolled	FPGA-CSC4 unrolled 4×	FPGA-CSC4 rolled	FPGA-CSC4 unrolled 4×
Instruction cycles	190,771	190,240	107,821	192,972	109,358
Instruction cycles / pixel	1.46	1.45	0.82	1.47	0.83
Pixels / instruction cycle	0.66	0.69	1.22	0.68	1.20
Instruction-cache stalls	937	697	1,342	702	1,368
Read data-cache stalls	90,112	90,112	90,112	90,112	90,112
Issues / cycle	4.81	3.47	3.84	3.35	3.95
I cycles + read D$ stalls	280,883	280,352	197,933	283,084	199,470
Instruction cycles / pixel	2.14	2.14	1.51	2.16	1.52
Pixels / instruction cycle	0.47	0.47	0.66	0.46	0.66

Finally, we would like to mention that a more realistic *average-case* scenario will assume a memory access pattern for reducing the number of read cache misses. However, this is subject to optimization at a complete MPEG decoder level, and therefore, beyond the chapter scope. Thus, we do not have statistically relevand data for the time being. Since we are not able to make a reliable estimation according to the *average-case* scenario, we proceed to a conservative evaluation by taking into account the entire number of read cache misses (worst-case scenario), and claim that the FPGA-augmented TriMedia–CPU64 can perform color space conversion 40% faster than the standard TriMedia–CPU64. Given the fact that the experimental TriMedia is a 5 issue-slot VLIW processor with 64-bit datapaths and a very rich multimedia instruction set [113], such an improvement within the target media processing domain indicates that the TriMedia + FPGA hybrid shows clear benefits for doing color space conversion.

Now since our experiments are done, we would like to make several considerations in connection to the penalty induced by the FPGA core area. According to DeHon, an additional area of 500,000 to 1,000,000 λ^2 per LUT is needed by the reconfigurable core [30]. For the ACEX EP1K100 device that includes about 5,000 LUTs, this figure translates to an average area of 30 mm^2 in a 0.18 μm technology. Giving the fact that TriMedia–CPU64 occupies about 35 mm^2 in the same technology, the FPGA augmentation with an EP1K100 device leads to an area increase of the processor area of 85%. However, this increase is not a major concern, since TriMedia was initially envisioned to be embedded on the same die with a set of coprocessors, which also required additional area. For example, in the Viper chip [32], the MPEG video decoder and video coprocessor occupy together an area of about 18 mm^2. Since such coprocessors cannot be used for any other task they have been designed for, trading-off hardwired coprocessors for a reconfigurable core is a good approach.

8.5 Conclusion

In this chapter we demonstrated that significant speed-up for $Y'CbCr$-to-$R'G'B'$ color space conversion can be achieved on FPGA-enhanced TriMedia over standard TriMedia. The main idea is to configure a pipelined Color Space Converter (CSC) on FPGA and to unroll the software loop issuing an CSC operation such that the penalty associated to firing-up and flushing the CSC pipeline is reduced. In particular, we provide configurable-hardware support for a CSC operation which can process four pixels per call. When mapped on an ACEX EP1K100 FPGA from Altera, the computing unit performing the CSC operation has a latency of 10 and recovery of 2 TriMedia@200 MHz cycles, and occupies 57% of the device. The simulations carried out on a TriMedia cycle accurate simulator indicate that by configuring the CSC unit on FPGA at application load-time, $Y'CbCr$-to-$R'G'B'$

color space conversion can be computed on extended TriMedia 40% faster over the standard TriMedia.

Chapter 9

Conclusions

We have addressed the augmentation of the TriMedia–CPU64 processor with a field-programmable gate array, and assessed the potential gain in performance such hybrid achieves when performing MPEG2 decoding. In essence, we have proposed an extension of the TriMedia–CPU64 instruction set architecture that incorporates support for the reconfigurable core. Our extension can be easily embedded into the TriMedia–CPU64 development tools since it requires only minor changes of the compiler, scheduler, and assembler. Then, we focused on a number of multimedia-oriented kernels: Inverse Discrete Cosine Transform (IDCT), Inverse Quantization (IQ), Entropy Decoder, Pel Reconstruction, and YCC-to-RGB Color Space Converter. The research activity called for a reimplementation of these kernels, and so included algorithm research, high-level architecture design, and VHDL design. A significant effort has been made to design FPGA-based computing units. Overall, we have shown that the augmentation of the TriMedia–CPU64 processor with a reconfigurable core results in performance-wise advantages at the expenses of a medium-size FPGA on the order of 5,000 LUTs and twelve 8-input 16-output RAMs.

This chapter summarizes our overall investigations and achievements. It is organized in three sections. Section 9.1 discusses the overall conclusions. Section 9.2 presents the major contributions. Section 9.3 proposes further research directions.

9.1 Summary

In this dissertation, we considered and solved a number of issues associated with reconfigurable computing technology. Our overall achievements can be summarized by the following.

In Chapter 2 we provided for a brief survey of the reconfigurable computing domain and proposed a taxonomy of Field-programmable Custom Computing Machines (FCCM). Since the programmer observes only the architecture of a computing machine, the previous classifications using implementation-based criteria do not seize well the implications of the new reconfigurable computing paradigm as perceived by the user. For this reason, we classified the FCCMs according to architectural criteria. In order to analyze the phenomena inside FCCMs, yet without reference to a particular instruction set, we introduce a formalism based on microcode, in which any task (operation) performed by a field-programmable computing facility is executed as a microprogram with two basic stages: `SET CONFIGURATION`, and `EXECUTE CUSTOM OPERATION`. Two classification criteria have been considered:

- The verticality/horizontality of the microcode.
- The explicit availability of a `SET` instruction.

Our approach is particularly important since it allows a view on an FCCM at the the architectural level, decoupled from lower implementation and realization hierarchical levels.

In Chapter 3 we described the arhitecture of the ρ–TriMedia processor consisting of a standard TriMedia–CPU64 core augmented with an FPGA–based Reconfigurable Functional Unit (RFU). In order to use the RFU, TriMedia–CPU64 instruction set is augmented with a kernel of new instructions: `SET_CONTEXT`, `ACTIVATE_CONTEXT`, and `EXECUTE`. Loading context information into RFU configuration memory is performed under the command of a `SET_CONTEXT` instruction, while the `ACTIVATE_CONTEXT` instruction swaps the active configuration with one of the idle on-chip configuration. `EXECUTE` instructions launch the operations performed by the FPGA-mapped computing units. With these new instructions, the user is given the freedom to define and use any computing facility subject to the FPGA size and TriMedia organization.

In the same chapter, we also proposed to use the subsequent opcode fields, which are currently set `NOP`s in a hardwired super-operation, as an argument for the RFU OPCODE. In this way, a large number of RFU-specific operations can be encoded, while only a single entry for the generic `EXECUTE` instruction needs to be allocated in the opcode space. We would like to mention that all the opcode fields in the 5-slot VLIW instruction are decoded separately, but only when the first opcode field specifies an `EXECUTE` instruction the subsequent opcodes are interpreted as an argument of the RFU OPCODE, and thus decoded locally at the RFU. This way, the generic `EXECUTE` instruction does not create pressure on the instruction decoder, neatly fits in the existing instruction format, fits the existing connectivity structure to the register file, and hence requires very little hardware overhead.

To demonstrate the effectiveness of ρ–TriMedia, we addressed MPEG decoding and proposed a reconfigurable design for each of the following multimedia kernels: IDCT, Entropy Decoder, IQ, Pel Reconstruction, and YCC-to-RGB Color Space Converter.

In Chapter 4 we described the computation of the 8×8 (2-D) IDCT on such extended TriMedia and propose a scheme to implement the 1-D IDCT operation on the RFU. When mapped on an ACEX EP1K100 FPGA, the proposed 1-D IDCT exhibits a latency of 16 and a recovery of 2 TriMedia@200 MHz cycles, and occupies 45% of the logic cells of the device. By configuring the 1-D IDCT computing facility on an RFU context, an IEEE-compliant 2-D IDCT can be computed with the throughput of 1/32 IDCT/cycle. This is an improvement of about 75% in terms of throughput over the standard TriMedia–CPU64.

In Chapter 5 we described an IQ-4 FPGA-based computing unit that can inverse quantize four DCT coefficients per call. When mapped on an ACEX EP1K100 FPGA, the proposed IQ-4 exhibits a latency of 18 and a recovery of 2 TriMedia@200 MHz cycles, and occupies 43% of the logic cells of the device. By configuring the IQ-4 computing facility on an RFU context, an 8×8 block of DCT coefficients can be inverse quantized with the throughput of 1/32 IDCT/cycle. This is an improvement of about 55% in terms of throughput over the standard TriMedia–CPU64.

In Chapter 6 we addressed entropy decoding and proposed a strategy to partially break the data dependency related to variable-length decoding. Three VLDs (VLD-1, VLD-2, VLD-3) instructions that can return 1, 2, or 3 symbols per call, respectively, were analyzed. We determined that VLD-2 instruction leads to the most efficient entropy decoding in terms of instruction cycles and FPGA area. The FPGA-based implementation of VLD-2 computing unit was also described in detail. When mapped on an ACEX EP1K100 FPGA, VLD-2 exhibits a latency of 8 TriMedia@200 MHz cycles, and uses all the Electronic Array Blocks and 51% of the logic cells of the device. The simulation results indicated that the VLD-2–based entropy decoder is more than 50% faster than its pure-software counterpart.

Chapter 7 combines the previously described reconfigurable designs for Entropy Decoder, Inverse Quantization, and Inverse Discrete Cosine Transform, and assesses the performance gain such extensions have when performing MPEG2-compliant Pel Reconstruction. Experimental results indicate that by configuring each of the VLD-2, IQ-4, and 1-D IDCT computing facilities on a different FPGA context, and by activating the contexts as needed, the FPGA-augmented TriMedia can perform MPEG2-compliant pel reconstruction with a speed-up of $1.4\times$ over the standard TriMedia.

In Chapter 8 we addressed Color Space Conversion (CSC), analyzed a CSC-4 instruction processing four pixels per call, and proposed a scheme to implement a CSC-4 unit on RFU. When mapped on an ACEX EP1K100 FPGA, the CSC-4 exhibits a latency of 10 and a recovery of 2 TriMedia@200 MHz cycles, and occupies 57% of the device. By configuring the CSC-4 facility on one RFU context at application load-time, color space conversion can be computed on FPGA-augmented TriMedia with a speed-up ranging from $1.4\times$ to $1.85\times$ over the standard TriMedia.

9.2 Contributions

The major contributions of this study can be summarized as follows.

- We have proposed a taxonomy of field-programmable custom computing machines using a formalism based on microcode. Two architectural issues have been used as classification criteria:
  - The verticality/horizontality of the microcode.
  - The explicit availability of a `SET` instruction.
- We have proposed an extension of TriMedia–CPU64 instruction set architecture that incorporates support for the reconfigurable array. The architectural extension consists of a kernel of three new instructions: `SET_CONTEXT`, `ACTIVATE_CONTEXT`, and `EXECUTE`.
- In order to provide architectural support for the reconfigurable core, we proposed to use the subsequent opcode fields, which are currently set `NOP`s in a hardwired super-operation, as an argument for the RFU OPCODE. In this way, a large number of RFU-specific operations can be encoded, while only a single entry for the generic `EXECUTE` instruction needs to be allocated in the opcode space. In the same time, the `EXECUTE` instruction neatly fits in the existing instruction format, too.
- We have proposed a programming methodology for FPGA-augmented TriMedia–CPU64 consisting of a C-level programming model and ACEX EP1K100 FPGA recommended mapping strategies.
- We have addressed the computation of 8×8 (2-D) IDCT on FPGA-augmented TriMedia–CPU64 and proposed a scheme to implement a 1-D IDCT computing unit on FPGA. We have determined that by configuring the 1-D IDCT on an RFU context, the 8×8 (2-D) IDCT can be com-

puted on FPGA-augmented TriMedia–CPU64 75% faster than on the standard TriMedia–CPU64.

- We have proposed an FPGA-based IQ-4 computing unit that can inverse quantize four DCT coefficients per call. We have determined that by configuring the IQ-4 on an RFU context, an 8×8 block of DCT coefficients can be inverse quantized on FPGA-augmented TriMedia–CPU64 55% faster than on the standard TriMedia–CPU64.
- We have addressed Entropy Decoding and proposed a strategy to partially break the data dependency related to variable-length decoding. We also proposed an FPGA-based implementation of a VLD-2 unit that can decode two variable-length symbols per call. We have determined that VLD-2–based entropy decoder is 50% faster than its pure-software counterpart.
- We have combined the previous three reconfigurable designs and determined that by configuring each of the VLD-2, IQ-4, and 1-D IDCT computing facilities on a different RFU context, and by activating the contexts as needed, the FPGA-augmented TriMedia can perform MPEG2-compliant pel reconstruction 40% faster that standard TriMedia.
- We have proposed an FPGA-based CSC-4 computing unit that can perform YCC-to-RGB color space conversion for four pixels per call. We have determined that by configuring the CSC-4 on an RFU context, YCC-to-RGB color space conversion can be performed on FPGA-augmented TriMedia–CPU64 40–85% faster than on the standard TriMedia–CPU64.
- Overall, we showed that significant improvement on FPGA-augmented TriMedia–CPU64 over standard TriMedia–CPU64 can be achieved with a medium-size (multiple-context) FPGA on the order of 5,000 4-input LUTs and 12 8-input 16-output RAMs. Given the fact that TriMedia–CPU64 is a 5-issue slot VLIW processor with 64-bit datapaths and a very rich multimedia instruction set, such an improvement within its target media processing domain indicates that ρ–TriMedia is a promising approach.

9.3 Proposed research directions

As a continuation of the research we suggest the following.

- In this dissertation we addressed only computing-dominant tasks. An interesting research area is to investigate whether the reconfigurable computing paradigm can provide improvements for control-dominant tasks, e.g., MPEG header extraction, or memory-dominant tasks, e.g., motion compensation.

- Since standard TriMedia provides a good support for transposition, the circuitry mapped on FPGA was essentially feed-forward. Thus, a rich 2-D–oriented interconnection network is actually not needed. Since 90% of the silicon area is used by the interconnection network in the current FPGAs, an interesting research direction would be to investigate a new FPGA architecture with only an 1-D–oriented interconnection network.

- A 2-D–oriented convolution is typically carried out by decomposing it into 1-D–oriented (vertical and horizontal) subtasks with a transposition stage in between. Thus, a 2-D–oriented convolution is a symmetrical tasks that does not require a 5-port register file. For this reason, a possible research direction would be to investigate a partitioned register file.

- In this dissertation we addressed basically video decompression tasks. As future work, it would be interesting to investigate more complex tasks, such as vector quantization and pattern recognition.

- One of the major problems of the multiple-context FPGAs is the large power consumption during a context switch. Thus, a new research direction would be to investigate strategies to reduce the power consumption in multiple-context FPGAs.

- In order to reduce the size of the reconfiguration information and thus make launching `SET` efficient at run-time, we suggest to investigate an FPGA architecture that supports partial reconfiguration. We would like to remind that in this dissertation we assumed that `SET` is launched at application load-time.

Appendix A

MPEG2 Statistics

Table A.1: **Total number of variable-length symbols (DCT coefficients and *end-of-block*), blocks, macroblocks, slices, and pictures/frames for several MPEG-2 conformance bit-strings.**

Statistics	Scene	**batman**	**popplen**	**sarnoff2**	**tennis**	**ti1cheer**
Variable-length symbols (DCT coefficients and *end-of-block*)	I (B14)	0	0	80,563	12,345	0
	I (B15)	172,745	47,003	0	120,754	80,818
	NI (P & B)	266,485	28,069	36,408	137,756	51,680
Blocks	I	0	3,960	8,100	9,504	7,920
	P	36,943	3,606	5,120	17,992	4,481
	B	25,295	1,145	3,645	10,250	5,275
	Total	62,238	8,711	16,865	37,746	17,676
Macroblocks with blocks	I	0	660	1,350	1,584	1,320
	P	6,473	970	1,258	4,457	1,049
	B	4,847	338	1,468	4,450	1,722
	Total	11,320	1,968	4,076	10,491	4,091
Coded macroblocks	I	0	660	1,350	1,584	1,320
	P	6,480	1,980	1,332	4,625	1,106
	B	4,860	1,247	2,671	6,305	2,045
	Total	11,340	3,887	5,353	12,514	4,471
Slices	IPB	252	90	120	288	120
Pictures (Frames)	IPB	7 (7+0)	3 (0+6)	4 (4+0)	8 (8+0)	4 (2+4)

Table A.2: **MPEG-2 statistics for several conformance bit-strings.**

Scene Statistics			batman	popplen	sarnoff2	tennis	ti1cheer	Average
Blocks/macroblock	I	μ	–	6.0	6.0	6.0	6.0	6.0
		σ	–	0	0	0	0	
	P	μ	5.7	3.7	4.1	4.0	4.1	4.3
		σ	0.2	0.9	0.5	0.4	0.7	
	B	μ	5.2	3.1	2.4	2.2	2.8	3.1
		σ	0.3	1.4	0.4	0.5	0.7	
Blocks/slice	I	μ	–	264	270	264	264	266
		σ	–	0	0	0	0	
	P	μ	257	80	171	167	155	166
		σ	9	33	26	27	56	
	B	μ	234	38	61	71	88	98
		σ	14	27	23	32	45	
Blocks/{picture,frame}	I	μ	–	3960	8100	9504	7920	n/a
		σ	–	0	0	0	0	
	P	μ	9236	1202	5120	5997	4481	n/a
		σ	236	198	0	221	0	
	B	μ	8432	573	1823	2563	1324	n/a
		σ	258	93	26	718	62	
Macroblocks with blocks/slice	I	μ	–	44	45	44	44	n/a
		σ	–	0	0	0	0	
	P	μ	45	22	42	41	36	n/a
		σ	0	8	3	5	11	
	B	μ	45	11	24	31	29	n/a
		σ	0	5	7	10	11	
Coded macroblocks/slice	I	μ	–	44	45	44	44	n/a
		σ	–	0	0	0	0	
	P	μ	45	44	44	43	38	n/a
		σ	0	0	1	3	10	
	B	μ	45	42	45	44	34	n/a
		σ	0	5	1	1	11	
Slices/{picture,frame}	I	μ	–	15	30	36	30	n/a
	P	μ	36	15	30	36	30	n/a
	B	μ	36	15	30	36	15	n/a

Bibliography

[1] ***. IEEE Standard Specifications for the Implementations of 8×8 Inverse Discrete Cosine Transform. IEEE Std 1180-1990, March 1991.

[2] ***. *TM-1000 Data Book*. Philips Electronics North America Corporation, TriMedia Product Group, 811 E. Arques Avenue, Sunnyvale, California 94088, U.S.A., 1998.

[3] ***. *Book 2 – Cookbook. Part D: Optimizing TriMedia Applications*. TriMedia Technologies, Inc., TriMedia Technologies, Inc., 1840 McCarthy Boulevard, Milpitas, California 95035, U.S.A., 2000.

[4] ***. *Book 4 – Software Tools. Part A: C Language Users Guide*. TriMedia Technologies, Inc., TriMedia Technologies, Inc., 1840 McCarthy Boulevard, Milpitas, California 95035, U.S.A., 2000.

[5] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami. A DAG-Based Design Approach for Reconfigurable VLIW Processors. In *IEEE Design and Test Conference in Europe*, IEEE Computer Society Press, pages 778–780, Munich, Germany, March 1999.

[6] G.M. Amdahl, G.A. Blaauw, and F.P. Brooks, Jr. Architecture of the IBM System/360. In *IBM Journal of Research and Development*, 8(2):87–101, 1964.

[7] P.M. Athanas and H.F. Silverman. Processor Reconfiguration through Instruction-Set Metamorphosis. In *IEEE Computer*, 26(3):11–18, March 1993.

[8] J. Babb, R. Tessier, and A. Agarwal. Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulators. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM '93)*, IEEE Computer Society Press, pages 142–151, Napa Valley, California, April 1993.

[9] N.B. Bhat and K. Chaudhary. Field Programmable Logic Device with Dynamic Interconnections to a Dynamic Logic Core. U.S. Patent No. 5,596,743, January 1997.

[10] R.A. Bittner, Jr. and P.M. Athanas. Wormhole Run-time Reconfiguration. In *5th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FCCM '97)*, ACM Press, pages 79–85, Monterey, California, February 1997.

[11] M. Bolotski, A. DeHon, and T. Knight, Jr. Unifying FPGAs and SIMD Arrays. In *Second International ACM/SIGDA Workshop on FPGAs*, ACM Press, pages 1–10, Berkeley, California, February 1994. ACM.

[12] F. Bonomini, F. De Marco-Zompit, G.A. Milan, A. Odorico, and D. Palumbo. Implementing an MPEG2 Video Decoder Based on the TMS320C80 MVP. Application Report SPRA332, Texas Instruments, Paris, France, September 1996.

[13] G. Brebner. Field-Programmable Logic: Catalyst for New Computing Paradigms. In *8th International Workshop on Field-Programmable Logic and Applications (FPL '98). From FPGAs to Computing Paradigm*, Springer-Verlag, *Lecture Notes in Computer Science (LNCS)*, Vol. 1482, pages 49–58, Tallin, Estonia, September 1998.

[14] S. Brown and J. Rose. Architecture of FPGAs and CPLDs: A Tutorial. In *IEEE Transactions on Design and Test of Computers*, 13(2):42–57, 1996.

[15] D.A. Buell and K.L. Pocek. Custom Computing Machines: An Introduction. In *Journal of Supercomputing*, 9(3):219–230, 1995.

[16] S. Cadambi, J. Weener, S.C. Goldstein, H. Schmit, and D.E. Thomas. Managing Pipeline-Reconfigurable FPGAs. In *6th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '98)*, ACM Press, pages 55–64, Monterey, California, February 1998.

[17] S.M. Casselman. Virtual Computing and the Virtual Computer. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM '93)*, IEEE Computer Society Press, pages 43–48, Napa Valley, California, April 1993.

[18] S.-F. Chang and D.G. Messerschmitt. Designing High-Throughput VLC Decoder. Part I – Concurrent VLSI Architectures. In *IEEE Transactions on Circuits and Systems for Video Technology*, 2(2):187–196, June 1992.

[19] K. Chaudhary, H. Verma, and S. Nag. An Inverse Discrete Cosine Transform (IDCT) Implementation in Virtex for MPEG Video Application. Application Note 208, Xilinx Corporation, San Jose, California, December 1999.

[20] Computer Engineering Laboratory, Electrical Engineering Department, Delft University of Technology, Delft, The Netherlands. The MOLEN Project. **`http://ce.et.tudelft.nl/MOLEN/`**.

[21] Altera Corporation. Configuring APEX 20K, FLEX 10K & FLEX 6000 Devices. Application Note 116, San Jose, California, December 1999.

[22] Altera Corporation. ACEX 1K Programmable Logic Family. Datasheet, San Jose, California, April 2000.

[23] Atmel Corporation. AT6000 Series Configuration. Application Note, San Jose, California, September 1999.

[24] Atmel Corporation. AT6000(LV) Series. Coprocessor Field Programmable Gate Arrays. Datasheet, San Jose, California, October 1999.

[25] Xilinx Corporation. XC6200 Field Programmable Gate Arrays. Datasheet, San Jose, California, October 1996.

[26] D.C. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling. Architecture Design of Reconfigurable Pipelined Datapaths. In *Advanced Research in VLSI*, pages 23–40, 1999.

[27] A. DeHon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In *2nd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM '94)*, pages 31–39, Napa Valley, California, April 1994.

[28] A. DeHon. Reconfigurable Architectures for General-Purpose Computing. A. I. 1586, Massachusetts Institute of Technology, Cambridge, Massachusetts, October 1996.

[29] A. DeHon, T. Knight Jr., E. Tau, M. Bolotski, I. Eslick, D. Chen, and J. Brown. Dynamically Programmable Gate Array with Multiple Context. U.S. Patent No. 5,742,180, April 1998.

[30] A. DeHon. The Density Advantage of Configurable Computing. In *IEEE Computer*, 33(4):41–49, April 2000.

[31] A. Donlin. Self Modifying Circuitry - A Platform for Tractable Virtual Circuitry. In *8th International Workshop on Field-Programmable Logic and*

Applications (FPL '98). From FPGAs to Computing Paradigm, Springer-Verlag, *Lecture Notes in Computer Science (LNCS)*, Vol. 1482, pages 199–208, Tallin, Estonia, September 1998.

[32] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A Multiprocessor SOC for Advanced Set-Top Box and Digital TV Systems. In *IEEE Design & Test of Computers*, 18(5):21–31, September-October 2001.

[33] C. Ebeling, D.C. Cronquist, and P. Franklin. RaPiD – Reconfigurable Pipelined Datapath. In *6th International Workshop on Field Programmable Logic and Applications (FPL '96). Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*, Springer-Verlag, *Lecture Notes in Computer Science (LNCS)*, Vol. 1142, pages 126–135, Darmstadt, Germany, September 1996.

[34] M.J. Flynn. Some Computer Organizations and Their Effectiveness. In *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[35] T. Garverick, J. Sutherland, S. Popli, V. Alturi, A. Smith, Jr., S. Pickett, D. Hawley, S.-P. Chen, S. Moni, B.S. Ting, R.C. Camarota, S.-M. Day, and F. Furtek. Versatile and Efficient Cell-to-Local Bus Interface in a Configurable Logic Array. U.S. Patent No. 5,298,805, March 1994.

[36] K.L. Gilson. Integrated Circuit Computing Device Comprising a Dynamically Configurable Gate Array Having a Microprocessor and Reconfigurable Instruction Execution Means and Method Therefor. U.S. Patent No. 5,361,373, November 1994.

[37] S.C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Reed Taylor, and R. Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *The 26th International Symposium on Computer Architecture*, pages 28–39, Atlanta, Georgia, May 1999.

[38] J. P. Gray and T. A. Kean. Configurable Hardware: A New Paradigm for Computation. In *Proceedings of the Decennial Caltech Conference*, pages 279–295, Pasadena, California, March 1989.

[39] S.A. Guccione and M.J. Gonzales. Classification and Performance of Reconfigurable Architectures. In *5th International Workshop on Field-Programmable Logic and Applications (FPL '95)*, Springer-Verlag, pages 439–448, Oxford, United Kingdom, August-September 1995.

[40] R. Haerkens and J. Sannen. Variable-Length Decoding in Software – A General Purpose Video Decoding Coprocessor. SwTV 176, Philips Research Laboratories, Eindhoven, The Netherlands, December 1997.

[41] R.W. Hartenstein, J. Becker, and R. Kress. Custom Computing Machines versus Hardware/Software Co-Design: From a Globalized Point of View. In *6th International Workshop on Field Programmable Logic and Applications (FPL '96). Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*, Springer-Verlag, *Lecture Notes in Computer Science (LNCS),* Vol. 1142, pages 65–76, Darmstadt, Germany, September 1996.

[42] R.W. Hartenstein, A.G. Hirschbiel, K. Schmidt, and M. Weber. A Novel Paradigm of Parallel Computation and its Use to Implement Simple High-Performance Hardware. In *Future Generation Computer Systems*, (7):181–198, 1991/1992.

[43] R.W. Hartenstein, R. Kress, and H. Reinig. A New FPGA Architecture for Word-Oriented Datapaths. In *4th International Workshop on Field-Programmable Logic and Applications (FPL '94). Field-Programmable Logic: Architectures, Synthesis and Applications*, Springer-Verlag, *Lecture Notes in Computer Science (LNCS)*, Vol. 849, pages 144–155, Prague, Czech Republic, September 1994.

[44] B.G. Haskell, A. Puri, and A.N. Netravali. *Digital Video: An Introduction to MPEG-2*. Kluwer Academic Publishers, Norwell, Massachusetts, 1996.

[45] S.A. Hauck. The Future of Reconfigurable Systems. In *Proceedings of the 5th Canadian Conference on Field Programmable Devices*, Montreal, Canada, June 1998.

[46] S.A. Hauck. The Roles of FPGA's in Reprogrammable Systems. In *Proceedings of the IEEE*, 86(4):615–638, April 1998.

[47] S.A. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao. The Chimaera Reconfigurable Functional Unit. In *5th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pages 87–96, Napa Valley, California, April 1997.

[48] J.R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *5th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pages 12–21, Napa Valley, California, April 1997.

[49] G.J. Hekstra, G.D. La Hei, P. Bingley, and F.W. Sijstermans. TriMedia CPU64 Design Space Exploration. In *Proceedings of International Conference on Computer Design*, pages 599–606, Austin, Texas, October 1999.

[50] J. Hoogerbrugge and L. Augusteijn. Instruction Scheduling for TriMedia. In *Journal of Instruction-Level Parallelism,* 1(1), February 1999.

[51] International Organization for Standardization. Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s – Part 2: Video. ISO/IEC 11172-2: 1993.

[52] International Telecommunication Unit. Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios. ITU-R Recommendation BT.601-5, October 1995.

[53] International Telecommunication Unit. Information technology – Generic coding of moving pictures and associated audio information: Video. ITU-T Recommendation H.262, February 2000.

[54] C. Iseli and E. Sanchez. A Superscalar and Reconfigurable Processor. In *4th International Workshop on Field-Programmable Logic and Applications (FPL '94). Field-Programmable Logic: Architectures, Synthesis and Applications*, Springer-Verlag, *Lecture Notes in Computer Science (LNCS),* Vol. 849, pages 168–174, Prague, Czech Republic, September 1994.

[55] D. Ishii, M. Ikekawa, and I. Kuroda. Parallel Variable Length Decoding with Inverse Quantization for Software MPEG-2 Decoders. In *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS97)*, pages 500–509, Leicester, United Kingdom, November 1997. IEEE.

[56] J.A. Jacob and P. Chow. Memory Interfacing and Instruction Specification for Reconfigurable Processors. In *7th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '99)*, pages 145–154, Monterey, California, February 1999.

[57] W.M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[58] D. Jones and D.M. Lewis. A Time-Multiplexed FPGA Architecture for Logic Emulation. In *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*, pages 487–494, Santa Clara, California, May 1995.

[59] B. Kastrup, A. Bink, and J. Hoogerbrugge. ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator. In *7th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '99)*, pages 92–100, Napa Valley, California, April 1999.

[60] B. Kastrup, J. van Meerbergen, and K. Nowak. Seeking (the right) Problems for the Solutions of Reconfigurable Computing. In *9th International Workshop on Field-Programmable Logic and Applications (FPL '99)*, Springer-Verlag, *Lecture Notes in Computer Science (LNCS)*, Vol. 1673, pages 520–525, Glasgow, Scotland, September 1999.

[61] S. Kinouchi and A. Sawada. Huffman Code Decoding Circuit. U.S. Patent No. 5,617,089, April 1997.

[62] G. Kuzmanov and S. Vassiliadis. Arbitrating Instructions in an $\rho\mu$–coded CCM. In *13th International Conference on Field-Programmable Logic and Applications (FPL 2003). Reconfigurable Machines: a New Paradigm of Computing*, Springer-Verlag., *Lecture Notes in Computer Science (LNCS)*, Lisbon, Portugal, September 2003 (accepted for publication).

[63] S.-M. Lei and M.-T. Sun. An Entropy Coding System for Digital HDTV Applications. In *IEEE Transactions on Circuits and Systems for Video Technology*, 1(1):147–155, March 1991.

[64] A. Lew and R. Halverson, Jr. A FCCM for Dataflow (Spreadsheet) Programs. In *3rd IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '95)*, pages 2–10, Napa Valley, California, April 1995.

[65] H.-D. Lin and D.G. Messerschmitt. Finite State Machine has Unlimited Concurrency. In *IEEE Transactions on Circuits and Systems*, 38(5):465–475, May 1991.

[66] H.-D. Lin and D.G. Messerschmitt. Designing a High-Throughput VLC Decoder. Part II – Parallel Decoding Methods. In *IEEE Transactions on Circuits and Systems for Video Technology*, 2(2):197–206, June 1992.

[67] C. Loeffler, A. Ligtenberg, and G.S. Moschytz. Practical Fast 1-D DCT Algorithms with 11 Multiplications. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing – ICASSP '89*, pages 988–991, 1989.

[68] W.H. Mangione-Smith and B.L. Hutchings. Reconfigurable Architectures: The Road Ahead. In *Reconfigurable Architectures Workshop, RAW '97*, pages 81–96, Geneva, Switzerland, April 1997.

[69] W.H. Mangione-Smith, B.L. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R.W. Hartenstein, O. Mencer, J. Morris, K. Palem, V.K. Prasanna, and H.A.E. Spaanenburg. Seeking Solutions in Configurable Computing. In *IEEE Computer*, 30(12):38–43, December 1997.

[70] J.L. Mitchell, W.B. Pennebaker, C.E. Fogg, and D.J. LeGall. *MPEG Video Compression Standard*. Chapman & Hall, New York, New York, 1996.

[71] T. Miyamori and K. Olukotun. A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications. In *6th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pages 2–11, Napa Valley, California, April 1998.

[72] T. Miyamori and K. Olukotun. REMARC: Reconfigurable Multimedia Array Coprocessor. In *IEEE Transactions on Information and Systems*, E82-D(2):389–397, February 1999.

[73] E. Moscu-Panainte, K. Bertels, and S. Vassiliadis. Compiling for the Molen Programming Paradigm. In *13th International Conference on Field-Programmable Logic and Applications (FPL 2003). Reconfigurable Machines: a New Paradigm of Computing*, Springer-Verlag, *Lecture Notes in Computer Science (LNCS)*, Lisbon, Portugal, September 2003 (accepted for publication).

[74] MPEG Software Simulation Group. MPEG-2 Video Codec. http://www.mpeg.org/MPEG/MSSG/

[75] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, New York, New York, 2000.

[76] Y.-G. Park. High Speed Variable Length Code Decoding Apparatus. U.S. Patent No. 5,561,690, October 1996.

[77] K. Patel, B.C. Smith, and L.A. Rowe. Performance of a Software MPEG Video Decoder. In *Proceedings of the ACM Multimedia 93 Conference*, pages 75–82, Anaheim, California, 1993.

[78] G.G. Pechanek, C.W. Kurak, C.J. Glossner, C.H.L. Moller, and S.J. Walsh. M.F.A.S.T.: A Highly Parallel Single Chip DSP with a 2D IDCT Example. In *Proceeding of the International Conference on Signal Processing Applications and Technology*, pages 69–72, Boston, Massachusetts, October 1995.

[79] G.G. Pechanek and S. Vassiliadis. The ManArray Embedded Processor Architecture. In *Proceedings of the 26th Euromicro Conference: "Informatics: inventing the future"*, pages 348–355, Maastricht, The Netherlands, September 2000. IEEE.

[80] E.-J.D. Pol, B.J.M. Aarts, Jos T.J. van Eijndhoven, P. Struik, F.W. Sijstermans, M.J.A. Tromp, J.W. van de Waerdt, and P. van der Wolf. TriMedia CPU64 Application Development Environment. In *Proceedings of International Conference on Computer Design*, pages 593–598, Austin, Texas, October 1999.

[81] E.-J.D. Pol, Jos T.J. van Eijndhoven, K.A. Vissers, and B. Riemens. PROMMPT-2 Project Plan. SwTV 001, Philips Research Laboratories, Eindhoven, The Netherlands, September 1999.

[82] E.-J.D. Pol. VLD Performance on TriMedia/CPU64. Private Communication, May 2000.

[83] C. Poynton. *A Technical Introduction to Digital Video*. John Wiley & Sons, January 1996.

[84] B. Radunović and V. Milutinović. A Survey of Reconfigurable Computing Architectures. In *8th International Workshop on Field-Programmable Logic and Applications (FPL '98). From FPGAs to Computing Paradigm*, Springer-Verlag, *Lecture Notes in Computer Science (LNCS)*, Vol. 1482, pages 376–385, Tallin, Estonia, September 1998.

[85] K.R. Rao and P. Yip. *Discrete Cosine Transform. Algorithms, Advantages, Applications*. Academic Press, San Diego, California, 1990.

[86] T.G. Rauscher and P.M. Adams. Microprogramming: A Tutorial and Survey of Recent Developments. In *IEEE Transactions on Computers*, C-29(1):2–20, January 1980.

[87] R. Razdan and M.D. Smith. A High Performance Microarchitecture with Hardware-Programmable Functional Units. In *27th Annual International Symposium on Microarchitecture – MICRO-27*, pages 172–180, San Jose, California, November 1994.

[88] A.K. Riemens, K.A. Vissers, R.J. Schutten, F.W. Sijstermans, G.J. Hekstra, and G.D. La Hei. TriMedia CPU64 Application Domain and Benchmark Suite. In *Proceedings of International Conference on Computer Design*, pages 580–585, Austin, Texas, October 1999.

[89] C.R. Rupp. *CLAyFun Reference Manual*. National Semiconductor Corp., Santa Clara, California, July 1995.

[90] C.R. Rupp, M. Landguth, T. Garverick, E. Gomersall, and H. Holt. The NAPA Adaptive Processing Architecture. In *6th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pages 28–37, Napa Valley, California, April 1998.

[91] Z. Salcic and B. Maunder. CCSimP – An Instruction-Level Custom-Configurable Processor for FPLDs. In *6th International Workshop on Field Programmable Logic and Applications (FPL '96). Field-Programmable Logic: Smart Applications, New Paradigms and Compilers*, Springer-Verlag, *Lecture Notes in Computer Science (LNCS)*, Vol. 1142, pages 280–289, Darmstadt, Germany, September 1996.

[92] S. Sawitzki, A. Gratz, and R.G. Spallek. Increasing Microprocessor Performance with Tightly-Coupled Reconfigurable Logic Arrays. In *8th International Workshop on Field-Programmable Logic and Applications (FPL '98). From FPGAs to Computing Paradigm, Lecture Notes in Computer Science (LNCS)*, Vol. 1482, pages 411–415, Tallin, Estonia, September 1998.

[93] S.M. Scalera and J.R. Vázquez. The Design and Implementation of a Context Switching FPGA. In *6th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pages 78–85, Napa Valley, California, April 1998.

[94] M. Sima. Color Space Conversion on TriMedia/CPU64. Private Communication, August 2002.

[95] M. Sima, S.D. Cotofana, Jos T.J. van Eijndhoven, S. Vassiliadis, and K.A. Vissers. 8×8 IDCT Implementation on an FPGA-augmented TriMedia. In *9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2001)*, Rohnert Park, California, April 2001.

[96] M. Sima, S.D. Cotofana, S. Vassiliadis, Jos T.J. van Eijndhoven, and K.A. Vissers. MPEG Macroblock Parsing and Pel Reconstruction on an FPGA-augmented TriMedia Processor. In *IEEE International Conference on Computer Design*, pages 425–430, Austin, Texas, September 2001.

[97] M. Sima, S.D. Cotofana, S. Vassiliadis, and Jos T.J. van Eijndhoven. Variable Length Decoder Implemented on a TriMedia/CPU64 Reconfigurable Functional Unit. In *12th Annual Workshop on Circuits, Systems and Signal*

Processing (ProRISC 2001), pages 211–218, Veldhoven, The Netherlands, November 2001.

[98] M. Sima, S.D. Cotofana, S. Vassiliadis, J.T.J. van Eijndhoven, and K.A. Vissers. MPEG-compliant Entropy Decoding on FPGA-augmented TriMedia/CPU64. In *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2002)*, pages 261–270, Napa Valley, California, April 2002.

[99] M. Sima, E.-J.D. Pol, J.T.J. van Eijndhoven, S.D. Cotofana, and S. Vassiliadis. Entropy Decoding on TriMedia/CPU64. In *2nd Workshop on System Architecture MOdeling and Simulation (SAMOS 2002)*, STW Press, Samos, Greece, July 2002.

[100] M. Sima, S. Vassiliadis, S.D. Cotofana, J.T.J. van Eijndhoven, and K.A. Vissers. Field-Programmable Custom Computing Machines. A Taxonomy. In *12th International Conference on Field-Programmable Logic and Applications (FPL 2002)*, Springer-Verlag, *Lecture Notes in Computer Science (LNCS)*, Vol. 2438, pages 79–88, Montpellier, France, September 2002.

[101] H. Singh, M.-H. Lee, G. Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Application. In *IEEE Transactions on Computers*, 49(5):465–481, May 2000.

[102] M.-T. Sun. VLSI Architecture and Implementation of a High Speed Entropy Decoder. In *IEEE International Symposium on Circuits and Systems*, pages 200–203, 1991.

[103] M.-T. Sun. *VLSI Implementations for Image Communications*, volume 2, chapter Design of High-Throughput Entropy Codec, pages 345–364. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1993.

[104] M.-T. Sun and K.-H. Tzou. High-Speed Flexible Variable-Length-Code Decoder. U.S. Patent No. 5,173,695, December 1992.

[105] Lucent Technologies. ORCA Series 2 Field-Programmable Gate Arrays. Datasheet, Allentown, Pennsylvania, June 1999.

[106] Lucent Technologies. ORCA Series 3C and 3T Field-Programmable Gate Arrays. Datasheet, Allentown, Pennsylvania, June 1999.

[107] S.M. Trimberger. Microprocessor-based FPGA. International Patent Application under P.C.T., No. WO 95/04402, February 1995.

[108] S.M. Trimberger. Reprogrammable Instruction Set Accelerator. U.S. Patent No. 5,737,631, April 1998.

[109] S.M. Trimberger. Reprogrammable Instruction Set Accelerator Using a Plurality of Programmable Execution Units and an Instruction Page Table. U.S. Patent No. 5,748,979, May 1998.

[110] S.M. Trimberger, D. Carberry, A. Johnson, and J. Wong. A Time-Multiplexed FPGA. In *5th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '97)*, pages 22–28, Napa Valley, California, April 1997.

[111] P.P. Vaidyanathan. *Multirate Systems and Filter Banks*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[112] J.T.J. van Eijndhoven. 16-bit compliant software IDCT on TriMedia/CPU64. Internal Report NL-TN 171, Philips Research Laboratories, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands, December 1997.

[113] J.T.J. van Eijndhoven, F.W. Sijstermans, K.A. Vissers, E.-J.D. Pol, M.J.A. Tromp, P. Struik, R.H.J. Bloks, P. van der Wolf, A.D. Pimentel, and H.P.E. Vranken. TriMedia CPU64 Architecture. In *Proceedings of International Conference on Computer Design*, pages 586–592, Austin, Texas, October 1999.

[114] J.T.J. van Eijndhoven and F.W. Sijstermans. Data Processing Device and method of Computing the Cosine Transform of a Matrix. U.S. Patent No. 6,397,235, March 2002.

[115] J.T.J. van Eijndvhoven, G.A. Slavenburg, and S. Rathnam. VLIW Processor has different functional units operating on commands of different widths. U.S. Patent No. 6,076,154, June 2000.

[116] S. Vassiliadis, B. Juurlink, and E. Hakkennes. Complex Streamed Instructions: Introduction and Initial Evaluation. In *Informatics: inventing the future. Proceedings of the 26-th Euromicro Conference (EUROMICRO '00)*, volume 1, pages 400–408, Maastricht, The Netherlands, September 2000.

[117] S. Vassiliadis, S. Wong, and S.D. Cotofana. The MOLEN $\rho\mu$-coded Processor. In *11th International Conference on Field-Programmable Logic and Applications (FPL 2001)*, Springer-Verlag, *Lecture Notes in Computer Science (LNCS)*, Vol. 2147, pages 275–285, Belfast, Northern Ireland, United Kingdom, August 2001.

[118] S. Vassiliadis, S. Wong, and S.D. Cotofana. Microcode Processing: Positioning and Directions. In *IEEE Micro*, 23(4):21–30, July 2003.

In *11th International Conference on Field-Programmable Logic and Applications (FPL 2001)*, Springer-Verlag, *Lecture Notes in Computer Science (LNCS)*, Vol. 2147, pages 275–285, Belfast, Northern Ireland, United Kingdom, August 2001.

[119] J. Villasenor and W.H. Mangione-Smith. Configurable Computing. *Scientific American*, pages 55–59, June 1997. http://www.sciam.com/0697issue/0697villasenor.html.

[120] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II Compiler and Architecture. In *IEEE Workshop on FPGAs for Custom Computing Machines (FCCM '93)*, pages 9–16, Napa Valley, California, April 1993.

[121] M. Edward Wazlowski. *A Reconfigurable Architecture Superscalar Coprocesor*. PhD thesis, Brown University, Providence, Rhode Island, May 1996.

[122] M.J. Wirthlin and B.L. Hutchings. A Dynamic Instruction Set Computer. In *3rd IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '95)*, pages 99–109, Napa Valley, California, April 1995.

[123] M.J. Wirthlin, B.L. Hutchings, and K.L. Gilson. The Nano Processor: A Low Resource Reconfigurable Processor. In *2nd IEEE Workshop on FPGAs for Custom Computing Machines (FCCM '94)*, pages 23–30, Napa Valley, California, April 1994.

[124] R.D. Wittig and P. Chow. OneChip: An FPGA Processor With Reconfigurable Logic. In *4th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '96)*, pages 126–135, Napa Valley, California, April 1996.

Samenvatting

In dit verslag presenteren we een uitbreiding van de TriMedia–CPU64™ VLIW processor met een Field-Programmable Gate Array (FPGA) en stellen we de potentiële prestatie vast van zulke hybriden voor media georiënteerde taken. De FPGA is aangesloten op de TriMedia–CPU64 net als elk ander functioneel apparaat, omdat alleen kleine veranderingen van de processor organisatie toegestaan zijn. De resulterende kruising wordt hierna gerefereerd als ρ–TriMedia. We beschrijven eerst een uitbreiding van de TriMedia–CPU64 instructie set architectuur, die ondersteuning bevat voor de FPGA. In essentie wordt er een kernel van nieuwe instructies gegeven, zijnde SET en EXECUTE. De SET instructie regelt de herconfiguratie van de FPGA en de EXECUTE instructie start de operaties op die door de FPGA-mapped rekenkundige blokken worden uitgevoerd. De aanpak is algemeen, waardoor de gebruiker de vrijheid heeft om elk aangepast rekenkundig blok te definiëren en te gebruiken. Bovendien kan een groot aantal herconfigureerbare operaties gecodeerd worden, terwijl er maar één plaats voor de EXECUTE instructie toegewezen moet worden in de opcode ruimte, als men de opcode velden gebruikt van aangrenzende VLIW instructie sloten om een argument te definieren voor de EXECUTE opcode. Op deze manier creëert de herconfigureerbare operatie geen druk op de instructie decoder, past het netjes in het bestaande instructieformaat, past de bestaande verbindingsstructuur met het register file en heeft daardoor maar weinig hardware controle nodig. Om dan de potentiële prestaties van de ρ–TriMedia vast te kunnen stellen, richten we ons op de MPEG standaard en gebruiken we een aantal media kernels die veel berekeningen vereisen: Inverse Discrete Cosine Transform, Inverse Quantization, Entropy Decoder en YCC-naar-RGB kleurenspectrum omzetter. Voor elke kernel is er FPGA-gebaseerd rekenkundig blok ontworpen. Als herconfigureerbare kern werd de ACEX™ EP1K100 FPGA van Altera gebruikt. De experimenten, die uitgevoerd werden op een cyclus accurate TriMedia–CPU64 simulator, duiden erop dat een versnelling van meer dan 40% met de ρ–TriMedia bereikt kan worden bij een aantal objecten volgens de MPEG2 standaard. Uit het feit dat de TriMedia–

CPU64 een 5 instructie slot VLIW processor is met een 64-bit datapad met een grote instructie set, georiënteerd op media operaties, kan geconcludeerd worden dat zo’n verbetering binnen zijn doel van het domein van media berekeningen, met een relatief klein FPGA, duidt op een veelbelovende aanpak van FPGA-uitgebreide TriMedia–CPU64 (ρ–TriMedia).

Rezumat în limba română

Lucrarea intitulată **„Procesorul reconfigurabil TriMedia”** (ρ–TriMedia) se încadrează în domeniul procesoarelor VLIW care au setul de instrucţiuni optimizat pentru prelucrari multimedia. În particular, am prezentat o modalitate de a ataşa o structură reconfigurabilă la procesorul TriMedia–CPU64 dezvoltat de firma olandeză Philips şi am evaluat câştigul în viteză de calcul care se obţine. Ideea fundamentală în mărirea performanţei procesorului gazdă este de a defini resurse de calcul ce sunt optimizate pentru aplicaţia considerată şi de a le configura pe structura reconfigurabilă. Astfel, operaţii complexe sunt executate în hardware (reconfigurabil) în loc de software. Execuţia codului pe procesorul extins cuprinde două faze: `SET`, în cadrul căreia se realizează configurarea structurii reconfigurabile, şi `EXECUTE`, când se lansează în execuţie unităţile de calcul ce au fost configurate în faza anterioară. Întrucât abordarea este generică, aplicaţii dintre cele mai diverse pot utiliza resursele de calcul menţionate. Ca studiu de caz, am ales o etapă importantă din decodarea MPEG şi anume reconstrucţia imaginii codate până la nivel de pixel. Alegerea nu este întâmplătoare, întrucât gama de prelucrări necesare reconstrucţii imaginii se regăseşte şi în alte standarde înrudite cu MPEG, cum ar fi, de exemplu, JPEG sau MJPEG. În particular, patru nuclee de funcţii multimedia au fost analizate şi anume: transformata cosinus discretă inversă, cuantizare inversă, decodare entropică şi conversie de culoare din spaţiul YCC către spaţiul RGB. Pentru fiecare nucleu în parte, a fost proiectată o resursă de calcul optimizată pentru configurare pe FPGA. Drept structură reconfigurabilă, am utilizat o arie de porţi programabilă de capacitate relativ redusă şi anume ACEX EP1K100 dezvoltată de firma Altera. Rezultatele experimentale arată o mărire a vitezei de calcul a procesorului extins de peste 40% în raport cu viteza procesorului de bază. Întrucât TriMedia–CPU64 este un procesor VLIW pe 64 de biţi, care are cinci sloturi pentru lansarea în paralel a operaţiilor, prezentând în acelaşi timp un foarte bogat set de instrucţiuni care este, în plus, optimizat pentru prelucrări multimedia, îmbunătăţirea de performanţă manţionată demonstrează că procesorul extins (ρ–TriMedia) reprezintă o structură viabilă.

Περίληψη στα Ελληνικά

Στην παρούσα διατριβή παρουσιάζουμε μια επαύξηση του επεξεργαστή TriMedia–CPU64 VLIW με μία Διάταξη Πυλών Επαναδιατασσόμενη Λογικής (FPGA) και αξιολογούμε τις δυνητικές επιδόσεις του υβριδίου αυτού κατά την εκτέλεση εργασιών με προσανατολισμό στα πολυμέσα. Επειδή επιτρέπονται μόνο ελάχιστες τροποποιήσεις της οργάνωσης του επεξεργαστή, η FPGA συνδέεται στον TriMedia–CPU64 VLIW όπως και κάθε άλλη μόνιμα συνδεδεμένη λειτουργική μονάδα. Το υβρίδιο που προκύπτει αναφέρεται ως ρ–TriMedia. Περιγράφουμε πρώτα μια επέκταση της αρχιτεκτονικής του σετ εντολών του TriMedia, που ενσωματώνει υποστήριξη για την FPGA. Κατά βάση, παρέχεται ένας πυρήνας νέων εντολών με την ονομασία `SET` και `EXECUTE`. Η εντολή `SET` ελέγχει τον επαναπρογραμματισμό της FPGA, και η εντολή `EXECUTE` εκτελεί τις λειτουργίες που πραγματοποιούν οι χαρτογραφημένες από την FPGA υπολογιστικές μονάδες. Η προσέγγιση είναι γενική, συνεπώς παρέχεται στο χρήστη η ελευθερία να καθορίζει και να χρησιμοποιεί οποιαδήποτε προσαρμοσμένη υπολογιστική μονάδα. Επιπλέον, με χρήση των πεδίων των κωδικών εντολών σε γειτονικούς καταχωρητές εξόδου VLIW, για τον καθορισμό ενός ορίσματος του κώδικα εντολών της `EXECUTE`, είναι δυνατή η κωδικοποίηση μεγάλου αριθμού επαναπρογραμματιζόμενων λειτουργιών με μια μόνο απλή καταχώριση για την εντολή `EXECUTE` στο πεδίο των κωδικών εντολών. Με τον τρόπο αυτό, η επαναπρογραμματιζόμενη λειτουργία δεν επιβαρύνει τον αποκωδικοποιητή εντολών, είναι απόλυτα συμβατή με την υπάρχουσα μορφή εντολών καθώς και την υπάρχουσα δομή διασύνδεσης με το αρχείο καταχωρητών και ως εκ τούτου απαιτεί ελάχιστη κατασκευαστική επιβάρυνση. Για την αξιολόγηση των δυνητικών επιδόσεων του ρ–TriMedia, εστιάζουμε στο πρότυπο MPEG και εξετάζουμε έναν αριθμό πυρήνων πολυμέσων υψηλών υπολογιστικών απαιτήσεων: Αντίστροφος Διάκριτος Συνημιτονοειδής Μετασχηματισμός (Inverse Discrete Cosine Transform), Αντίστροφος Κβαντισμός (Inverse Quantization), Εντροπική Αποκωδικοποίηση (Entropy Decoding) και Μετατροπή Χρωμάτων YCC σε

RGB (YCC-to-RGB Color Space Conversion). Για κάθε πυρήνα σχεδιάζεται μια υπολογιστική μονάδα βασισμένη στην FPGA. Ως επαναπρογραματιζόμενος πυρήνας χρησιμοποιείται η ACEX EP1K100 FPGA της Altera. Τα πειράματα που έγιναν σε έναν προσομοιωτή ακριβούς κύκλου (cycle-accurate simulator) TriMedia–CPU64 καταδεικνύουν ότι στον ρ–TriMedia επιτυγχάνεται ταχύτητα 40% μεγαλύτερη σε σχέση με τον τυπικό TriMedia–CPU64 για μια σειρά σκηνών που συμμορφώνονται με το MPEG2. Δεδομένου ότι ο TriMedia–CPU64 είναι ένας επεξεργαστής VLIW 5 εξόδων με δίαυλο δεδομένων 64-bit και διαθέτει ένα ιδιαιτέρως πλούσιο σετ εντολών με προσανατολισμό στα πολυμέσα, μία τέτοια βελτίωση στον τομέα επεξεργασίας πολυμέσων, που επιτυγχάνεται με μία σχετικά μικρή FPGA, καθιστά τον επαυξημένο με FPGA TriMedia–CPU64 (ρ–TriMedia) ως μία πολλά υποσχόμενη προσέγγιση.

List of Publications

Journal Papers

1. **Mihai Sima**, Sorin Cotofana, Stamatis Vassiliadis, Jos T.J. van Eijndhoven, and Kees Vissers, "Pel Reconstruction on FPGA-augmented TriMedia," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, in press.

2. **Mihai Sima**, Sorin Cotofana, Jos T.J. van Eijndhoven, Stamatis Vassiliadis, and Kees Vissers, "IEEE-compliant IDCT on FPGA-augmented TriMedia," in *Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, Kluwer Academics, in press.

International Conferences

3. **Mihai Sima**, Stamatis Vassiliadis, Sorin Cotofana, Jos T.J. van Eijndhoven, "Color Space Conversion for MPEG Decoding on FPGA-augmented TriMedia Processor," in *The 14th IEEE International Conference on Application-specific Systems, Architectures, and Processors (ASAP 2003)*, The Hague, The Netherlands, June 2003, pp. 250-259.

4. **Mihai Sima**, Stamatis Vassiliadis, Sorin Cotofana, Jos T.J. van Eijndhoven, and Kees Vissers, "Field-Programmable Custom Computing Machines. A Taxonomy," in *International Workshop on Field-Programmable Logic and Applications (FPL 2002)*, Montpellier, France, September 2002, Springer-Verlag, *Lecture Notes in Computer Science*, Vol. 2438, pp. 79–88.

5. Jari Nikara, Stamatis Vassiliadis, Jarmo Takala, **Mihai Sima**, and Petri Liuha, "Parallel Multiple-Symbol Variable-Length Decoding," in *IEEE International Conference on Computer Design (ICCD 2002)*, Freiburg, Germany, September 2002, pp. 126–131.

6. **Mihai Sima**, Evert-Jan Pol, Jos T.J. van Eijndhoven, Sorin Cotofana, and Stamatis Vassiliadis, “Entropy Decoding on TriMedia/CPU64,” in *System Architecture MOdeling and Simulation Workshop (SAMOS 2002)*, Samos, Greece, July 2002.

7. **Mihai Sima**, Sorin Cotofana, Stamatis Vassiliadis, Jos T.J. van Eijndhoven, and Kees Vissers, “MPEG-compliant Entropy Decoding on FPGA-augmented TriMedia/CPU64,” in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2002)*, Napa Valley, California, April 2002, pp. 261–270.

8. **Mihai Sima**, Sorin Cotofana, Stamatis Vassiliadis, Jos T.J. van Eijndhoven, and Kees Vissers, “MPEG Macroblock Parsing and Pel Reconstruction on an FPGA-augmented TriMedia Processor,” in *IEEE International Conference on Computer Design (ICCD 2001)*, Austin, Texas, September 2001, pp. 425–430, **Best Paper Award**.

9. **Mihai Sima**, Sorin Cotofana, Stamatis Vassiliadis, Jos T.J. van Eijndhoven, and Kees Vissers, “A Reconfigurable Functional Unit for TriMedia/CPU64: A Case Study,” in *System Architecture MOdeling and Simulation Workshop (SAMOS 2001)*, Samos, Greece, July 2001, Springer-Verlag, *Lecture Notes in Computer Science*, Vol. 2268, pp. 224–241.

10. **Mihai Sima**, Sorin Cotofana, Jos T.J. van Eijndhoven, Stamatis Vassiliadis, and Kees Vissers, “8×8 IDCT Implementation on an FPGA-augmented TriMedia,” in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2001)*, Rohnert Park, California, April 2001.

11. Dragos Burileanu, Claudius Dan, **Mihai Sima**, and Corneliu Burileanu, “A Parser-Based Text Preprocessor for Romanian Language TTS Synthesis,” in *6th European Conference on Speech Communication and Technology (Eurospeech ’99)*, Budapest, Hungary, September 1999, pp. 2063–2066.

12. Dragos Burileanu, **Mihai Sima**, and Adrian Neagu, “A Phonetic Converter for Speech Synthesis in Romanian,” in *14th International Congress of Phonetic Sciences (ICPhS ’99)*, San Francisco, California, August 1999, pp. 503–506.

13. **Mihai Sima**, Dragos Burileanu, Corneliu Burileanu, and Victor Croitoru, “Full-Custom Software for Start/End Point Detection of Isolated-Spoken Words,” in *12th International Conference on Control System and Computer Science (CSCS12)*, Bucharest, Romania, May 1999, pp. 19–24.

14. **Mihai Sima**, Dragos Burileanu, Corneliu Burileanu, and Victor Croitoru, “Application of Neural Network Paradigms in Speech Recognition for a Romanian Voice Dialing System,” in *International Conference on Communications (Comm ’98)*, Bucharest, Romania, November 1998, pp. 233–238.

15. Dragos Burileanu, **Mihai Sima**, Corneliu Burileanu, and Victor Croitoru, “A Neural Network-Based Speaker Independent System for Word Recognition in Romanian Language,” in *Workshop on Text, Speech, Dialogue (TSD ’98)*, Brno, Czech Republic, September 1998, pp. 177–182.

Local Conferences

16. Dragos Burileanu, **Mihai Sima**, Cristian Negrescu, and Victor Croitoru, “Robust Recognition of Small-Vocabulary Telephone-Quality Speech,” in *Second Conference on Speech Technology and Human-Computer Dialogue (SpeD 2003)*, Bucharest, Romania, April 2003, pp. 145-154.

17. **Mihai Sima**, Stamatis Vassiliadis, Jos T.J. van Eijndhoven, and Sorin Cotofana, “YUV-to-RGB Color Space Conversion on FPGA-augmented TriMedia-32 Processor,” in *Workshop on Circuits, Systems and Signal Processing (ProRISC 2002)*, Veldhoven, The Netherlands, November 2002.

18. **Mihai Sima**, Sorin Cotofana, Stamatis Vassiliadis, and Jos T.J. van Eijndhoven, “Variable Length Decoder Implemented on a TriMedia/CPU64 Reconfigurable Functional Unit,” in *Workshop on Circuits, Systems and Signal Processing (ProRISC 2001)*, Veldhoven, The Netherlands, November 2001.

19. **Mihai Sima**, Sorin Cotofana, Stamatis Vassiliadis, and Jos T.J. van Eijndhoven, “An 8-Point IDCT Computing Resource Implemented on a TriMedia/CPU64 FPGA-based Reconfigurable Functional Unit,” in *Workshop on Embedded Systems (PROGRESS 2001)*, Veldhoven, The Netherlands, October 2001, pp. 211–218.

20. **Mihai Sima**, Stamatis Vassiliadis, Sorin Cotofana, Jos T.J. van Eijndhoven, and Kees Vissers, “A Taxonomy of Custom Computing Machines,” in *Workshop on Embedded Systems (PROGRESS 2000)*, Utrecht, The Netherlands, October 2000, pp. 87–93.

21. **Mihai Sima** and Victor Croitoru, “Experiments on Isolated-Spoken Word Recognition” in *Romanian Academy Workshop on Acoustics*, Book 29, Bucharest, Romania, 1999, pp. 209–212 (in Romanian).

22. **Mihai Sima**, Victor Croitoru, and Dragoş Burileanu, "Performance Analysis on Speech Recognition using Neural Networks," in *Development and Application Systems Conference*, Suceava, Romania, May 1998, pp. 259–266.

23. **Mihai Sima** and Victor Croitoru, "Outlook on Intelligent Networks," in *Journal of Electronics and Computer Science*, Vol. 1, Technical University of Piteşti, Piteşti, Romania, 1996.

24. Victor Croitoru, Georgel Savu, **Mihai Sima**, and Ion Vonica, "Phone Directory for Voice and Data Integrated Terminals," in *Symposium on Electronics and Telecommunications*, Timişoara, Romania, 1994.

25. **Mihai Sima**, "Gasoline Electronic Economizer," in *New Ideas in Automobile Construction Symposium*, Piteşti, Romania, 1989.

26. **Mihai Sima**, "MOS Integrated Circuit for Dual-Tone Multifrequency Signals Decoding," in *The Student Research National Conference*, Timişoara, Romania, 1989.

27. Georgel Savu, and **Mihai Sima**, "MOS Integrated Circuit for Dual-Tone Multifrequency Signals Generation," in *The Student Research National Conference*, Iaşi, Romania, 1988, **Best Poster Award**.

Curriculum Vitae

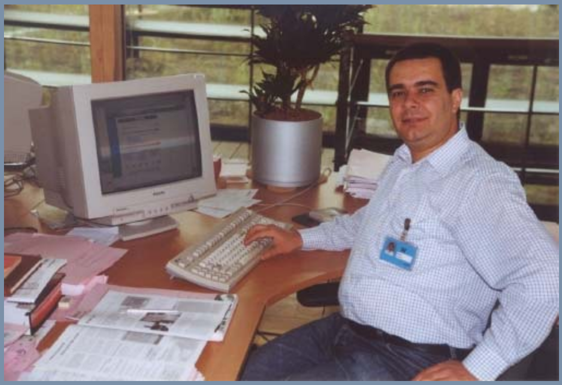
Mihai SIMA was born in Bucharest, Romania on the 5th of August, 1965. From 1979 he took the secondary education at the 'Mihai Viteazul' Lyceum in Bucharest, where he graduated in 1983 with the *Baccalaureate* degree. In 1984 he became a student of the Faculty of Electronics and Telecommunications, Polytechnical Institute of Bucharest, Romania. He obtained the *Electrical Engineer* degree in the Electronics and Telecommunications specialization in 1989.

From 1990 to 1993 he had been with 'Microelectronica' (MOS IC) company in Bucharest, Romania, as a Design and Test engineer. His work was focused on DC characterisation of CMOS integrated circuits. He was also involved in Instrumentation Electronics, with an emphasize on magnetic field and temperature measurements.

During 1993-1999 he had been with the Department of Telecommunications, Faculty of Electronics and Telecommunications, Polytechnical Institute of Bucharest, Romania as Research Engineer. His research was focused on speaker-independent speech recognition and text-to-speech synthesis by means of neural networks.

In 1999 he joined the Electrical Engineering Department, Delft University of Technology, Delft, The Netherlands, where he carried out a PhD stage with the Computer Engineering group under the supervision of Prof.dr. Stamatis Vassiliadis. His research activity was supported by a doctoral fellowship from Philips Research Laboratories, where he worked as a Guest Scientist. The outcome of this work is presented in this dissertation.

His research interests include computer architecture and engineering, reconfigurable computing, field-programmable gate arrays, multimedia processing, and speech recognition.



The ability of providing a hardware platform which can be metamorphosed under software control is a new approach in processor designing, very promising in terms of performance/cost ratio. The main idea is to give the programmer the freedom to adapt the processor functionality according to the characteristics of the program to be run. In this way, the designer decisions regarding the processor functionality are no longer suspicious to become a constraint for different application classes. Therefore, the designer decisions become the programmer decisions and a new processor can be defined on-the-fly. That is, you can actually have a new processor in microseconds rather than years.

The Computer Engineering Laboratory logo symbolizes the Antikythera Mechanism, the oldest known computing engine made by Human, which accurately reproduces the motion of the Sun and the Moon against the background of fixed stars. The mechanism was discovered in 1900 in a sunken ship just off the coast of Antikythera, an island between Crete and the Greek mainland. Several kinds of evidence point to 80 B.C. for the date of the shipwreck.