The Molen Polymorphic Media Processor

Media applications inherit high media specific computational power and wide data bandwidth requirements, potentially limiting performance efficient implementations on general purpose processors. This dissertation describes a reconfigurable processor, which can diminish and even overcome these application specific limitations while remaining as flexible as a general purpose processor. The proposal is referred to as *The Molen Polymorphic Media Processor* and it is based on the co-processor architectural paradigm. The basic idea comprises a core general purpose processor, which controls the execution and the reconfiguration of a reconfigurable co-processor, tuning the latter to specific media algorithms. A fully operational prototype implemented in the Xilinx Virtex II Pro™ technology is described. An experimental evaluation of the prototype is performed considering MJPEG, MPEG-2, and MPEG-4. The experimentally obtained speedups approach up to 98% of the theoretically attainable maximums.

The Molen Polymorphic Media Processor

Georgi Krasimirov Kuzmanov

ISBN : 90-9018801-0

Georgi Krasimirov Kuzmanov

Stellingen behorende bij het proefschrift /
Propositions to the Ph.D. thesis

The Molen Polymorphic Media Processor

van / by

Georgi Krasimirov KUZMANOV

Delft, 13 December 2004.

1. To speedup a media application, designer's efforts should be twofold: first, to increase the media specific computational power; second, to increase the data memory bandwidth.

2. To speedup program execution, conventional caches exploit linear spacial data locality. Many visual media algorithms, however, inherit multidimensional spacial locality. Therefore, conventional caches are not quite beneficial for such algorithms.

3. General purpose machines, augmented with reconfigurable hardware, can entirely fill the gaps between pure GPPs and pure ASICs both in flexibility and in performance.

4. "... the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude." *Amdahl, G.M. [1967]*
   Corollary: In GPP designers' society, accelerating an application 20% is considered spectacular. Meanwhile, in the ASIC world, an acceleration of 200% may be considered next to miserable.

5. A university engineering researcher should give industry the clearest indications for the feasibility and worthiness of his ideas.
   Corollary: Having a good idea is as important as properly presenting it to the potentially interested parties.

6. The most powerful engine that drives progress forwards is the human's curiosity. The general driving question is "What if...?".

7. People are unlimited in their desires, but limited in their capabilities.
   Corollary: Competing with yourself is the hardest competition to win.

8. A real help is not to pay one's bills but to teach one how to help oneself.

9. "Time exists in us, so do we exist in time. It changes us, so do we change it." *Vassil Levski (1837-1873)*, Bulgarian National Hero
   Consequently, we can not abandon our historical time, but we can change it.

10. Though moussaka is widely known to be Greek, it is actually a Mediterranean dish prepared according to different local traditions. One should taste Bulgarian moussaka to realize the obvious difference to, e.g., Samian moussaka. As a rule, however, nothing tastes better than mamma's mousaka.

These propositions are considered defendable and as such have been approved by the supervisor Prof. dr. Stamatis Vassiliadis.

1. Om een media toepassing te versnellen, zou de ontwerper zich twee doelen moeten stellen: Ten eerste het versnellen van de media specifieke rekenkracht; ten tweede het versnellen van de data geheugen bandbreedte.

2. Om de uitvoering van programmas te versnellen exploiteren conventionele caches de lineaire ruimtelijke lokaliteit. Vele visuele media algoritmen daarintegen hebben multidimentionale ruimetelijke lokaliteit. Hierdoor zijn conventionele caches niet goed voor dergenlijke algoritmen.

3. General Purpose Processoren (GPP), aangevuld met herconfigureerbare hardware kunnen zowel qua flexibiliteit als qua prestaties de kloof tussen GPPs en pure ASICs volledig dichten

4. "... de moeite die gespendeerd wordt aan het bereiken van hoge parallele verwerkingssnelheden gaat verloren tenzij het gepaard gaat met vergelijkbare sequentiele verwerkingssnelheden." *Amdahl, G.M. [1967]*
   Corollary: In de GPP gebied wordt het versnellen van een applicatie met 20% als spectaculair gezien. In de ASIC wereld daarintegen kan een versnelling van 200% als teleurstellend gezien worden.

5. Een universitair onderzoeksingenieur moet duidelijke indicaties over de haalbaarheid en waarde van zijn ideeën geven aan de industrie.
   Corollary: Het hebben van een goed idee is net zo belangrijk als het goed presenteren aan potentiële geintereseerden.

6. Het sterkste mechanisme dat de vooruitgang drijft is de menselijke nieuwsgierigheid. De algemene sturende vraag is: "Wat als...?".

7. Mensen zijn ongelimiteerd in hun verlangens maar gelimiteerd in hun kunnen.
   Corollary: De strijd met jezelf is de moeilijkste om te winnen.

8. Echte hulp is niet het betalen van iemands schulden maar iemand leren hoe zij zichzelf kan helpen.

9. "Tijd bestaat in ons, dus bestaan we in de tijd. Tijd verandert ons, dus veranderen wij de tijd." *Vassil Levski (1837-1873)*, Bulgaarse Nationale Held
   Het gevolg hiervan is dat wij onze historische tijd niet kunnen ontsnappen, maar wij kunnen het wel veranderen.

10. Hoewel moussaka bekend staat als een Grieks gerecht is het eigenlijk een mediteraans gerecht voorbereid volgens verschillende lokale tradities. Men zou de Bulgaarse moussaka moeten proeven om zich de duidelijke verschil te realiseren ten opzichte van bv. de Samiaanse moussaka. Maar de regel is in ieder geval dat niets beter smaakt dan mamma's moussaka.

Deze stellingen worden verdedigbaar geacht en zijn als zodanig goedgekeurd door de promotor Prof. dr. Stamatis Vassiliadis

The Molen Polymorphic Media Processor

The Molen Polymorphic Media 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 13 december 2004 om 13:00 uur

door

Georgi Krasimirov KUZMANOV

Computer Systems Engineer
Technical University of Sofia
geboren te Sofia, Bulgarije

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

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter	Technische Universiteit Delft
Prof. dr. S. Vassiliadis, promotor	Technische Universiteit Delft
Prof. dr.-Ing. J. Becker	Universität Karlsruhe
Prof. dr. ir. E. Deprettere	Universiteit Leiden
Prof. dr. ir. H.J. Sips	Technische Universiteit Delft
Prof. dr. John Long	Technische Universiteit Delft
Prof. dr. W. Luk	Imperial College London
Prof. dr. A. Popov	Technical University of Sofia
Prof. dr. C.I.M. Beenakker, reservelid	Technische Universiteit Delft

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Kuzmanov, Georgi Krasimirov
The Molen Polymorphic Media Processor
Georgi Krasimirov Kuzmanov. – [S.l. : s.n.]. – Ill.
Thesis Technische Universiteit Delft. – With ref. –
Met samenvatting in het Nederlands.
Съдържа кратък обзор на български език.

ISBN 90-9018801-0

Subject headings: reconfigurable machines, media processing, MJPEG, MPEG, microcode, performance, prototyping.

Printed in The Netherlands

To all my teachers, family, and friends
with gratitude and love

The Molen Polymorphic Media Processor

Georgi Krasimirov Kuzmanov

Abstract

In this dissertation, we address high performance media processing based on a tightly coupled co-processor architectural paradigm. More specifically, we introduce a reconfigurable media augmentation of a general purpose processor and implement it into a fully operational processor prototype. The proposed media Molen prototype is implemented on the Xilinx Virtex II Pro™ technology. Its entire "backbone" infrastructure utilizes *less than 1% of the reconfigurable resources* of the prototyping chip xc2vp20. Consequently, virtually the entire reconfigurable area is available for implementations of media processing units and memory. Such a reconfigurable area is used to addresses *computational intensive kernel* and *memory intensive access* requirements of media applications. More specifically, we consider for reconfigurable implementation several MPEG-4 performance limiting kernels including the repetitive padding, the accepted quality function, and the discrete wavelet transform. Compared to pure software execution, we obtain up to *two orders of magnitude* kernel speedups. The memory bandwidth limitation problem is solved by introducing a scalable, rectangularly addressable memory organization for accessing block-organized visual data. When implemented in hardware, the proposed memory organization suggests *8X data transfer speedups*. We evaluate the proposed processor prototype also at the application level by experimenting on MJPEG, MPEG-2, and MPEG-4. The experiments clearly indicate that our proposal can be employed to accelerate media applications. More specifically, the performance results obtained at the application level suggest that *overall application speedups of 2X-3X* can be expected, approaching *up to 98% of the theoretically attainable maximum application speedups*. Reconfigurable technologies, other than Virtex II Pro™ , are also considered and suggest similar performance improvements giving clear indications that our proposal is *general and technology independent*.

Acknowledgements

This dissertation was born after four years of enthusiasm and confusion, hopes and disappointments, hard work till late hours but also joy of sharing thoughts with people from numerous lands and cultures. I was granted the chance to enjoy and suffer all these emotions during my PhD study by one person who took the ”risk” of approving me for a PhD position just after two phone interviews. First and foremost, I would like to thank my advisor, prof.dr. Stamatis Vassiliadis, for being that person. It has been my privilege to work with dr. Vassiliadis whose professional expertise is indisputable and widely known. But I also appreciate the chance to know the visionary Stamatis who, with his energy, open mindness, and sunny character, made me feel the science fun.

My immediate thanks go to all my colleagues and friends from the Computer Engineering (CE) Lab family. The truly international environment they created enriched my personality beyond the scientific and professional frames of computer engineering. Special thanks to Georgi Gaydadjiev with whom we shared numerous scientific ideas, but also moments of fun. I thank Pyrrhos Stathis and especially Casper Lageweg, for the ”extraordinary efforts” they made to ensure that the propositions and the abstract of this dissertation sound just as good in Dutch. Also thanks to my roommate Jari Nikara for the interesting technical discussions including, by the way, our common passion for old time classic cars. I also give credits to my friends at the ”Bulgarian” lunch table in the faculty canteen for dissipating much of the homesickness I may have experienced otherwise.

I would like to acknowledge the institutions that financially supported my research, namely PROGRESS, the embedded systems research program of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs, and the Technology Foundation STW. Special thanks to my colleagues from the Artemis project (AES.5021), prof.dr. Ed Deprettere, dr. Andy Pimentel, dr. Jos Eindhoven, and Todor Stefanov, with whom we had a fruitful and successful collaboration.

To date, I also consider this dissertation an emanation of my lifetime. Therefore, I would like to express gratitude to all my teachers who greatly contributed for building my background of knowledge and will to qualify for the position in the CE Lab and to meet the challenges involved. My special thanks go to prof.dr. Angel Popov with whom I published my first papers and who, believing in me, recommended me to prof.dr. Vassiliadis. Thanks go to my former colleagues from "Info MicroSystems" Ltd., Sofia, and especially to the company head dr. Marin Marinov. They greatly contributed to the successful start of my engineering career in the friendly and enthusiastic atmosphere of a typical hightech startup.

Throughout the past four years, my warmest thoughts have always been with those Bulgarian friends of mine, with whom every time we meet is as if we have never separated. Despite the thousands of kilometers between us, I felt their close support, therefore I thank them warmly. Hereby, I would also like to thank my family and my close relatives for their lifetime love and support.

Finally, with my deepest love and gratitude I would like to thank my parents, Rumiana and Krasimir, for their love, patience, trust, advices, and support during the entire life of mine. I hope that I have been a son deserving them.

G.K. Kuzmanov Delft, The Netherlands, 2004

Contents

List of Tables

List of Figures

List of Acronyms

ACQ	ACcepted Quality (function in MPEG-4)
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Integrated Processor
BAB	Binary Alpha Block (in MPEG-4)
BRI	Block of Reconfigurable Instructions
CCU	Custom Computing (Configurable) Unit
c-set	complete set
DCT/IDCT	Discrete Cosine Transform / Inverse DCT
DWT/IDWT	Discrete Wavelet Transform / Inverse DWT
FLWT	Fast Lifting Wavelet Transform
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
HDL	Hardware Description Language
ISA	Instruction Set Architecture
LAM	Linearly Addressable Memory
LR	Link Register (of PowerPC)
MC	Motion Compensation (in all MPEGs)
ME	Motion Estimation (in all MPEGs)
MIR	MicroInstruction Register
MPEG	Motion Pictures Experts Group
JPEG	Joint Pictures Experts Group
MJPEG	Motion JPEG
PE	Processing Element
PN	Propagation Node (for MPEG-4 padding in this thesis)
p-set	partial set
RP	Reconfigurable Processor
SAD	Sum of Absolute Differences
VO	Video Object (in MPEG-4)
VOP	Video Object Plane (in MPEG-4)
XREG	eXchange REGister
XRn	XREG n
π**ISA**	polymorphic ISA
$\rho\mu$**-code**	reconfigurable microcode
ρ**CS-**α	reconfigurable Control Store address
ρ**CSAR**	reconfigurable Control Store Address Register
2DAM	Two-Dimensionally Addressable Memory

Polymorphism: The capability of assuming different forms;
the capability of widely varying in form.

Chapter 1

Introduction

The industrial impact of digital technology and its growing economical importance urged the development of media standards for digital visual compression such as JPEG, MJPEG, MPEG-1, and MPEG-2. The latest complete visual coding standard, MPEG-4 [1, 2], includes numerous new functionalities (e.g., content based coding, interactivity, natural and synthetic scenes and objects) that impose severe speed limitations to existing general purpose computers. In this dissertation, we assume such general purpose processors (GPP) and improve their performance in the media domain by introducing reconfigurable hardware extensions. We assume GPP platforms, as opposed to application specific, because of the flexibility, programmability, and compatibility features they posses [3]. In the present dissertation we consider high performance. Other parameters, such as power consumption, may lead to different architecture implementations and realizations[1] of reconfigurable processors and are considered as future research topics. We assume reconfigurable hardware extensions of GPPs, because they have shown considerable potential for speedups of computationally demanding algorithms.

This introductory chapter is organized in four sections. Section 1.1 delivers the minimal background on media processing required. A general reconfigurable design approach is sketched in Section 1.2. The thesis objectives are presented in Section 1.3 outlining the dissertation scope. Section 1.4 overviews the remaining contents of the dissertation.

[1] In this dissertation, we employ the terminology definition from [3] for the three conceptual issues of any computer design: the **architecture** of any computer system is the conceptual structure and functional behavior as seen by its immediate user (the programmer); the **implementation** is the logical organization of the dataflow and controls of a computer system; and the **realization** is the physical structure embodying the implementation.

1.1 Media background

In this section, we provide the minimal required background on the media applications considered in this dissertation, namely the visual data compression standards, with a special emphasis on MPEG-4. We also discuss some general open questions regarding the performance of the media processing computers.

1.1.1 Visual data compression

We start the introduction to the visual data compression with a brief presentation of the digital presentation of visual data.

Color spaces: In digital visual systems, data are presented in still pictures or frames (a sequence of still pictures) of natural scenes, sampled at regular intervals of time. Each picture/frame comprises a number of samples (referred to as picture elements, pels, or pixels), represented digitally by one or more numbers and organized in a two-dimensional rectangular array. A pixel of a gray scale (monochrome) image is represented by a single number, which indicates its brightness (luminance). To represent colors, multiple numbers per pixel are required. These numbers are organized in different systems, each referred to as a *color space*. A very popular color space is the RGB, where three independent numbers represent the intensity of each primary color of light, i.e., red/green/blue. In systems, utilizing RGB color space, each color is usually presented with the same number of bits. The RGB color presentation, however is not the most efficient one regarding data compression. Another popular color space widely utilized in visual data compression is the so called Y:Cr:Cb. This color space exploits the sensitivity of the human visual system to luminance, which is higher than the sensitivity to chrominance. The luminance number (Y) is the weighted average of the primary colors red, green, and blue. The two chrominance components (Cr and Cb) represent the difference between the red intensity and the luminance Y (Cr) , and the blue intensity and Y (Cb), respectively. The transformations between the RGB and the Y:Cr:Cb color spaces are extensively explained in the literature and we will not give further details on them. We just note that the key advantage of the Y:Cr:Cb over RGB is that the Cr and Cb components can be presented with a lower resolution than Y, because the human eye is less sensitive to color than to luminance. This makes Y:Cr:Cb more attractive for storage efficient digital visual presentation and for visual data compression. Therefore, both the still pictures compression standard JPEG, as well as the moving pictures standards MPEG adopt the Y:Cr:Cb full color representation.

Macroblocks: The basic building block of a JPEG or an MPEG picture is the macroblock. A macroblock comprises one Y 16x16 block of luminance pixels and two chrominance (Cr and Cb) blocks. The dimensions of the Cr and Cb blocks depend on the particular color resolution. In the most popular macroblock format, the so called 4:2:0, the dimensions considered for a chrominance block are 8x8. The dimensions of the chrominance blocks (thus the chrominance resolution) may vary resulting in different macroblock formats, e.g., 4:2:2 and 4:4:4 (more details can be easily found in the literature).

Digital visual data compression is mainly based on exploiting specific properties of the human visual system to reduce the redundancy in the visual data. In processing still pictures (JPEG), the aim is to reduce *spacial redundancies* in the image. For moving pictures (MPEG), both *spacial* and *temporal redundancies* have to be reduced.

Spacial redundancy: A common approach to reduce the spatial redundancy in a picture is the utilization of *orthogonal transforms*. The two-dimensional Discrete Cosine Transform (2D DCT) is the most popular and the most widely implemented transform in visual data compression. It is performed over each 8x8 block and is used as a basic approach in JPEG and all MPEG standards. Basically, the DCT decomposes visual data into discrete spatial frequencies, concentrating the image energy in a small number of large valued transform coefficients. The DCT transform coefficients can be processed in a manner, consistent with the properties of the human eye. In a quantization algorithm, following the DCT, the small coefficient values are discarded and only the substantial ones are considered for further processing. Thus, reducing the amount of visual data, some level of compression is obtained. The DCT is a *block-based transform*. Other orthogonal transforms consider the entire image rather than discrete pixel blocks and they are referred to as *image-based transforms*. In JPEG2000, as well as in some algorithms for still images compression in MPEG-4, the image based Discrete Wavelet Transform (DWT) is utilized. DWT is based on the so called *wavelets*, a mathematical concept for function decomposition [4]. Some of the features of the wavelets, that make them very successful and widely implemeted in recent image compression algorithms are:

- Wavelets provide high compression ratios: in terms of visual quality they perform much better than competing technologies like DCT.
- The wavelet transforms are symmetric: both the forward and the inverse transform have the same complexity, allowing fast compression and decompression.

- Multi-resolution signal analysis allows progressive transmission and zooming, without the need for extra storage.
- Wavelets can be used for various image-processing operations. The possibility to combine image processing and compression is a very appealing factor.

Temporal redundancy: To exploit temporal redundancy, all MPEG standards adopt *motion compensation* techniques. Motion compensation is a process of coding differences (motion) between frames in a video sequence [5]. These differences are estimated as a displacement between pixel areas in the current frame (being encoded) and a previously encoded frame. The measurment of this displacement is the *motion vector*. A process, called *motion estimation*, is performed to determine the motion vectors for each macroblock. This process includes a search algorithm for best matching between the block to be encoded and an area of previously encoded frame. As a criteria for best block matching, the minimal Sum of Absolute Differences (SAD) function is usually used. The SAD sums all absolute differences between the corresponding pixels in two pixel blocks. The best matched pixel area in the reference picture is the one that minimizes its SAD with the current block.

1.1.2 MPEG-4 - the content-based coding standard

MPEG-4 [1, 2] aims at providing descriptions of tools and algorithms for efficient storage, transmission and manipulation of video data in various multimedia environments. The basic approach relies on the *content-based coding*, which, combined with various new functionalities, makes MPEG-4 radically different from its predecessors. This approach contributes to more efficient compression and better visual quality at comparable bitrates. Furthermore, content-based representation of visual data gives the end user opportunities for interaction with the content of a visual scene.

Video objects and video object planes: For content-based coding, MPEG-4 uses the concept of a *video object plane* (VOP). A VOP is an arbitrarily shaped region of a frame, which usually corresponds to a semantic object in the visual scene. A sequence of VOPs in the time domain is referred to as a Video Object (VO). This means that we can view a VOP as a ”frame” of a VO. Each of the video objects is transmitted by a separate bitstream of arbitrary-shaped VOPs. Each VOP in MPEG-4 is defined by its *shape* and *texture*, which are coded differently.

VOP shape: In MPEG-4, shape is used to distinguish an object from the background and to identify the borders of a VOP. The shape information is provided in *binary* or *grayscale* format. The binary format represents the object shape as a pixel map, which has the same size as the bounding rectangular box of the VOP. Each pixel from this bitmap takes one of two possible values, which indicate whether a pixel belongs to the object or not. The binary shape representation of a VOP is referred to as *binary alpha plane*. This plane is partitioned into 16x16 *binary alpha blocks* and each binary alpha block (BAB) is associated with the macroblock, which covers the same picture area. In the grayscale shape format, each pixel can take a range of values, which indicate its transparency. The transparency value can be used for different shape effects (e.g.,blending of two images).

VOP texture: Texture encoding of a VOP macroblock is performed with respect to its shape. There are three types of macroblocks in an arbitrary shaped VOP: *opaque macroblocks* completely located inside the VOP, *boundary macroblocks* containing the VOP boundary pixels, and *transparent macroblocks* entirely outside the VOP boundary. Transparent macroblocks are discarded and not encoded and the internal macroblocks are processed by conventional 2D DCT. For boundary macroblocks, different techniques such as shape adaptive DCT (SA-DCT) are employed.

Motion estimation: In MPEG-4, motion estimation is similar to MPEG-1/2 with some modifications. The most important new features in motion estimation algorithms for arbitrary shaped VOPs are the special *padding* techniques and the agreement on a *coordinate system*. The purpose of padding is to ensure more accurate block matching by replacing the pixels outside the boundary of the VOP with certain values. In MPEG-4, an object can be anywhere in a video frame, so an *absolute frame coordinate system* is used for referencing the position and motion of all VOPs.

Synthetic objects: In addition to the scenes and objects of natural video, referred to as *natural scenes and objects*, MPEG-4 also presents the option to combine synthetic scenes and objects with natural ones. The standard treats synthetic objects as a subset of the computer graphics and includes *facial*, *body* and *2D mesh* animation. Synthetic object processing, however is not considered in this dissertation.

Profiles@Levels and real-time implementability: Assuming audio-visual data compression standards, MPEG-4 [2] is the first to address content-based coding concepts. These new concepts impose a large number of specific techniques, approaches, and tools, which implement the standard on various in-

teractive multimedia environments. Unlike its predecessors, MPEG-4 is much more demanding in terms of computational complexity with even more data intensive algorithms. To allow the efficient implementation of the standard, the MPEG-4 requirements define several application profiles. Within each profile, a number of levels constrain the computational complexity and the required data bandwidth of the application. Each profile level states the values for certain parameters, which are used to judge whether an application meets the functional and implementational requirements of the level. Table 1.1 presents the required data processing speed according to the MPEG-4 Visual

Table 1.1: MPEG-4 Visual Profiles@Levels definitions and processing speed in MacroBlocks per second [MB/s].

Profile	**Level**	**Session Size**	**# VO**	**Max. MB/s**	**Boundary MB/s**
Main	L4	1920x1088	32	489600	244800
	L3	CCIR 601	32	97200	48600
	L2	CIF	16	23760	11880
	L1	N.A.	N.A.	N.A.	N.A.
Core	L2	CIF	16	23760	11880
	L1	QCIF	4	5940	2970
Simple Scalable	L2	CIF	4	23760	N.A.
	L1	CIF	4	7425	N.A.
Simple	L3	CIF	4	11880	N.A.
	L2	CIF	4	5940	N.A.
	L1	QCIF	4	1485	N.A.

Profiles@Levels definitions [6]. The *Simple* Visual Profile provides efficient coding of rectangular video objects. The *Simple Scalable* Profile is useful for applications, providing more than one level of quality, e.g., Internet use. The *Core* Profile is the first to deal with arbitrary-shaped and temporally scalable objects, useful where a relatively simple content interactivity is required (e.g., Internet multimedia). The most demanding visual profile is the *Main* Profile. It augments the functionality of the Core profile by coding of interlaced, semi-transparent, and sprite objects. It can be used for interactive and entertainment-quality broadcast and DVD applications [2]. At the highest level of the Main profile (L4 in Table 1.1) a session with a frame size of 1920x1088 is processed, containing up to 32 video objects (VO) at a maximum of 489600 macroblocks[2] per second. The last column of the table represents the required

[2]In Table 1.1, MB/s denotes *macroblocks per second* and should not be confused with MBytes per second

boundary macroblocks per second, which is an important criteria for evaluating the devices we are presenting further in this dissertation. Considering the above explanations, we can conclude that the general performance demands of the Simple MPEG-4 Profile are approximately the same as of MPEG-2, since in this profile only rectangular video objects are defined. Therefore, it is most challenging to meet the requirements of the most-demanding Core and Main Visual Profile Levels of MPEG-4, where arbitrary-shaped visual objects are processed. Complexity analysis [7] indicates that real-time software implementations of the intermediate CoreProfile@Level1 require more than 5 billion RISC-like instructions per second. Consequently, we can safely conclude, that real time implementations of the highest profiles and levels of MPEG-4 would cost substantially more instructions per second (up to the order of 100 billion). These processing requirements will significantly exceed the capabilities of the general purpose processors, despite near future technology improvements.

1.1.3 Media design requirements and potential limitations

The specific functionalities of the media standards in many cases require performance, exceeding the capabilities of the contemporary GPPs. Moreover, many of these media functionalities impose performance requirements that may not be met by the GPPs, despite the future silicon technology advances in industry. To be more specific, as numerous explorations and analysis in the literature and in practice suggest, the most crucial performance requirements a media system must meet are for a *high computational power* and an *enormous data throughput*. Therefore, new processor designs, capable of meeting these two key media performance requirements, are needed. To design adequately performing media processors, we can approach the problems from different points of view concerning the *architecture*, the *implementation*, and the *realization* of the media processor.

Architectural prospective: From the architectural point of view, we can define some issues that would help the implementation to meet the performance requirements. An architecture, entirely dedicated to the application field (e.g., the MPEG standards), would obviously enable the implementation of a high performance specialized processor, but this is the most costly solution. A far more flexible and cost-effective approach is to redefine or possibly augment an existing general purpose architecture.

Increased computational power: To increase the application specific computational power of a GPP, a popular approach is to define new, application specific instructions as an extension of the general purpose Instruction Set Architecture

(ISA) [8–12]. The application domain must be analyzed for computationally demanding functions or program kernels that would effectively improve system performance when implemented as fast specialized instructions. In MPEG 1, 2 and 4, examples of such kernels are the DCT/IDCT transforms, as well as the motion estimation and compensation algorithms. In MPEG-4, the new functionality can be accelerated by defining new instructions supporting the shape encoding, padding, or DWT.

Larger data throughput: Another important element of an architecture is its *basic data structure* also referred to as data type. If we carefully choose these structures, we can also expect performance benefits. While in most GPP the data types are bits, bytes and words, in many media standards, the 8x8 *pixel block* can be defined as a basic data structure. Similarly, the MPEG-4 *binary alpha block* can be referred to as a separate basic data type. Thus, utilizing important general features of the block-organized data, such as *data locality* and *data reusability*, combined with ISA extensions supporting block processing, significant speedups can be enabled for underlying implementations.

Implementation prospective - potential limitations: The implementor must accomplish the conceptual structure defined by the architecture into a logical organization utilizing limited budget of hardware resources. In media domain, however, the implementation process may face some problems, for instance:

- The architectural approach to increase computational power by introducing a unique instruction for each considered media functionality imposes serious implementation problems regarding the number of the newly defined instructions. *The fixed and limited instruction format may become prohibitive to implement larger numbers of unique instructions.*
- Another implementation drawback can be caused by the *diversity of profiles and functionality contexts defined in the same media application*. Typically, media standards do not state precisely all the algorithms that should be used to implement the described functionality and leave more freedom to the implementors. Furthermore, the implementations of some standard functionalities are optional (e.g., sprite encoding in MPEG-4) and in many cases they can not even coexist with other optional functionalities. Considering the above issues, the implementation of a hardwired accelerator for each new functionality is not the perfect solution due to three key reasons:
    1. The *lack of flexibility* in hardwired implementations and the long design cycles required, make such a solution inadequate to the dy-

namic changes in the media application domain.

2. A *large number of hardwired functional accelerators* may require silicon area exceeding the available device budget. In such cases, implementing the hardwired accelerators altogether would become practically prohibitive.
3. MPEG encoders and decoders have *different computational requirements*. Therefore, it is not cost-efficient to implement the standard functionality into hardwired circuits, which in some application contexts can be extremely performance efficient, while in others may not be utilized at all.

- *Limited hardware resources* may cause severe implementation problems. While in desktop implementations some performance gains might be achieved at the cost of enhanced amount of hardwired resources, this may not be a solution in the embedded systems domain, where the budged of system resources is by default far more limited.

Reconfigurable hardware, coexisting with a general purpose processor, has been considered as a good candidate to address the media implementation and performance limitations addressed above. Some machines employing such an approach are presented below.

Reconfigurable machines: First of all, it is noted that in this dissertation, we utilize the term *reconfigurable machine* as a general purpose processor augmented with reconfigurable hardware (e.g., FPGA)[3]. Numerous design concepts and organizations have been proposed to support this reconfigurable computing paradigm. Some popular proposals of reconfigurable processors are: PRISC [13], OneChip [14], RISA [15], Garp [16], ConCISe [17], PRISM and PRISM-II [18, 19], Chimaera [20], the SONIC architectures [21–24], etc. Furthermore, in the recent years, many industry leading vendors released soft GPP cores for reconfigurable processing. Popular examples are the Microblaze™ of Xilinx [25], Nios™ of Altera [26], Xtensa™ of Tensilica [27], Avispa™ and Moustique™ of Silicon Hive (`http://www.siliconhive.com/`). Xilinx and Altera made a step further introducing FPGAs with embedded hard GPP cores- the PowerPC™ and ARM™ , respectively. Even though the approach to combine a GPP with reconfigurable hardware is promising and many paradigms have been proposed

[3]We note that reconfigurable designs, which do not incorporate a GPP are not considered in this dissertation and will not be discussed further.

(for a complete list of reconfigurable approaches in addition to the ones referenced above, see [28–30]), the architectures and organizations of such hybrid processors can be viewed mostly as open topics. Moreover, there exist common shortcomings that characterize to various degrees the previously referenced reconfigurable proposals described by the following:

Shortcomings of current reconfigurable proposals: Traditional general purpose media extensions, such as [10–12, 31], require long development cycle, permanent op-code space for each domain considered, and restrict the number of the functions to be implemented in hardware to very few. Reconfigurable processors have partially resolved the above problems as they allow to map a program portion to hardware, possibly even automatically [32, 33]. Currently, however, schemes assuming a GPP augmented with reconfigurable fabric (e.g., [20, 34, 35]) still introduce a new instruction for each portion of the application implemented in the FPGA. As a result, for a specific application domain intended to be implemented in the FPGA, the designer and the compiler are restricted by the unused opcode space. Due to the larger number of new reconfigurable operations supported, the *opcode space explosion* problem is still presented and it can become severe for some applications. Moreover, current reconfigurable proposals introduce some additional disadvantages, summarized below:

- **Lack of compatibility**: this shortcoming is related to the opcode space explosion problem. Due to the fact that each newly introduced instruction has its unique format and encoding, general ISA compatibility is not achievable.
- **Limited number of parameters**: In several proposals, the operations mapped on an FPGA can only have a small number of input and output parameters (e.g., [36, 37]). For example, in the architecture presented in [36], due to the encoding limits, the fragments mapped into the FPGA have at most 4 inputs and 2 outputs. Similarly, in [37], the maximum number of input registers is 9 and output parameters can be passed trough only one output register.
- **No support for parallel execution** of sequential operations on the FPGA: The parallel execution of sequential operations can be an important and powerful feature for reconfigurable computing, provided that the data dependency allows it. Many reconfigurable architectures do not take into account this issue and their mechanism for FPGA integration cannot be extended to support parallelism (see for examples in [28]).

- **Technology dependence**: each approach has a specific definition and implementation bounded for a specific reconfigurable technology and design. Therefore, the applications cannot be ported to a new reconfigurable platform without substantial efforts.
- **No modularity**: There are no mechanisms allowing reconfigurable implementation hardware to be developed separately and ported transparently for the software, as indicated in [38]. This shortcoming is also related to the aforementioned technology dependance problem.

The Molen processor paradigm [39,40] addresses and solves the shortcomings of the current reconfigurable proposals discussed above. More details on how all these shortcomings are resolved by Molen are presented in Chapter 2. In all referenced reconfigurable approaches, including Molen, however, the introduction of GPPs coexisting with reconfigurable hardware imposes design approaches different from the traditional ones. In the section to follow, we explain such a general reconfigurable design approach.

1.2 General reconfigurable approach

A general Hardware-Software Co-design methodology, used to fit a given (media) application into a GPP augmented with reconfigurable hardware is sketched in Figure 1.1. The design process is performed in several interactive stages. First, an analysis of the application algorithms is performed. This stage requires extensive profiling and software-hardware (SW/HW) partitioning of the application. Candidate functions or kernels for hardware implementation are identified through the SW/HW partitioning and considered for further hardware design. SW/HW interface solutions have to be made at this initial design stage as well, and later considered for program code annotation and hardware implementation. The remaining design stages are performed in two separate tracks, interacting with each other - one in software and the other in hardware.

Software track: Consider Figure 1.1. The original application code is first modified according to the SW/HW partitioning and the interface solutions made in the preceding stage. Usually, these modifications include code annotations utilizing either high level programming language techniques (e.g.,in C, Java, etc.) or a lower assembler level language. The modified/annotated application code is then compiled and linked for the targeted GPP architecture. If code annotations are made in high level programming language manner, an

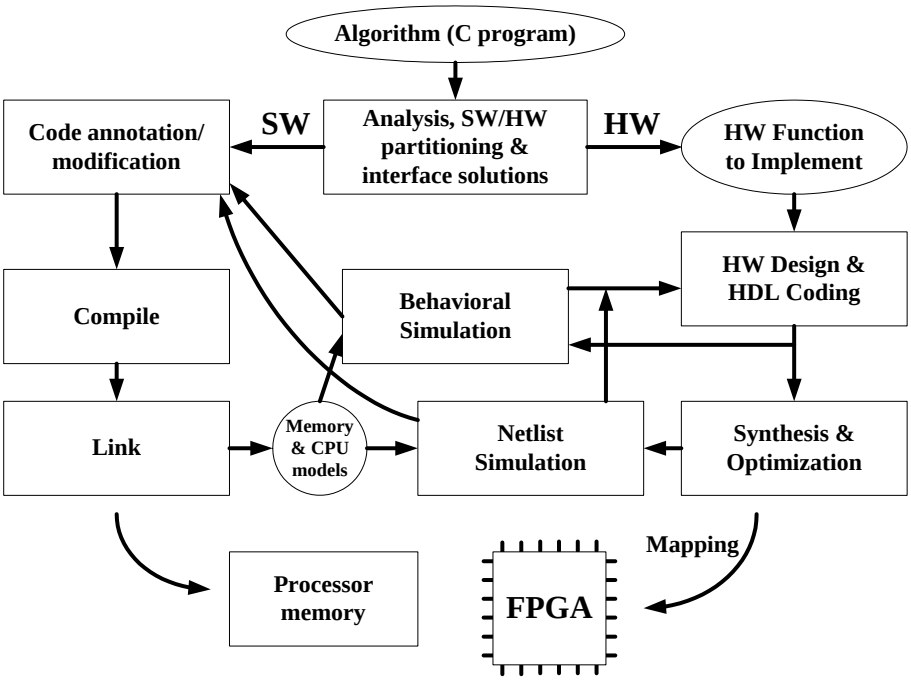


Figure 1.1: Make applications fit - a typical reconfigurable design flow.

accordingly modified retargetable compiler has to be used. In the case of lower assembler level annotations, the native compiler for the GPP architecture can be employed. The result of the compilation and link processes is a single or a number of binary sequences (codes), each of them dedicated for a certain location in the target memory organization. The generated binary codes are loaded into corresponding memory models for SW/HW co-simulation.

Hardware track: Consider Figure 1.1. Hardware units supporting the functions extracted for HW implementation are designed and coded in hardware description language HDL. The HDL models are simulated at behavioral level to validate the functional correctness of the designs. Behavioral simulations may be performed over stand-alone models of the units. It is far more essential, however, behavioral simulations to be performed over a model of the entire reconfigurable processor, i.e., including the compiled application programs. The results of these simulations may impose changes in the initial hardware design as well as some changes in the program code annotations. After the reconfigurable design is validated at behavioral level, the HDL codes of the hardware units are synthesized and optimized. Once again, the resulting netlist design

description is co-simulated with the software to detect possible design errors. Performed at lower level of abstraction, this simulation is the final validation of the entire reconfigurable design before its physical implementation. Finally, the synthesized and optimized design is mapped onto the targeted reconfigurable device (FPGA) and a configuration bitstream is generated.

Software-Hardware Tracks Interaction: The interaction between the software and hardware design tracks, as depicted in Figure 1.1, is mainly performed during several design validation phases. For design validation, we consider SW/HW co-simulations at different levels of abstraction. There are numerous methods for simulating the reconfigurable design, which can be adapted to the approach from Figure 1.1. A discussion regarding the relevance and appropriateness of each of these methods would be outside the scope of this thesis, thus not considered further. In the particular approach, cycle accurate event-driven HDL simulations are assumed. During the simulation phases, design errors may occur both in the hardware and in the software. In these phases, the design process is iterative and after relevant changes in the designs (resp. their source codes), the process is repeated. Once an error free design is obtained and validated, the software-hardware co-simulation is considered complete. Next, binary codes for the distinct locations of the targeted memory organization are generated. An FPGA configuration bitstream is generated from the synthesized and optimized HDL code of the hardware design track. Finally, the linked binary codes of the application software are loaded into the physical memories of the processor and the generated FPGA bitstream is loaded into the targeted reconfigurable device(s).

1.3 Dissertation objectives

In this dissertation, we focus on media applications with emphasis on the visual data compression standards MJPEG and MPEG. We are particularly interested in MPEG-4 due to the computationally demanding functionalities it incorporates. To solve the performance problems regarding the execution of media applications on GPPs, we introduce reconfigurable computing extensions supporting the specific computational requirements of the considered media algorithms. As identified in Subsection 1.1.3, current reconfigurable proposals suffer a number of drawbacks, which have been resolved by the Molen processor paradigm [39, 40][4]. In this dissertation, we investigate how such a paradigm can be augmented and applied to media processing represented by MJPEG,

[4] A detailed discussion on how such drawbacks are resolved is presented in Chapter 2.

MPEG-2, and MPEG-4. More specifically, the following objectives determine the scope of this dissertation:

- **Solve media computational complexity problems:** We consider several hardware units that perform media specific operations efficiently. More specifically, we consider the MPEG-4 repetitive padding, the MPEG-4 accepted quality function, and the discrete wavelet transform. We also consider the sum of absolute differences, the discrete cosine transform, and the inverse discrete cosine transform. Experiments suggest that dramatic performance improvements, *up to two orders of magnitude*, can be expected for the kernels considered. (The media specific computational demanding problems are covered in **Chapter 3**).

- **Address and solve specific media memory access problems:** The memory bandwidth limitation problem is solved by introducing a new scalable memory organization, which is controlled at microarchitectural level and delivers sufficient amount of data to the units processing block-organized visual data. Experiments suggest that data transfer *speedups of 8x* can be expected. (**Chapter 4** presents the details).

- **Address reconfigurable processor prototyping:** We propose a Molen prototype implementation on the Virtex II Pro technology of Xilinx, referring to the embedded PowerPC core as to a ”black box”. Thus, without having to redesign the GPP core, we emulate reconfigurable operations using the original PowerPC ISA. The implemented Molen organization efficiently redirects (arbitrates) reconfigurable and standard instructions either to the GPP or to the reconfigurable units. A data communication mechanism between the GPP and the reconfigurable units exploits dedicated parameter exchange registers and shared memory space. Important software considerations supporting the prototype are delivered. The entire Molen ”backbone” infrastructure is implemented in reconfigurable hardware, consuming *less than 1% of the available reconfigurable resources* of the prototyping chip xc2vp20. This leaves virtually the entire FPGA area for the application specific reconfigurable implementations. (The prototype is described in **Chapter 5**).

- **Experimental prototype evaluation:** We carry out series of experiments on MJPEG, MPEG-2 and MPEG-4 to evaluate the performance efficiency of the implemented Molen prototype. The theoretical boundaries of the maximum attainable speedups are investigated and established as reference for our measurements. The experimentally obtained

performance results for the Virtex II Pro Molen prototype suggest that speedups of 2X-3X can be expected. In some scenarios, the speedups approach *up to 98% of the theoretically established maximum attainable speedups*. We also investigate the influence of the attained kernel speedups of the implemented reconfigurable accelerators on the overall speedup of the application. The boundaries of the cost-effective local speedups of the accelerated kernels are investigated and determined. (The experimental evaluations are presented in **Chapter 6**).

- **Technology independence:** Although the Virtex II Pro technology has been considered for the final prototype, in our designs we considered other technologies of Xilinx as well as technologies of other vendors, such as Altera. Evaluations for MIPS GPPs, rather than just for PowerPC are also presented. Thus, we prove the applicability of our approach on different technologies, i.e., its *technology independence*. (We prove the technology independence of the proposal by assuming different reconfigurable technologies in **Chapters 3**, **5**, and **6**).

An overview of how the research objectives have been attained and how they are presented in this dissertation follows.

1.4 Dissertation overview

This dissertation contains seven chapters in total described by the following:

In Chapter 2, a brief description of the Molen $\rho\mu$-coded polymorphic processor is presented. This reconfigurable machine is described starting with the introduction of the general approach, followed by a concise description of the machine organization, the underlying microarchitecture, programming paradigm, and a discussion on the program sequence control. The discussion also emphasizes on some specific features of this conceptually distinct machine organization, which help to overcome several common shortcomings of the recent reconfigurable proposals. This dissertation targets the Molen polymorphic processor as a research platform for accelerating media applications.

Chapter 3 introduces three original hardware accelerator designs of high profile MPEG-4 specific functions. The operation of the Molen processor is based on the co-processor architectural paradigm. More specifically, a general-purpose processor controls the execution and the configuration of a reconfigurable co-processor, tuning the latter for specific algorithms. The proposed accelerators

in Chapter 3 are intended to be implemented as operational units within the reconfigurable co-processor, thus increasing the computational power of the Molen processor. To be more specific, three accelerating units are considered, supporting the following MPEG-4 operations: *repetitive padding*, *accepted quality function*, and lifting based *discrete wavelet transform*.

To increase the data memory bandwidth required by a number of multimedia accelerators, a supporting memory organization is proposed in Chapter 4. As an alternative of traditional linearly addressable memories, we suggest a memory organization based on a rectangular array of memory modules. We also discuss the interface between the proposed memory organization and a linearly addressable memory accompanied by comprehensive examples. Synthesis and experimental results indicate reasonably small reconfigurable hardware costs and promising high performance figures. The design is envisioned to be more cost-effective compared to related works.

A Xilinx Virtex II Pro based prototype of the Molen processor is described in Chapter 5. Utilizing the embedded PowerPC processor we implement the Molen paradigm by emulating reconfigurable operations with the original PowerPC ISA. A minimal functionally complete ISA of only four additional instructions is implemented by the proposed Molen prototype. The discussion is focused on the microarchitectural support for the implemented ISA extension emulated on the embedded PowerPC 405 processor in the Virtex II Pro FPGA. Some important considerations regarding the software support of the proposed Molen prototype are discussed as well.

Due to the closely coupled co-processor based Molen organization we achieve performance efficient processing, proved by experiments in Chapter 6. An evaluation methodology comprising three approaches with respect to the requirements of the prototype and the application are considered. Theoretical grounds supporting the methodology are established to analyze the prototype performance data for three considered applications, namely MJPEG, MPEG-2, and MPEG-4.

Finally, concluding remarks are presented in Chapter 7. The chapter summarizes the dissertation, outlines its contributions and proposes future research directions.

Chapter 2

Molen Background

In this chapter, we briefly present the Molen polymorphic processor paradigm symply referred to as Molen in the remainder of the presentation. The bases of a Molen processor are originally introduced in [39]. The general proposal is: *by displaying means to maintain the reconfiguration at architectural level, to achieve a high flexibility in tuning the system for the specific application*. The operation of Molen is based on the co-processor architectural paradigm. Details regarding the general approach, architecture, microarchitecture, organization and implementation of Molen are gradually presented in this chapter and in the chapters to follow.

More specifically, this chapter is organized as follows. The general Molen approach is presented in Section 2.1. Details on the underlying organization and microarchitecture of a Molen polymorphic processor are discussed in Section 2.2. Section 2.3 describes the Molen programming paradigm and adds details on the program sequence control. Finally, the chapter is concluded with Section 2.4.

2.1 General approach

In the discussion to follow, we present the general concept of transforming an existing program to one that can be executed on the Molen reconfigurable computing platform and hints to the new mechanisms, intended to improve existing approaches. The conceptual view of how program P (intended to execute only on the GPP) is transformed into program P' (executing on both the GPP core and the reconfigurable hardware) is depicted in Figure 2.1. The purpose

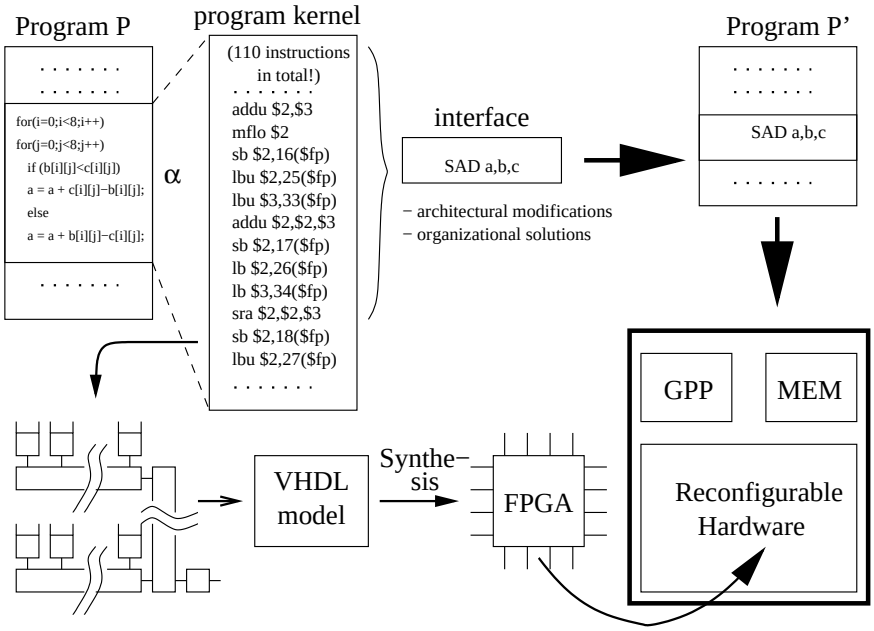


Figure 2.1: The general Molen approach: program transformation example.

is to obtain a functionally equivalent program P' from program P which (using specialized instructions) can initiate both the configuration and execution processes on the reconfigurable hardware. The sum of absolute differences (SAD) calculation, a well known multimedia operation, is considered as an example in Figure 2.1. The steps involved in this transformation are the following:

1. **Identify** pieces of software code "α" in program P to be mapped in reconfigurable hardware
2. **Design** a hardware unit performing the functionality of the extracted program kernel "α" and describe the design in HDL (e.g.,VHDL). Show that "α" can be implemented in hardware in an existing technology, e.g., FPGA, and map "α" onto reconfigurable hardware.
3. **Eliminate** the identified code "α" from program P. Insert an equivalent code A (e.g., SAD a,b,c), which calls the hardware through a preestablished SW/HW calling interface. This interface reflects the architectural and organizational modifications of the original GPP and comprises:
    - Parameters and results communication between the GPP and the reconfigurable processor.

- Configuration code, inserted to configure the reconfigurable hardware.
- Emulation code, used to perform the functionality of the hardware accelerated kernel "α".

4. **Compile and execute** program P' with original code plus code having functionality A (equivalent to "α", i.e., SAD a,b,c) on the GPP/reconfigurable processor.

The above steps illustrate a programming paradigm in which both software and hardware descriptions are present in the same program. It should also be noted that the only constraint on "α" is its implementability, which possibly implies complex hardware. Consequently, due to the complexity of this hardware, the microarchitecture may have to support emulation [41], which in turn requires the utilization of microcode. This reconfigurable microcode is termed ($\rho\mu$-code) as it is different from the traditional microcode. The difference is that such microcode does not execute on fixed hardware facilities. It operates on facilities that the $\rho\mu$-code itself "designs" to operate upon.

2.2 Organization and microarchitecture

In this section we briefly describe the Molen organization and the underlying microarchitecture.

The Molen organization: The two main components in the Molen machine organization (depicted in Figure 2.2) are the 'Core Processor', which is a general-purpose processor (GPP), and the 'Reconfigurable Processor' (RP). Instructions are issued to either processors by the 'Arbiter' by means of a partial decoding of the instructions received from the instruction fetch unit. Data are fetched (stored) by the 'Data Fetch' unit from(to) the main memory. The 'Memory MUX' unit is responsible for distributing(collecting) data to(from) either the reconfigurable or the core processor. The reconfigurable processor is further subdivided into the $\rho\mu$-code unit and the *custom configured unit* (CCU). The CCU consists of reconfigurable hardware, e.g., an FPGA, and memory. Essentially, the CCU is intended to support additional and future functionalities that are not implemented in the core processor. Pieces of application code can be implemented on the CCU in order to speed up the execution of the overall application code. A clear distinction exists between code that is executed on the reconfigurable unit (the RP targeted code) and code that is executed on the core processor (remaining code). Data must be transferred across the code

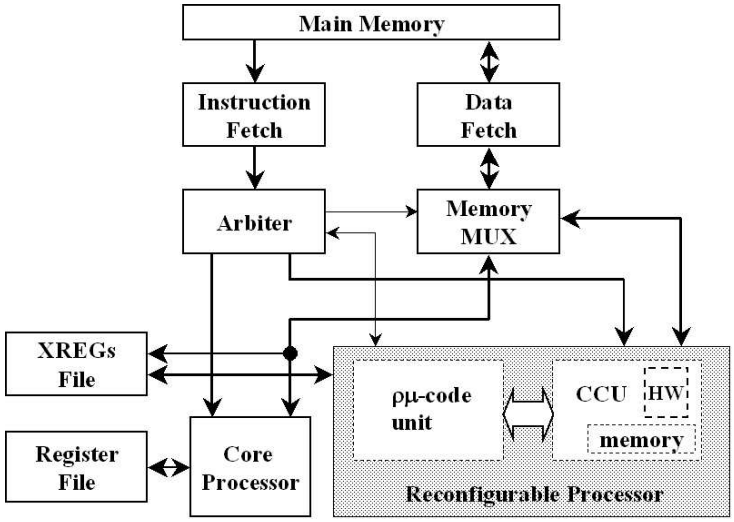


Figure 2.2: The Molen machine organization.

boundaries in order for the overall application code to be meaningful. Such data includes predefined parameters (or pointers to such parameters) or results (or pointers to such results). The parameter and result passing is performed through a mechanism utilizing so-called exchange registers (XREGs) depicted in Figure 2.2. This mechanism is described in Section 2.3.

The support of operations by the reconfigurable processor can be initially divided into two distinct phases: set and execute. In the set phase, the CCU is configured to perform the supported operations. Subsequently, in the execute phase the actual execution of the operations is performed. This decoupling allows the set phase to be scheduled well ahead of the execute phase and thereby hiding the reconfiguration latency. Furthermore, no specific instructions are associated with specific operations to configure and execute on the CCU as this will greatly reduce the opcode space. Instead, pointers to *reconfigurable microcode* ($\rho\mu$-code), which emulates both the configuration and the execution of programs, are used. Consequently, two types of $\rho\mu$-code are distinguished:

- reconfiguration microcode that controls the configuration of the CCU;
- execution microcode that controls the execution of the implementation configured on the CCU.

The Molen microarchitecture: Experienced microcode designers will recognize that for performance reasons, there is a necessity of having microcode

that resides permanently in the control store and microcode that is pageable. To represent this difference, a bit from the instruction word is dedicated to implement resident/pageable microcode.In the instruction format, depicted in Figure 2.3, the location of the microcode is indicated by the resident/pageable-bit (R/P-bit) which implicitly determines the interpretation of the address field, i.e., as a memory address α (R/P=1) or as a ρ-control store address ρCS-α (R/P=0) indicating a location within the $\rho\mu$-code unit. This location contains the first instruction of the microcode which must always be terminated by a dedicated microinstruction, e.g., *end_op*.

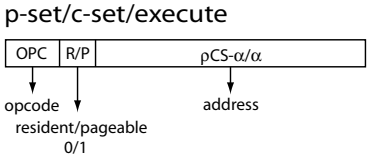


Figure 2.3: The p-**set**, c-**set**, and **execute** instruction format.

The $\rho\mu$-code unit: The internal organization of the $\rho\mu$-code unit is depicted in Figure 2.4. The $\rho\mu$-code unit comprises three main parts: the sequencer, the ρ-control store, and the $\rho\mu$-code loading unit. The sequencer mainly determines the microcode execution sequence. The ρ-control store is used as a storage facility for microcode. The $\rho\mu$-code loading unit, as its name suggests, is responsible for the loading of reconfigurable microcode from the memory. The execution of microcode starts with the sequencer receiving an address from the arbiter (see Figure 2.2) and interpreting it according to the R/P-bit. When receiving a memory address, it must be determined whether the microcode is already cached in the ρ-control store or not. This is done by checking the residence table (see Figure 2.5) which stores the most frequently used translations of memory addresses into ρ-control store addresses and keeps track of the validity of these translations. It can also store other information: least recently used (LRU) and possibly additional information, e.g., required for virtual addressing[1] support. In the case that a memory address is received and the associated microcode is not present in the ρ-control store, the $\rho\mu$-code unit initiates the loading of microcode from the memory into the ρ-control store. In the case a ρCS-α is received or a valid translation into a ρCS-α is found, the ρCS-α is transferred to the 'determine next microinstruction'-block. This block determines the next microinstruction to be executed:

[1] For the simplicity of the discussion, we assume that the system only allows real addressing.

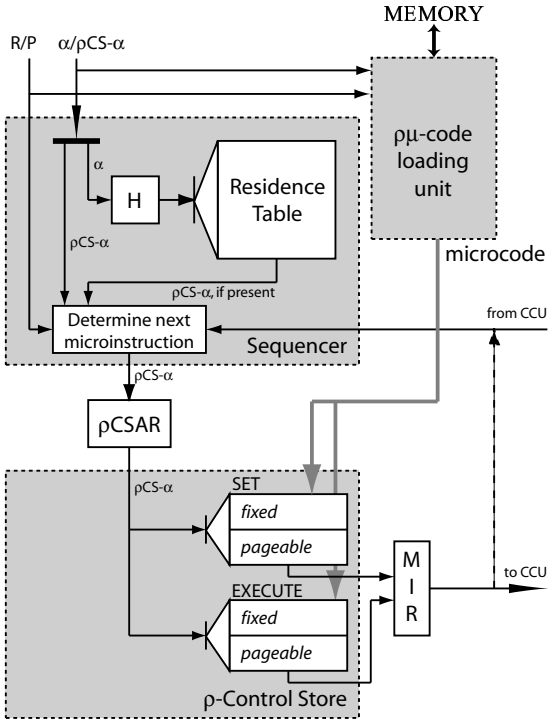


Figure 2.4: $\rho\mu$-code unit internal organization.

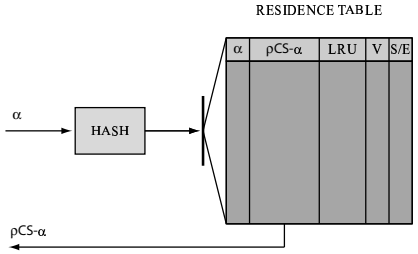


Figure 2.5: The sequencer residence table.

- When receiving the address of the first microinstruction: Depending on the R/P-bit, the correct ρCS-α is selected, i.e., from the instruction field or from the residence table.
- When already executing microcode: Depending on previous microinstruction(s) and/or results from the CCU, the next microinstruction address is determined.

The resulting ρCS-α is stored in the ρ-control store address register (ρCSAR) before entering the ρ-control store. Using the ρCS-α, a microinstruction is fetched from the ρ-control store and then stored in the microinstruction register (MIR) before it controls the CCU reconfiguration or before it is executed by the CCU. The ρ-control store comprises two sections, namely a *set* section and an *execute* section. Both sections can be identical, probably differing in microinstruction word sizes only. Each sections is further divided into a *fixed* part and a *pageable* part. The fixed part stores the resident reconfiguration and execution microcode of the *set* and *execute* phases, respectively. Resident microcode is commonly used by several invocations (including reconfigurations) and it is stored in the fixed part so that the performance of the *set* and *execute* phases is possibly enhanced. Which microcode resides in the fixed part of the ρ-control store is determined by performance analysis of various applications and by considering various software and hardware parameters. Other microcode is stored in memory and the pageable part of the ρ-control store acts like a cache to provide temporal storage. Cache mechanisms are incorporated into the design to ensure the proper substitution and access of the microcode in the ρ-control store. This is exactly what is provided by the residence table which invalidates entries when microcode has been replaced (utilizing the valid (V) bit) or substitutes the least recently used (LRU) entries with new ones. Finally, the residence table can be separate or common for both the set and execute pageable ρ-control store sections. In assuming a common table implementation, an additional bit needs to be added to determine which part of the pageable ρ-control store is addressed (depicted as the S/E-bit in Figure 2.5).

The arbiter performs partial decoding of instructions in order to determine where instructions should be issued. Its organization is of great importance for the time- efficient operation of the entire Molen organization. We will describe an arbiter implementation in Chapter 5 adding more substantial details.

2.3 Programming paradigm and sequence control

In the previous section, we briefly introduced the Molen organization and microarchitecture. In this section, we present the Molen programming paradigm [42], the instruction set architecture (ISA) that supports it, and the program sequencing required to implement this programming paradigm.

The Molen programming paradigm is a sequential consistency paradigm targeting the previously described organization, which allows parallel and concurrent hardware execution. Further in our discussion and experiments, we will

assume that the programming paradigm is intended for single program execution, which is not its general limitation, however. The Molen programming paradigm requires only a one-time architectural extension of few instructions to provide a large user reconfigurable operation space. The complete list of the eight required instructions, denoted as polymorphic ($\pi o \lambda \nu \mu o \rho \phi \iota \kappa \acute{o}$) instruction set architecture (πISA), is as follows:

Six instructions are required for controlling the reconfigurable hardware:

- Two **set** instructions: these instructions initiate the configurations of the CCU. When assuming partial reconfigurable hardware, we provide two instructions for such purpose, namely:
  - the partial set (p-**set** $<address>$) instruction performs those configurations that cover common and frequently used functions of an application or set of applications. In this manner, a considerable number of reconfigurable blocks in the CCU can be preconfigured.
  - the complete set (c-**set** $<address>$) instruction performs the configurations of the remaining blocks of the CCU (not covered by the p-set). This *completes* the CCU functionality by enabling it to perform the less frequently used functions. Due to the reduced amount of blocks to configure, reconfiguration latencies can be reduced.

  We must note that in case no partial reconfigurable hardware is present, the c-**set** instruction alone can be utilized to perform all configurations.

- **execute** $<address>$: this instruction controls the execution of the operations implemented on the CCU. These implementations are configured onto the CCU by the set instructions.

- **set prefetch** $<address>$: this instruction prefetches the needed microcode responsible for CCU reconfigurations into a local on-chip storage facility (the $\rho\mu$-code unit) in order to possibly diminish microcode loading times.

- **execute prefetch** $<address>$: the same reasoning as for the **set prefetch** instruction holds, but now relating to microcode responsible for CCU executions.

- **break**: this instruction is utilized to facilitate the parallel execution of both the reconfigurable processor and the core processor. More precisely, it is utilized as a synchronization mechanism to complete the parallel execution.

Two $move$ **instructions** for passing values between the register file and exchange registers (XREGs) since the reconfigurable processor is not allowed direct access to the general-purpose register file:

- **movtx** $\text{XREG}_a \leftarrow \text{R}_b$: (move to XREG) used to move the content of general-purpose register R_b to XREG_a.
- **movfx** $\text{R}_a \leftarrow \text{XREG}_b$: (move from XREG) used to move the content of exchange register XREG_b to general-purpose register R_a.

The $<address>$ field in the instructions introduced above denotes the location[2] of the reconfigurable microcode responsible for the configuration and execution processes, described in Subsection 2.2. It must be noted that a single address space is provided with at least $2^{(n-op)}$ addressable functions, where n represents the instruction length and op the opcode length.

It should be noted that it is not imperative to include all instructions when implementing the Molen organization. The programmer/implementor can opt for different ISA extensions depending on the required performance to be achieved and the available technology. There are basically three distinctive πISA possibilities with respect to the Molen instructions introduced earlier - the *minimal*, the *preferred* and the *complete* πISA extension. In more detail, they are:

- **the minimal πISA:** This is essentially the smallest set of Molen instructions needed to provide a working scenario. The four basic instructions needed are **set** (more precisely: c-**set**), **execute**, **movtx** and **movfx**. By implementing the first two instructions (**set**/**execute**) any suitable CCU implementation can be loaded and executed in the RP. Furthermore, reconfiguration latencies can be hidden by scheduling the **set** instruction considerably earlier than the **execute** instruction. The **movtx** and **movfx** instructions are needed to provide the input/output interface between the RP targeted code and the remainder application code.
- **the preferred πISA:** The minimal set provides the basic support, but it may suffer from time-consuming reconfiguration latencies, which could not be hidden, and that can become prohibitive for some real-time applications. In order to address this issue, two **set** (p-**set** and c-**set**) instructions are utilized to distinguish among frequently and less frequently used CCU functions. In this manner, the c-**set** instruction only configures a smaller portion of the CCU and thereby requiring less reconfiguration time. As the reconfiguration latencies are substantially (or

[2] Indirect pointing could be required in order to extend the $\rho\mu$-code addressing space.

completely) hidden by the previously discussed mechanisms, the loading time of microcode will play an increasingly important role. In these cases, the two **prefetch** instructions (**set prefetch** and **execute prefetch**) provide a way to diminish the microcode loading times by scheduling them well ahead of the moment that the microcode is needed. Parallel execution is initiated by a πISA **set**/**execute** instruction and ended by a general-purpose instruction.

- **the complete πISA:** This scenario involves all πISA instructions including the **break** instruction. In some applications, it might be beneficial performance-wise to execute instructions on the core processor and the reconfigurable processor in parallel. In order to facilitate this parallel execution, the preferred ISA is further extended with the **break** instruction. The **break** instruction provides a mechanism to synchronize the parallel execution of instructions by halting the execution of instructions following the **break** instruction. The sequence of instructions performed in parallel is initiated by an **execute** instruction. The end of the parallel execution is marked by the **break** instruction. It indicates where the parallel execution stops.

The exchange registers: The exchange registers (XREGs) are used for passing operation parameters to the reconfigurable hardware and returning the computed values after operation execution. In order to avoid dependencies between the reconfigurable processor and the core processor, the needed parameters are moved from the register file to the XREGs (**movtx**) and the results stored back in the register file (**movfx**). During the execute phase, the defined $\rho\mu$-code is responsible for taking the parameters of its associated operation from the XREGs and returning the result(s). A single **execute** instruction does not pose any specific challenge, because the complete set of exchange registers is available. When executing multiple **execute** instructions simultaneously, overlapping utilization of the available XREGs has to be avoided. This assumes an agreement on the conventions for the XREGs allocation.

Parameter exchange, parallelism and modularity: As shown earlier, the exchange registers solve the limitation on the number of parameters as present in other reconfigurable computing approaches (e.g., [36, 37]). If the parameters do not exceed the number of XREGs, parameters are passed by value, otherwise - by reference. This allows an arbitrary number of parameters to be exchanged between the calling (software) and called (hardware) functions, where only the hardware resources determine the upper bound. The Molen architecture also addresses an additional shortcoming of other reconfigurable

computing approaches concerning parallel execution. In case that two or more functions considered for CCU implementation do not have any true dependencies, they can be executed in parallel. There is always a physical maximum of how many operations can be executed in parallel on the CCU. This is, however, an implementation dependent issue, e.g., reconfigurable hardware size, number of XREGs, etc. and can not be considered as a limitation of the Molen architecture. In addition, it should be emphasized that the Molen hardware/software (HW/SW) division ability is not limited to functions only. In case the targeted kernel is part of a function, e.g., a highly computational demanding loop, it can be appropriately transformed for use in the Molen programming paradigm by defining a clear set of interface parameters and passing them via the XREGs (as values or references) to the CCU implementation of the kernel.

The Molen paradigm facilitates modular system design. For instance, hardware implementations described in an HDL (VHDL, Verilog or System-C) language are mappable to any FPGA technology, e.g., Xilinx or Altera, in a straightforward manner. The only requirement is to satisfy the Molen set and execute interface. In addition, a wide set of functionally similar CCU designs (from different providers), e.g. sum of absolute differences (SAD) or IDCT, can be collected in a database allowing easy design space explorations.

Interrupts and miscellaneous considerations: The Molen approach is based on the GPP co-processor paradigm (see for example [43–45]). Consequently, all known co-processor interrupt techniques are applicable. In order to support the core processor interrupts properly, the following parts are essential for any Molen implementation:

1. Hardware to detect interrupts and terminate the execution before the state of the machine is changed are assumed to be implemented in both core processor and reconfigurable processor.

2. Interrupts are handled by the core processor. Consequently, hardware to communicate interrupts to the core processor is implemented in CCU.

3. Initialization (via the core processor) of the appropriate routines for interrupt handling.

It is assumed that the implementor of a reconfigurable hardware follows a co-processor type of configuration. With respect to the GPP paradigm, the FPGA co-processor facility can be viewed as an extension of the core processor architecture. This is identical with the way co-processors, such as floating point, vector facilities, etc., have been viewed in the conventional architectures.

2.4 Conclusions

Many current reconfigurable proposals fall short of expectations due to a number of shortcomings, the most essential of which are opcode space explosion, lack of ISA compatibility, technology dependence, no design modularity, limited number of processing parameters, and no support for parallel reconfigurable execution. All these critical issues have been addressed and successfully solved by the Molen polymorphic processor paradigm. *The basis of the Molen processor is established on the capability to control program execution and hardware reconfiguration allowing intermingling of program code and hardware description and by utilizing emulation microcode.* This special microcode is termed as reconfigurable microcode ($\rho\mu$-code) as it is different from the traditional one. The difference is that instead of executing on fixed hardware facilities, the $\rho\mu$-code itself "designs" the facilities to operate upon. The main advantages of the Molen approach can be summarized as follows:

- **Compact ISA extension.** For a given ISA, a single architectural extension comprising 4 to 8 additional instructions provides unlimited number of reconfigurable functionalities per single programming space. This realization is application independent and resolves the opcode space explosion problem as well as provides ISA compatibility and portability of reconfigurable programs.
- **Technology independent and modular design.** The design concept is not bound to any particular reconfigurable technology. It allows reconfigurable modules designed by a third party to be ported easily into the Molen organization.
- **Arbitrary number of parameters and parallel processing.** The Molen processor organization and the programming paradigm based on sequential consistency allow an arbitrary number of parameters as well as parallel executions of no data dependent operations.

In the chapter to follow, we are going to present some computationally demanding MPEG-4 specific hardwired kernels. These kernels will be considered for CCU implementations in a Molen prototype design, described in Chapter 5, and the experimental overall performance gains induced will be evaluated in Chapter 6.

Chapter 3

MPEG-4 Hardwired Kernels

Regardless the particular technology a Molen processor (described in the preceding chapter) is implemented in, a number of computationally demanding kernels has to be considered for custom computing units (CCUs) designs to increase processor performance. Such designs should be capable of accelerating the considered kernels in reconfigurable hardware with respect to their pure software execution. In this chapter, we consider three computationally demanding MPEG-4 kernels for hardware implementation. The considered kernels are defined in the higher profiles of MPEG-4. This chapter emphasizes on the technology independence of the proposed designs, therefore they are evaluated as stand-alone units both for reconfigurable and ASIC realization. Multiple reconfigurable technologies by the two major FPGA vendors Altera and Xilinx are considered. When we evaluate the Molen prototype in Chapter 6, where entire programs are implemented, we consider the Xilinx Virtex II Pro technology.

We start with two hardwired solutions for *MPEG-4 repetitive padding*, a performance restricting algorithm for real time MPEG-4 execution. The first solution regards application specific implementations, the second - general purpose processing. For the application specific implementations, we propose a systolic array structure and it is shown that the requirements of all MPEG-4 visual profile levels can be met. The second approach regards an augmentation of a general purpose ALU with an extra functionality added to perform repetitive padding. This approach allows the MPEG-4 repetitive padding algorithm to run in real-time at an order of magnitude higher speeds than the requirements of the most-demanding MPEG-4 visual profile levels. Speed and hardware estimations are reported for 32, 64, and 128-bit augmented ALUs. The trade-offs

between chip-area and performance are discussed and possible configuration alternatives are proposed for both of the proposed padding structures.

Another considered candidate for hardware acceleration in MPEG-4 is the *ACcepted Quality Function* (ACQ), which determines whether the encoding of the shape of an object gives an accepted quality according to some specified lossy coding conditions. We propose a hardware implementation of the ACQ function as a *systolic structure of processing elements*. The structure is scalable and can be implemented according to different memory bandwidth restrictions. The ACQ function can be implemented either as a reconfigurable or as a hardwired unit in a dedicated MPEG-4 processor.

An important function in MPEG-4 (but also in other recent standards, e.g., JPEG2000) is the Discrete Wavelet Transform (DWT). At algorithmic level, the so called *lifting scheme* represents the fastest implementation of the DWT. In this chapter, we propose a lifting-based DWT hardware module design, which is mapped on the Xilinx Virtex II FPGA. To accelerate the transform, we introduce pipelining, data reusability and data parallelism. The design is scalable, allowing more dramatic improvements in performance to be achieved at higher degrees of parallelism.

It should be noted, that we also consider some extensively investigated MPEG-2 kernels, such as SAD, DCT, and IDCT, which are also parts of MPEG-4. The latter kernels have been implemented in the Molen prototype and their overall impact is evaluated in Chapter 6.

The remainder of this chapter is organized as follows. The alternatives for MPEG-4 repetitive padding hardware designs are described in Section 3.1. In Section 3.2, the ACQ function design is presented. The lifting based DWT hardware implementation is introduced in Section 3.3. Finally, the chapter is concluded with Section 3.4.

3.1 Hardwired repetitive padding

This section addresses one important feature in MPEG-4, the *repetitive padding* technique, defined at all Levels in the Core and Main Profiles of the standard. Software profiling results, reported in [7, 46, 47], indicate that padding is a computationally demanding and time consuming process, which restricts the real time operation of the MPEG-4 codecs. We present two general hardware approaches to implement the repetitive padding algorithm in real time. The first approach assumes MPEG-4 application specific process-

ing (ASIP) designs. It can be used as a hardware accelerator for an ASIP MPEG-4 processor or reconfigurable processing [13,16,39,48–50].The second approach aims at (hardware) augmentations of general-purpose Arithmetic-Logical-Units (ALU) with application specific functional extensions. We show that both of the approaches are beneficial for improving the execution of the repetitive padding at little cost. More specifically, the following is shown regarding performance and cost:

- **Performance** - real time processing for all MPEG-4 profiles and levels, utilizing the padding algorithm can be achieved. More specifically it is shown that: Assuming available technologies, data processing rates from 77 K up to 280 K macroblocks per second (MB/s) are achieved employing 4 - 16 processing elements mapped on the xc4085xplg559-09 Xilinx FPGA and the epf10k20rc240-4 Altera FPGA. We show that even higher processing speeds are achievable when more processing elements are implemented (e.g. a 64 processing elements structure processes 950 K MB/s). It is also established that the required operating speed is low [1]. For example the 16-pixel line processing FPGA implementations produce the wanted results in a frequency varying between 11 and 25 MHz. For a 64-bit augmented ALU example, running at a frequency of 1 GHz, we achieve 7.8 million MB/s, which also allows the MPEG-4 repetitive padding to run at real time for all profiles and levels it is defined in.
- **Hardware costs** - we establish that scalable implementations, tunable to the different Profiles@Levels requirements are feasible and show that: To achieve the performance discussed in the previous paragraph, a low number of FPGA cells (419 Xilinx CLBs and 1024 Altera LCs) is required for a 16-pixel processing unit. Only 344 AND-OR gates extra hardware penalty costs are required for a 64-bit padding-aware ALU augmentation example. We also show that the 32 and 128-bit implementation costs are 172 and 688 extra AND-OR gates respectively.

We also note that the proposed approaches are general and can be utilized in different architectures and implementations. The remainder of the discussion in this section is organized as follows. In Subsection 3.1.1 we give some background knowledge and detailed motivation for our research. Subsection 3.1.2 describes in details the ASIP padding structure. The general purpose ALU padding augmentation is presented in Subsection 3.1.3. Subsection 3.1.4 gives

[1] We distinguish (data) processing speed, measured in [macroblocks/sec] (or [MB/s]) from the device operating speed (frequency), measured in [Hz].

quantitative evaluations of both approaches and presents analytical and simulation results in numbers.

3.1.1 Background and motivation

Like its preceding visual data compression (MPEG) standards, MPEG-4 adopts motion estimation and motion compensation techniques to exploit temporal redundancies in the encoded video sequences. In MPEG-4, motion estimation and compensation are defined over VOPs instead of frames.

The repetitive padding algorithm: For more accurate block matching in motion compensation/ decompensation of VOPs, MPEG-4 adopts the padding process. *The purpose of padding in MPEG-4 is to ensure more accurate block matching in motion compensation algorithms for arbitrary shaped visual objects.* The padding process defines the full-color values (luminance + chrominance) for pixels outside the shape of a VOP. In padding, two types of macroblocks are of interest. Macroblocks, which lie on the boundary of the VOP are referred to as boundary blocks and are processed with *repetitive padding*. Exterior macroblocks (completely outside the VOP) are padded using the *extended padding method.* Since repetitive padding is the most demanding padding algorithm, in this section we will consider the padding of boundary macroblocks. The standard repetitive padding algorithm, as defined in [1], is equivalent to the following steps:

- ***Step 1.*** *Initialization.* Define any pixel outside the object boundary as a zero pixel. Make a duplicate binary alpha map.
- ***Step 2.*** *Horizontal Repetitive Padding.* Scan each horizontal line of a block. Each scan line is composed of *zero* and *nonzero* line segments (according to the shape bits in the binary alpha map).
  - In zero segments, between an end point of the scan line and the end point of a nonzero segment, all zero pixels are replaced by the pixel value of the end pixel of nonzero segment.
  - In zero segments, between the end points of two different nonzero segments, all zero pixels take the average value of these two end points.

  Nonzero segments are not processed. All shape bits, corresponding to padded pixels are set in the duplicate binary alpha map.

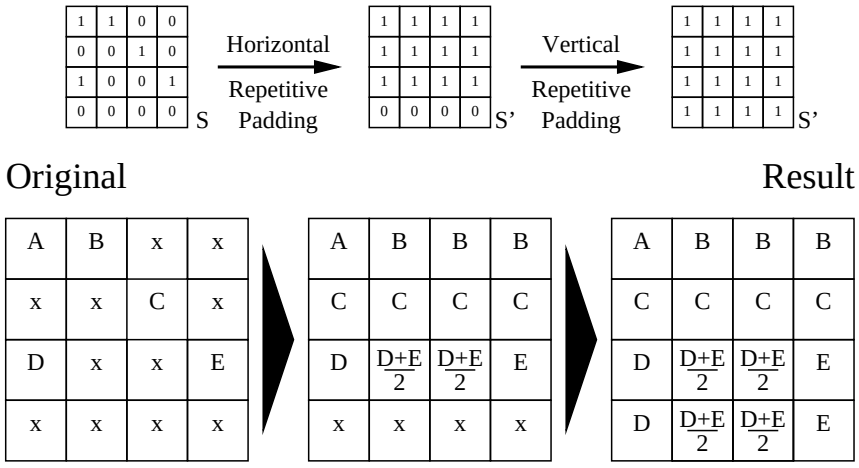


Figure 3.1: The repetitive padding algorithm.

- ***Step 3.*** *Vertical Repetitive Padding.* Scan each vertical line of the block and perform the identical procedure as described for the horizontal line. The updated shape information from the duplicate binary alpha map is used.

Figure 3.1 illustrates the repetitive padding algorithm with a simplified examples of a 4x4 pixel BAB and a 4x4 pixel luminance block. The original data structures are in the left part of the Figure, where the definition of the zero and non-zero pixels is depicted according to Step 1. The luminance block contains color values (in the Figure these values are indicated by $\{A,B,C,D,E\}$) for the pixels, belonging to the shape of the VOP (non-zero pixels) and $\{x\}$ value (don't care) for the rest (zero pixels). The central two squares of Figure 3.1 illustrate the resulting data after the horizontal repetitive padding (Step 2). The duplicate BAB is indicated by S' and the 4x4 luminance block is padded accordingly. In this part, the example illustrates both cases, mentioned in Step 2,i.e., replicating a boundary pixel and estimating the average of two boundary pixels. Finally (Step 3), the vertical repetitive padding is performed, identically to the horizontal, and the resulting BAB and luminance blocks are shown in the right-most area of Figure 3.1. The same procedure (Steps 1 trough 3) is executed for each of the two chrominance blocks from the padded macroblock, as well.

After the boundary macroblocks are repetitively padded vertically and horizontally, the *extended padding* is performed on the blocks immediately next to

the boundary macroblocks. The extended padding algorithm, however, simply replicates the boundary pixel of an adjacent, padded macroblock, which implies a simple pixel copying. Because of the low complexity of this data processing, we are not discussing extended padding further.

Motivation: In this section we advocate hardwired solutions for repetitive padding. The rational behind such a proposal is as follows: A summary of the computational complexity of the QCIF, Core Profile Level 1 of MPEG-4 is reported in [7]. Since this is the lowest profile level, utilizing the padding algorithm, we shall consider its real-time requirements as the minimum for a hardware implementation. At this level, the computational power (reported in [7]) for the software encoding of a single object is in the order of 4500 Million (RISC-like) Instructions Per Second (MIPS). Assuming a software performance optimization by a factor of up to 10 (assumed to be feasible in [7]), the total computational complexity is within the computational capabilities of the contemporary general purpose processors (500-1000 MIPS). In the case of 4 video objects (see Table 1.1), however, the real-time software feasibility becomes problematic with its requirements of approximately 4 times higher computational workload. Given the above considerations, the need of a hardware acceleration of MPEG-4 is evident, even at this low profile level. Further analysis of the requirements for the software implementation (see again [7]) indicates that the padding algorithm occupies some 175 MIPS for a single video object, or around 700 MIPS for the maximum 4 video objects, stated at Level 1 of the Core profile (Table 1.1). Considering Table 1.1, we can estimate that the required speed of 5940 MB/s for the Core Profile Level 1 is approximately 82 times lower than the speed requirements of the highest - Main@Level4 Profile (489600 MB/s). A simple arithmetic estimation, based on Table 1.1, indicates that for the highest MPEG-4 profile level, the non-optimized software padding would require approximately 57 000 MIPS and when extremely optimized (i.e., 10 times speed-up) - in the order of 6000 MIPS. Even for the significantly less complex decoder part of MPEG-4, the padding algorithm will require some 24 000 MIPS for non-optimized software implementation down to 2500 MIPS in dramatically optimized programming. All these approximated estimations of the MPEG-4 requirements are systematized in Table 3.1, after considering the data from Table 1.1 and [7].

3.1.2 The application specific processor approach

Since padding is performed over horizontal and vertical pixel lines in identical manner, we propose a scalable systolic structure to process pixel blocks per

Table 3.1: Computational demands of the MPEG-4 Core@L1 and Main@L4.

Profile	**MPEG-4 Algorithm**	**# VO**	**Boundary MB/s**	**Software MIPS Requirements**
Core@L1	All MPEG-4 Algorithms	1	742	4 500
		4	2 970	18 000
	Repetitive Padding	1	742	175
		4	2 970	700
Main@L4	Repetitive Padding	1	7 650	1 794
		32	244 800	57 400

line basis. Consequently, we define an elementary processing element (PE) and a topology to connect functional groups of processing elements.

The processing element. A single processing element (PE), which is dedicated to process each pixel of a block, is depicted in Figure 3.2. The same processing element is used for luminance and chrominance padding. The following equations describe the functionality of the processing element:

$$OS = (S \vee \overline{LI_N \vee RI_N}), \tag{3.1}$$

$$O = OS \wedge I \vee \overline{OS} \wedge [(LI_{\overline{N}} + RI_{\overline{N}}) >> i], \tag{3.2}$$

$$i = LI_N \wedge RI_N,$$

$$LO = S \wedge |\overline{S, I}| \vee \overline{S} \wedge RI, RO = S \wedge |\overline{S, I}| \vee \overline{S} \wedge LI, \tag{3.3}$$

$$S' = S \vee LI_N \vee RI_N; \tag{3.4}$$

where $\vee$ and $\wedge$ represent logical *OR* and *AND* operations respectively, overline stands for logical negation, $A >> i$ denotes "shift i positions right the binary vector A", and + is an arithmetic summation of binary vectors,
OS stands for Output Select signal,
N represents the width of the processed data (further, we assume N=8),
LI, RI are left and right input vectors with width N+1,
LO, RO are left and right output vectors with width N+1,
I, O are data input and output vectors with width N,
S is the shape (input) bit before processing,
S' is a mask output bit after processing,
$LI_{\overline{N}}$ denotes the first N (least-significant) bits of LI (bits 0 to N-1),

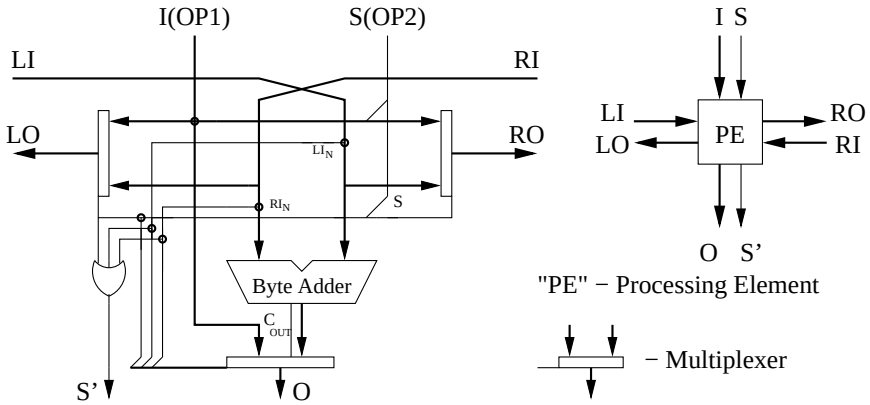


Figure 3.2: The padding processing element.

LI_N represents the N^{th} (the most-significant) bit of LI,
and $\overline{|S, I|}$ denotes the concatenation of bit S and vector I.

The operation of the PE is explained by the following:

- If the input shape bit S is set (the pixel belongs to the object), then:
  - The output O takes the value of the input I, i.e. the pixel keeps its color.
  - The value of the input (pixel) I is propagated to the left and to the right (via outputs $LO_{\overline{N}}$ and $RO_{\overline{N}}$) for further processing. The shape input bit S is propagated by the same multiplexers and occupies the most-significant bits of LO and RO.
  - The output bit $S^{'}$ is set, meaning the pixel has been processed.
- If the input shape bit S is zero (the pixel does not belong to the object and has to be padded), then:
  - The output O takes the average value of the $LI_{\overline{N}}$ and $RI_{\overline{N}}$ inputs, i.e. the pixel takes the padded value.
  - The LI value is propagated via RO and the RI - via LO including color and shape information.
  - The output bit $S^{'}$ is set, meaning the pixel has been processed.

The systolic structure: To process a line from a macroblock, we implement the systolic structure of processing elements, depicted in Figure 3.3. For the

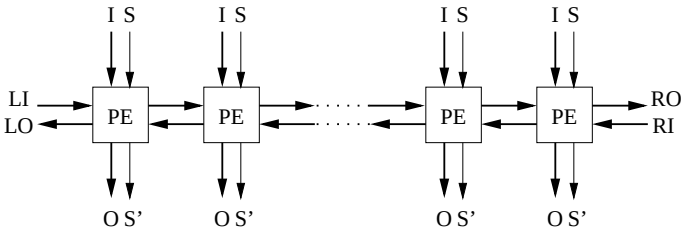


Figure 3.3: A single scan line/column padding structure.

proper circuit operation, the left-most and right-most inputs of the structure should be initialized with zero vectors. This would mean that there are no pixels to the left and to the right of the macroblock, which could influence the padding values. This structure is scalable and can contain an arbitrary number of processing elements. Since a macroblock consist of one 16x16 luminance and two 8x8 chrominance blocks, it is efficient to implement structures of 8 or 16 PE. Furthermore, it is possible to implement several structures, identical to the one in Figure 3.3. For example, if we implement eight such structures, we will be able to process eight lines in parallel. This is possible, because in the padding algorithm there is no data dependency between any two lines or columns. The data dependency is just between the pixels in the same line/column. Even a larger, two dimensional structure for processing an entire block in parallel is implementable. Implementations, which process more than one macroblock lines in a time, however, require higher data throughput and the utilization of more complicated addressing approaches would become necessary.

We can easily evaluate the processing speed of the structure, given its operating frequency. Let us assume a chain of *n* PE as depicted in Figure 3.3, operating at frequency F_n [Hz]. Assume the value of *n* having practical significance to be 4, 8, 16. Further assume two more parameters of the particular implementation, denoting the following: $\mathbf{N_n^{P8}}$ *and* $\mathbf{N_n^{P16}}$ *regard the numbers of cycles, necessary to process an 8-pixel (chrominance) and a 16-pixel (luminance) line respectively.* Some potential values of these parameters are shown in Table 3.2. The processing of 16 pixels by any nPE configuration will take $\frac{N_n^{P16}}{F_n}$ [seconds] and for a 256-pixel luminance block- $\frac{16 \cdot N_n^{P16}}{F_n}$ [s]. Identically, the processing of two 8x8-pixel chrominance blocks will take $\frac{16 \cdot N_n^{P8}}{F_n}$ [s] to the same unit configuration. Since a macroblock consists of 256 luminance and 128 (2 x 64) chrominance pixels, padded vertically and horizontally, a whole macroblock will be padded for $\frac{32}{F_n} \cdot (N_n^{P8} + N_n^{P16})$ [s]. If we implement a

configuration, which processes two and more (say *k*) 16-pixel lines in a time, we can formulate the *Processing speed* as follows:

$$Processing\_speed = \frac{F_n \cdot k}{32 \cdot (N_n^{P8} + N_n^{P16})} \tag{3.5}$$

Formulation (3.5) is still valid for $n < 16$, assuming in this case that *k*=1.

Table 3.2: Values of N_n^{P8} and N_n^{P16}.

n	**Average**		**Worst Case**	
	N_n^{P8}	N_n^{P16}	N_n^{P8}	N_n^{P16}
4	2.5	5.5	3	7
8	1	2.5	1	3
16	0.5	1	0.5	1
16·k	0.5	1	0.5	1

In Table 3.2 we separate the values for each of the parameters into two groups, namely: average values and worst case values. The numbers represent the count of processing cycles at operating frequency F_n for different numbers of processing elements (column "n"). The cycle count N_n^{P8} is unproportionally greater for chrominance line padding when $n < 8$ compared to the case when $n \geq 8$. Identical is the case with the cycle count N_n^{P16} when $n < 16$ compared to the case when $n \geq 16$. The reason for this is that when we use structures with less processing elements, we cut the data propagation chain within the line to be padded by dividing this line into sublines to be processed. In such cases extra processing cycles are required to complete the computations, because padding is highly data dependent regarding the data within a line (column). For example, if we assume padding of a chrominance line (8 pixels) and a 4 PE structure, we will be required at most 3 cycles. In the first cycle we pad the first 4 pixels, but in this cycle we may need data from the second 4 pixels of the line. Therefore we propagate the padding result from the first 4 pixels to the the second cycle, in which we already have all the required data to pad the second 4 pixels. The result of this second cycle can already be used to pad the first 4 pixels of the line. Note that within these three cycles any chrominance line can be padded, regardless the pixel data it contains and no more cycles are required. Therefore we denote this cycle number as the worst case (see Table 3.2). Further cycle reduction can be done by analyzing the last shape bits of the sublines being padded. In the last example, if the right-most shape bit of the first 4 pixels of the chrominance line is 1, the subline need not be processed

again in a third cycle, because no data from the second subline is required. Therefore two cycles would be enough to pad the whole chrominance line and we can estimate the required number of cycles for padding a chrominance line with 4 PE to be on average 2.5 (Table 3.2). The timing of such a padding scheme is discussed in more details in the section to follow and Figure 3.7(b).

Possible configurations: For the line/column repetitive padding unit, there could be several configuration options. We report three possibilities:

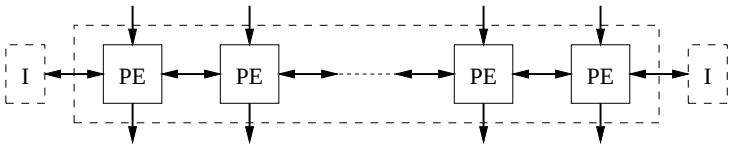
1. 16 PE unit - processes one luminance line/column and two chrominance per operating cycle.
2. 8 PE or 4 PE unit - processes a half/quarter of luminance and one/half chrominance line/column. An additional control circuit is required, to maintain intermediate computational results.
3. 32, 64, ..., 256 PE unit- processing more than two luminance and more than four chrominance lines/columns per operating cycle. The extreme configuration would process the whole macroblock.

Figure 3.4 depicts a general view of the discussed possible configurations of the padding unit. The blocks, named ”I” are buffers which store the Initialization values for configurations, containing multiple of 16 PE. For Configurations with less than 16 PE, these buffers are used for storing intermediate values, in order not to cut the data propagation chain for longer (up to 16 pixel) lines. The buffering organization and timing is discussed in more details in the section to follow.

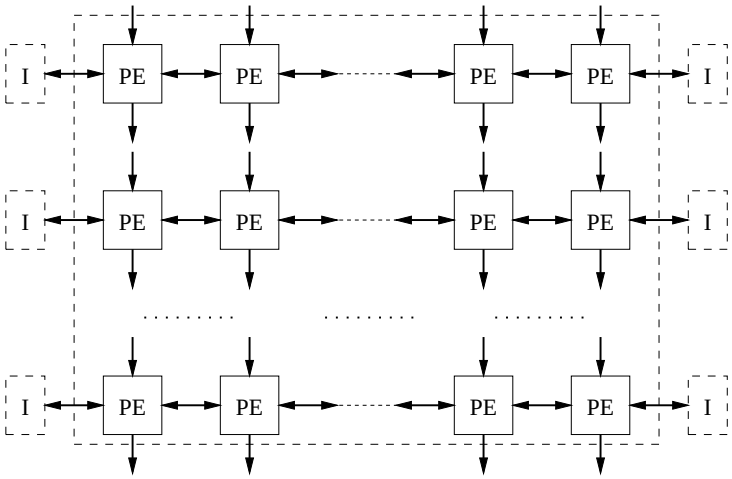
3.1.3 The augmented ALU

In this section we describe a second approach for repetitive padding. We consider a general purpose ALU, augmented to support repetitive padding. To accommodate padding we consider sub-word data parallelism. Our general assumption is: *accommodate padding without creating critical ALU paths and preserve the ALU functionality.* Since 8-bit integer chrominance and luminance data representations are frequently used, we assume the same data formats (our scheme with proper considerations will accommodate ”N-bit” quantities as stated in MPEG-4, N being 10, 12 etc.).

Pixel processing: A single byte processing structure, which is dedicated to process each pixel of a block, is depicted in Figure 3.5. Its organization is similar to the organization reported in Figure 3.2, but it is extended with additional



(a) Up to 16 PE;



(b) 32, 64,..., 256 PE.

Figure 3.4: Possible configurations - ”I” denotes initialization and/or intermediate result buffer.

pipelining to fit into the general purpose ALU cycle. We pipeline the processing flow by dividing it into two stages. The first stage contains a Propagation Node (PN) and two multiplexers. The multiplexers are required to preserve the original functionality of the ALU. A byte controlled adder and an output multiplexer build the second pipeline stage. The byte controlled adder is a part of the original multi-byte ALU adder, with controllable carries between the bytes. The padding output multiplexer can be merged with the existing ALU output multiplexer and that is depicted in Figure 3.5 by the dash-lined arrow, from the logical part of the ALU (LU). The function of the PN is to propagate the appropriate values to its adjacent padding PNs and to supply data and control signals to the byte controlled adder and the output multiplexer. Its functionality can be described by Equations (3.3) and (3.4). Table 3.3 represents the control signals of the output multiplexer (SR($\overline{C_{OUT}, ADD}$) means Shift Right with 1-bit the ADDer output together with its carry). Signal ”Control” determines whether the ALU will perform padding, or its original operation(s).

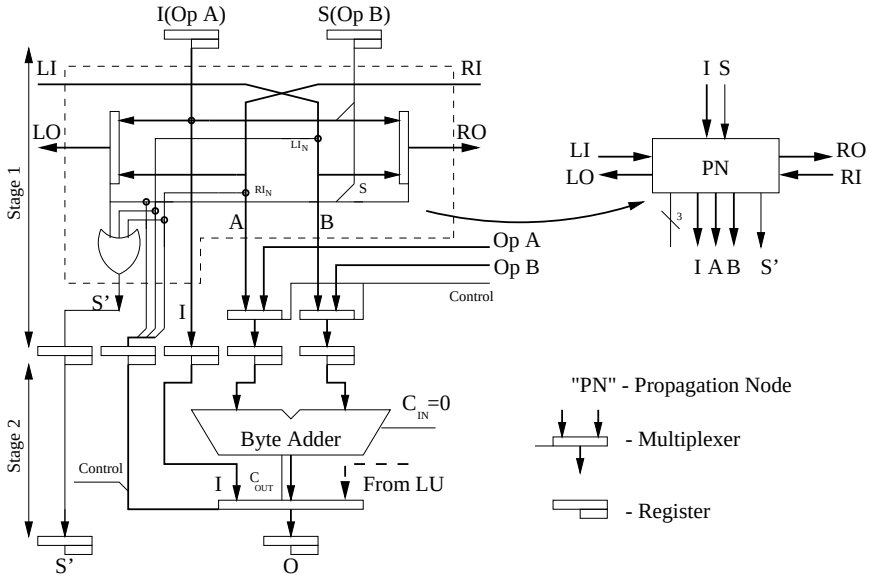


Figure 3.5: ALU augmentation for a single pixel padding.

Table 3.3: Truth table for the control signals of the output multiplexer.

Control	S	RI_N	RI_N	O
0	X	X	X	ADD
1	0	0	0	I
1	0	0	1	ADD
1	0	1	0	$\overline{\text{ADD}}$
1	0	1	1	SR($\overline{C_{OUT}}$, $\overline{ADD}$)
1	1	X	X	I

Line / column padding: To process a line or a column from a block by an n-byte ALU, we have to implement a chain of n PN (i.e. n-pixel parallel padding) similar to the structure in Figure 3.3. A section of such processing circuitry for two adjacent pixels is depicted in Figure 3.6. The added ALU logic has to perform the functions described in equations (3.1) - (3.4). Each node from the propagation chain propagates the pixel values from left to right and from right to left when its corresponding shape bit S is zero. When a shape bit is 1, the corresponding propagation node transmits the value of the pixel (driven via input I) to its adjacent propagation nodes. The proper multiplexer input is selected to drive the appropriate value out of the ALU. This structure is scalable and can contain an arbitrary number of propagation nodes, depending on the ALU width (i.e. n elements for an n-byte ALU).

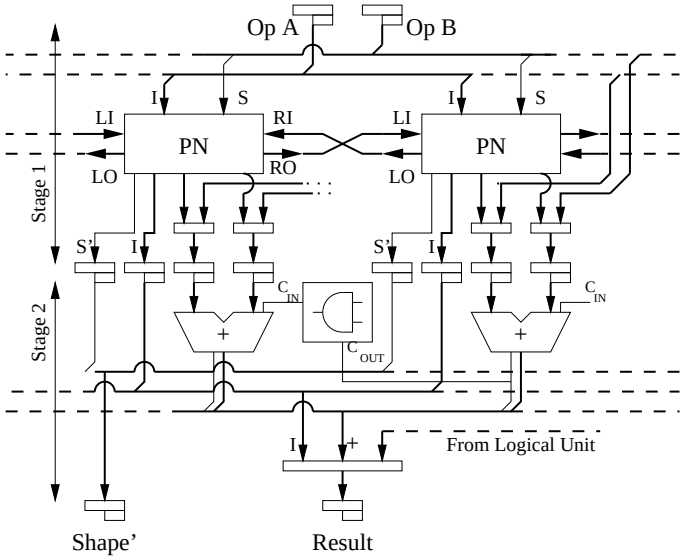
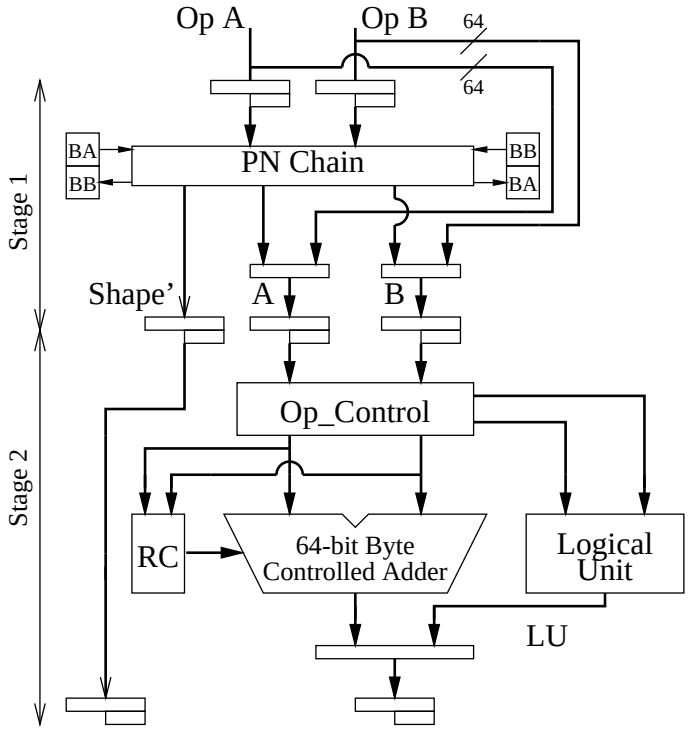
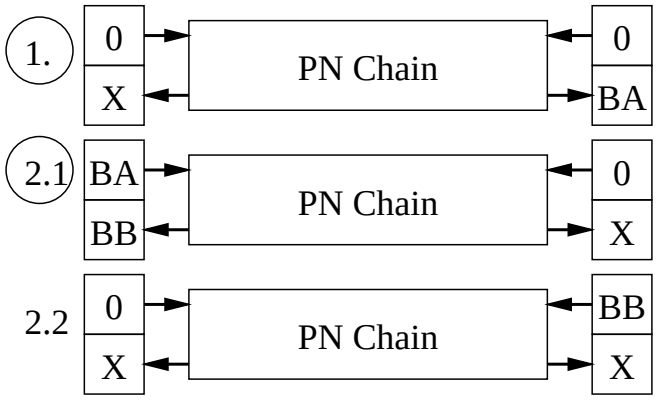


Figure 3.6: Scan line / column padding augmentation of an ALU.

Putting everything together: Since a macroblock consist of one 16x16 luminance and two 8x8 chrominance blocks, it is efficient to implement structures processing 8 or 16 pixels simultaneously. This means that with 64 or 128-bit ALUs we will be able to pad an entire line or column of chrominance and/or luminance blocks in pipeline manner having a pipeline length of two ALU cycles. The proposed structure is also capable of performing padding when implemented on smaller ALUs. In the remainder of the section, we will describe in more details the padding process flow, performed by a 64-bit ALU. Data buffering and initialization are identical for the application specific implementation, described earlier. First, for the proper circuit operation, the left-most and right-most inputs of the structure should be initialized. Figure 3.7(a) depicts a general view of the 64-bit initialization for luminance line / column padding. The operand control circuit (Op_Control) is a part of the operands critical path. It is responsible for setting the adder operands and performs operations like sign extension, operand masking etc. The result control (RC) circuit deals with flags handling like overflows, carries, equal zero etc. The BA and BB buffers are 8-bits wide and contain the initialization values, required by the unit to start the operation. Since a luminance line (16x8-bit) can not be processed in one pass by a 64 bit (8x8-bit) ALU, we assume that the left-most half (the left-most subline) of the line is processed first. In the previous section it has been noted that a full line can be padded in the worst case (maximal)



(a) General view of the pipeline stages (Op_Control- operand control, RC- result control, LU- logical unit);



BN - Buffer N: X-Don't care: 0-zero.

(b) Data buffering by cycles.

Figure 3.7: Data initialization and buffering for luminance line / column processing by a 64-bit ALU.

number of cycles, regardless the data dependency within the line. For the assumed ALU and a luminance line this number of cycles is 3. It has also been noticed that a cycle reduction by exploiting intraline data dependency is possible. The cycle partitioning of the luminance line padding in such a scenario is illustrated in Figure 3.7(b). Depending on the right-most shape bit of the subline, padded in the first cycle, the full-line padding would require one or two more cycles:

If the right-most shape bit of the first half of the luminance line is "0", it means that data from the other half of the line is required to pad the first half. Thus the data, propagated to the right is stored into BA register. In the next cycle this data is used to fully pad the second half of the line, and the byte, which has to be propagated to the left is stored in buffer BB. Now, a third cycle is executed on the first half of the luminance line, with the proper right-to-left propagation value stored in the BB buffer.

If the right-most shape bit of the first half of the luminance line is "1", the pixel value to be propagated right is stored into buffer BA, and the first half of the luminance line is completely padded. Just one more cycle is required to pad the second half of the luminance line, provided the right propagation value is driven to the left input of the circuit from BA.

Since the right-most shape bit of the first half of the luminance line is available before the next operands (describing the other half of the line) are issued, we have branch determination (a perfect branch prediction) [51] for the pipeline. Since the PN structure is similar to the application specific PE, Equation (3.5) is valid for a padding augmented ALU if we consider a long data sequence. We can consider n as the number of bytes the ALU processes in a cycle and N_n^{P8}, N_n^{P16}- the average number of cycles spent to process one chrominance and one luminance line *in a long data sequence*. Therefore, we can conclude that *in a long data sequence, a complete luminance line processing by a 64-bit padding augmented and pipelined ALU would take on average 2.5 cycles to perform ($N_n^{P16} = 2.5$). The padding of a chrominance line by the same ALU would take approximately one cycle ($N_n^{P8} = 1$).* The processing flow is similar for a 32-bit ALU, but the required average cycle number for luminance line processing is 5.5 (at least 4, at most 7 cycles), and the analysis is made on the right-most shape bits of each byte to be processed. Given the shape information of the whole line is available a-priori we can still make a branch determination, i.e. we can predict the operands issue sequence perfectly. The data in Table 3.2 can be used again in Equation (3.5) to evaluate the processing speed of ALUs with different operand widths.

Note on Figure 3.7(b) that the left and right-most inputs of the whole luminance line are initialized with ”0”, including the propagation of the shape bit. This is also valid for the chrominance line processing and for all up or down scaling ALU implementations (say 32 or 128 bit ALUs). In addition, if we assume the worst-case scenario for all cases, no shape bit analysis would be required. An implementation according such an assumption would lead to simpler data sequence control and would still yield very high processing speeds.

3.1.4 Simulation results and evaluation

In this section we evaluate the proposed implementations using the following measurements: *processing speed in MB/s* (macroblocks per second) *and chip area* (in logical blocks). For the evaluations we have taken into account: *the ability to exploit parallelism, pipelining, system inherent delay, and the variation of the processing cost in time* [52]. A comparison between our designs and padding units, previously reported in literature is also presented.

A. Systolic array implementation

To evaluate the proposed array padding structure, we have explored configurations with different numbers of PEs. We assume reconfigurable technology for two reasons. First of all, reconfigurable technologies run slower than gate arrays or other direct hardware implementation technologies. The implication of the previous statement is: showing the viability of our approach in reconfigurable technologies also proves its viability to all other current and near future technologies with other speed/area characteristics. Second, we envision (for cost efficiency) that assuming application specific requirements, such a unit could be incorporated in a reconfigurable augmented processor, e.g., [13, 16, 39, 48–50], which in current research appears to gain acceptance. The evaluation has been made in terms of *chip area* and *speed*. We have written synthesizeable VHDL models of a single PE and a generic multi-element structure of PEs. To get realistic values for the parameters of the unit, we have synthesized the VHDL models for two popular FPGA families - Altera and Xilinx, using the standard synthesis and simulation tools, provided by the vendors. We have not chosen the cutting-edge-of-technology chips for the implementation, because we have been interested in achieving high performance with lower technological generations of FPGAs (proving the proposed scheme for worse case conditions). We evaluated both the Xilinx xc4085xlpg559-09 [53] and the Altera epf10k20rc240-4 [54] chips, because they can be run at compara-

ble frequencies (around 100 MHz). Their chip organization, however, is quite different and area is estimated for each of the families. For both chip families we have evaluated structures of 4, 8 and 16 PEs and speed has been reported in MHz. Two extra evaluations for 32 and 64 PEs Xilinx mappings, have been done to illustrate how the data organization and the number of processing elements influence the performance of the unit.

Area and speed evaluation: Table 3.4 reports the area estimates for the Xilinx chip in the absolute units the vendor defines - CLBs (Configurable Logic Blocks) and in percentage of the available gate array area. For the Altera chip, results are reported in Table 3.5 in similar manner but the units for the absolute area are referred to as Altera defines them, namely - LCs (Logical Cells).

Table 3.4: Area-performance results for the Xilinx xc4085xlpg559-09 chip.

#	**# CLB's**		**Speed**		
PE	total	%	**MHz**	**MB/s**	
				Average	Worst C.
4	45 of 3136	4	24.5	95 700	76 600
8	206 of 3136	7	18.2	162 500	142 200
16	419 of 3136	14	11.4	237 500	237 500
32	838 of 3136	27	**11.4**	**475 000**	**475 000**
64	1676 of 3136	53	**11.4**	**950 000**	**950 000**

Table 3.5: Area-performance results for the Altera epf10k20rc240-4 chip.

#	**# LC's**		**Speed**		
PE	total	%	**MHz**	**MB/s**	
				Average	Worst C.
4	254 of 1152	22	24.8	96 900	77 500
8	511 of 1152	44	19.8	176 800	154 700
16	1024 of 1152	88	13.4	279 200	279 200

The speed estimations for both FPGA families suggest similar results. Besides the operating frequency, measured in MHz, we also evaluated the actual data processing speed of the different configurations. Since VOPs may vary in size and resolution, the MPEG-4 requirements group has defined the binding criteria for implementation complexity in terms of *transferred macroblocks per second* (Table 1.1). For consistency with this definition, in the last two columns of Tables 3.4 and 3.5, we have estimated the processing speed in macroblocks

per second ([MB/s]) according to Equation (3.5), more specifically - the average and worst case values. The reported numbers indicate that the padding structure can meet the real-time requirements for a broad range of visual resolutions. If we consider the implementation with 16 PEs, the estimated operating frequencies (11.4 MHz and 13.4 MHz) mean that the padding unit can process up to 237 500 MB/s (macroblocks per second) or 279 167 MB/s, depending on the FPGA family. Note that for structures with PE number, which is a multiple of 16, the average and worst case speeds are equal. This is due to the data dependency within a line and the 16-pixel wide data structure to be processed (already discussed along with the introduction of Table 3.2). In such large structures, the required number of cycles to pad a 16-pixel line is fixed (N_n^{P16}=const).

The *Core* and *Main* profile levels of MPEG-4 require processing speeds in the range between 2970 and 244800 Boundary MB/s to maintain from 4 up to 32 VOPs. It is obvious that the operating speed ranges, achievable by the proposed padding unit, completely match the required values. Even the most demanding profile level, *level 4* of the *Main* MPEG-4 profile, requires 244 800 Boundary MB/s for a high resolution session type (1920 x 1088) and 32 objects. This rate is in the order of the reported speed results for the feasible padding unit implementations. Furthermore, the unit can also meet the requirements for the maximum operating speeds from Table 1.1 (5940 MB/s - 489600 MB/s). This allows the boundary macroblock detection to be avoided as a processing stage, preceding the repetitive padding and all blocks in the visual scene to be padded. The potentials of the structure indicate capabilities to meet even more-demanding future profiles of the visual data compression standards.

Data-area-speed dependency: Results in Table 3.4 and Table 3.5 also indicate that the actual operating speed depends both on the number of processing elements and the structure of the processed data. For configurations with less than 16 PE, the area-speed relation is not proportional. The situation is different for structures with more than 16 PEs. The last two rows of Table 3.4 illustrate the influence of the data organization (16x16 pixel macroblocks) and the configuration of the structure on the processing speed (for numbers of elements lower and higher than 16). Since any two different lines (columns) within a macroblock are padded independently from each other, by parallelizing the processing per lines (columns), we significantly increase the processing speed without changing the operating frequency. On the other hand, structures with less than 16 PE, require extra circuitry to maintain the intermediate computational results. Therefore the number of the processing cycles is much higher and depends on the data. Figure 3.8 depicts the discussed influence of

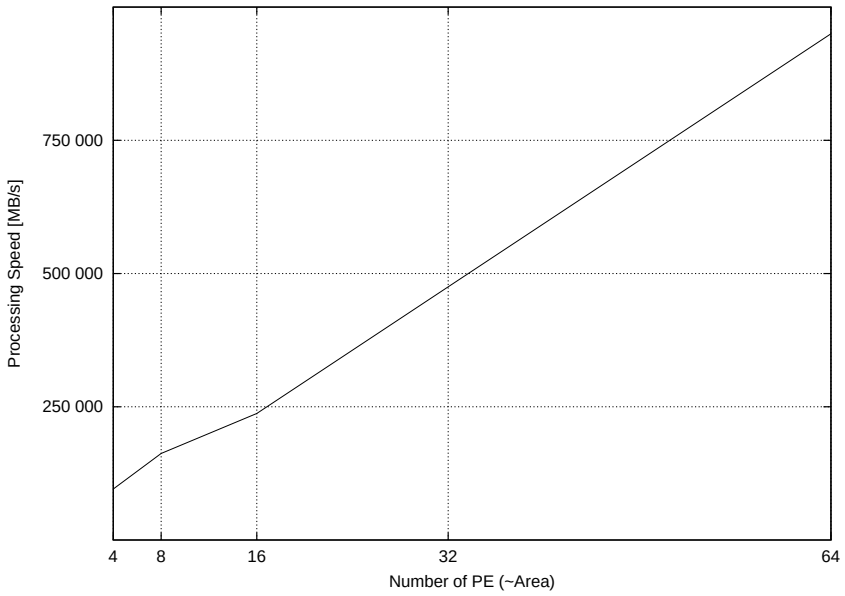


Figure 3.8: Data structure influence on the performance (mappings on Xilinx FPGA considered).

the data organization on the area-speed relation. It is evident that for structures with more than 16 PE the speed increases with the same rate as the area does. These properties of the implementation should be taken into account when either the area or the speed constraints of the unit are crucial.

B. ALU augmentation

As indicated in the previous section, our assumption is: *accommodate padding without creating critical ALU paths and preserve the ALU functionality*. Regarding the augmented general purpose ALU implementation, the key issue is not which frequency will the modified ALU perform but rather if such augmentation will create a critical path on the general purpose ALU (with the use of the byte controlled adder and the multiplexing), thereby potentially decreasing the processor operational frequency. Furthermore, it is of interest to establish if the expenses in terms of hardware are significant.

Critical paths / speed estimation. Before addressing the critical path of the stages, we discuss generally what is considered to be an ALU critical path. The ALU critical path in a general purpose design can be approximated to be

twice the delay of an adder when it is assumed that the execution of an ALU operation is performed in a single cycle. Given that a 64-bit adder using 2x2 AND-OR (or equivalent) gates requires 7 logic stages [55], the ALU can be approximated by 14 2x2 AND-OR logic stages. Regarding the critical path penalty issue, it has been noted that byte/nibble controlled adders (used in the past to perform, for example, decimal operations) will not increase the cycle time. The reason for such a possibility is that the masking is embedded implicitly in the stages. For a precise description and discussion for a controlled adder (more complex than the controlled adder proposed here), the interested reader is referred to [56].

The computation of padding requires two pipeline stages [2]: one computing the PN operation and one performing the masked ALU operation (see Figure 3.7(a)).

Pipeline stage 1: The following operations are performed in this stage:

- Operands are routed through the propagation chain.
- Data, to be processed in Stage 2, is loaded in the pipeline latches.
- Control signals for the output multiplexer are generated.

The first stage critical path is clearly linear to the length of input data and it is a serial operation. This critical path is equal to the number of bytes in the ALU operands plus one multiplexers. Given that the worst case is the largest input ALU , implemented in practice (64 bit), the critical path is equal to the delay of $1 + \frac{64}{8} = 9$ multiplexers which fits into a single ALU cycle. For usual 32-bit units the delay is equal to 5 multiplexer delays. It should be noted, that operands are passed through the Propagation Nodes only when a padding operation is performed. They are bypassing the first pipeline stage for a conventional ALU operation, adding one extra input in the already existing bypassing multiplexer. Consequently, the first cycle of padding computations will not imply a critical path problem. The bottomline is that for evaluating the performance of the scheme, proposed here, it is safe to assume no augmentation to the processor cycle times.

Pipeline stage 2: In this pipeline stage the following operations that could comprimise the critical path are performed:

- The Byte Controlled Adder performs masked additions over the data stored in the pipeline latches.

[2] NOTE: A pipeline stage is performed in a machine cycle, a logic stage is performed with the delay of a gate.

- The output multiplexer issues the appropriate results according to the generated control signals (see Table 3.3 and Figure 3.5).

The ALU critical path penalty could have been augmented by a single 2-way AND element (added possibly to the critical path of the byte controlled adder). Such a penalty has been shown to be avoidable [56] with implicit computations. Thus it should not extend the critical paths of a general purpose ALU implementation. The critical path penalty for reading from the general purpose register or bypassing the operands of the ALU is a 2-1 multiplexer and it should be noted that such a multiplexer already exists. It is used, for example, to perform bypassing of operands from other units, direct data passing from caches etc. The only foreseeable penalty is adding a single input to the already existing multiplexer, which is not anticipated to create critical path problems.

In estimating the expected performance we note that *an ALU instruction takes 1 cycle, padding takes 2 cycles*. To estimate the possible speed achievable from the proposed solution we consider the following: Let us assume an $n \cdot 8$-bit padding augmented ALU operating at frequency F_n [Hz]. Let us assume values of *n* that have practical significance - 4, 8, 16 (i.e., 32, 64 or even 128-bit ALU). To evaluate the speed of the ALU we can use Equation (3.5) which gives results for long data sequences. Assuming a value of $F_n = 1GHz$ (which is currently easily achievable for general purpose processors) and using the data from Table 3.2 into Equation (3.5), we calculated the implementation speeds, given in Table 3.6.

Table 3.6: Processing speed at clock frequency F_n=1 GHz.

ALU bits	**n**	**Speed [MB/s]**	
		Average	Worst Case
32-bit	4	3 906 250	3 125 000
64-bit	8	8 928 600	7 812 500
128-bit	16	20 833 300	20 833 300

The most demanding profile level, *level 4* of the *Main* MPEG-4 profile, requires 244 800 Boundary MB/s for a high resolution session type (1920 x 1088) and 32 objects (Table 1.1). This rates are an order of magnitude lower than what the augmented ALU implementations achieve (see Figure 3.9). The potentials of the structure indicate capabilities to meet even more-demanding future profiles of the visual data compression standards [57].

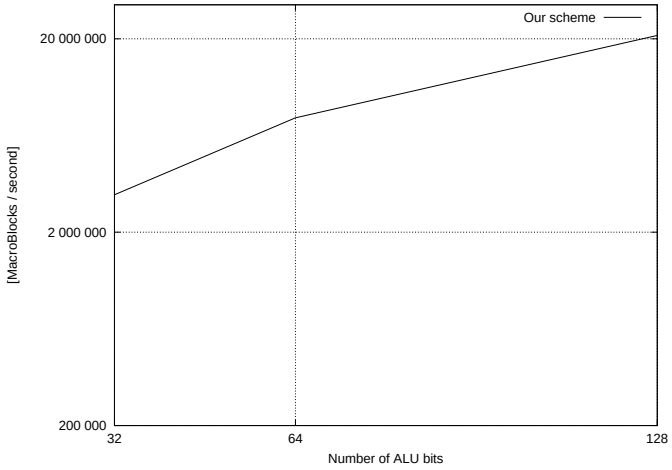


Figure 3.9: Processing speed for different ALU operand sizes and F_n=1 GHz. Note the logarithmic scale.

Table 3.7: Hardware gates estimations.

ALU bits	**n**	**Number of extra gates**
32-bit	4	172
64-bit	8	344
128-bit	16	688

Hardware estimations: We choose the 2x2 AND-OR logic block as a basis for the hardware estimations. The reason for such a choice lies on the fact that such a block is commonly available to most technology libraries [55]. A one-bit 2 to 1 multiplexor is a 2x2 AND-OR gate. The hardware penalty for a single byte padding structure is: 2 x 9-bit multiplexers, 2 x 8-bit multiplexers and 1 OR gate. That makes 2x9+2x8+1=35 2x2 AND-OR gates. An n-byte implementation will cost $n \cdot 35$ AND-OR gates plus additional cost for the ALU multiplexer of $n \cdot 8$ gates, i.e. $n \cdot 43$ 2x2 AND-OR gates. Table 3.7 contains the exact values of the hardware penalties for different ALU sizes. It is noted that our estimations, as indicated in Table 3.7 strongly suggest that the hardware cost is rather minimal when compared to current chip sizes. It is important to note that for padding in the worst case scenario no data analysis is required therefore it implies no additional hardware. When data analysis is

implemented, the hardware overhead can be estimated as additional inputs to the operand addressing circuit (in the presented 64-bit example it is a single input to the operand addressing circuit).

Related work: The repetitive padding algorithm is described in [1, 5], but some modifications have also been reported. In [58–60] new algorithms or algorithm modifications are proposed to redefine or even substitute the original repetitive padding. All of them suggest software improvements of the coding efficiency and visual quality and do not focus on the hardware execution, nor on performance. They are software algorithms and they are not hardware implementations, consequently they are not pertinent to the work presented here. The idea for a dedicated VLSI hardware padding accelerator was reported in [61]. The unit, proposed there, has a complex organization and processing control - it is dedicated for 64-bit data, contains 3 sub-units, operating in 4 internal states with low flexibility and scalability. The achieved processing speed is 245 000 MB/s at clock frequency 100 MHz. Compared to it, our proposals have the following advantages:

- *Faster processing* - if we assume the same operating speed (100 MHz) for both ASIC and ALU schemes, our 64-bit implementations are 3 times faster(781 250[3] MB/s vs. 245 000 MB/s).
- *Flexibility and scalability* - we have shown that our approaches allow high levels of scalability and flexibility which is not the case in [61].
- *Simple hardware* - the control schemes of our implementations are simpler, thus more cost effective than the design discussed in [61]. Our hardware overhead is just a few multiplexers vs. the 3 subunit design with high communication complexity.

A hardware acceleration of the padding, which appears to be faster, is discussed in [62]. In such a proposal, the padding algorithm is modified to support specific instruction set extensions as the horizontal and vertical padding processes are divided into two phases each. These two phases consequently scan the lines/columns into two opposite directions and perform the padding operations. In the proposed solution there is a hardwired dedicated padding unit with high control overhead, supporting 8 new instructions. Its estimated processing speed at 100 MHz clock frequency is 250 000 MB/s for 32-bit data width. This proposal differentiates with schemes described here in the following:

[3] Data in Table 3.6 should be divided by 10 to get the numbers for 100 MHz clock.

- *Higher processing speeds.* In [62], the processing speed is reported only for a 32-bit unit. Although the design is claimed to be scalable and implementable for larger data types, for 64- and 128-bit units data are not reported. For 32-bit data our units are over 20% faster (312 500 MB/s vs. 250 000 MB/s). If we predict the processing speed for 64 and 128-bit units according to the scheme proposed in [62], the processing speed of the unit discussed there increases (at most) linearly with the operand width. Our approaches allow an exponential speed up when the data width increases, because of the better data processing scheme. For 64-bit data and the same clock frequency (100 MHz) our units can be estimated to be over 50% faster, while for 128-bit data, the estimated speed up is over a factor of 2.

- *Simpler control.* To perform the padding algorithm, our units require only one additional instruction, while in [62] 8 new instructions are introduced. As a rule in computer engineering, a higher number of additional instructions imposes more severe architectural modifications and more complicated data paths and control circuitry in the implementation. It is always preferable to limit the opcodes added into an architecture. Our proposal is clearly better as it requires the minimum (one) instruction addition to an instruction set.

Both [62] and [61] present hardware estimations for 0.35 μm CMOS technology and do not report any technology independent hardware estimations (e.g., number of logical gates). Consequently, we can not make an exact and independent comparison between the hardware size complexities of these units.

Finally, in the present discussion we use the standard repetitive padding algorithm (differentiates from [58–60]) as we scan each line and column of a macroblock bidirectionally in parallel (differentiates from [62] where the padding algorithm is modified), thus saving a number of processing cycles. Our two approaches for the hardware acceleration of the algorithm are scalable (differentiates from [61]) and differentiate from all of the above mentioned references with the reconfigurable implementation and the general-purpose ALU modification (for padding not reported in literature before).

3.2 The accepted quality function

Previous research, reported in [46, 47, 63, 64], indicates that after motion estimation, the next computationally most demanding algorithm in MPEG-4 is the shape encoding. An essential part of the shape encoding process is the necessity to ascertain whether the encoded BAB has an accepted visual quality under some specified lossy coding conditions. To achieve this, the so-called "ACcepted Quality" (ACQ) function is defined. The ACQ function is intensively used in three basic procedures of the MPEG-4 encoding process, namely *mode decision*, *binary motion estimation and compensation* and *rate control*:

- **Mode decision:** A mode decision procedure is performed over each BAB and there exist seven modes to code the shape information of each macroblock [1].
- **Binary motion estimation and compensation** for shape: This is the most demanding part of shape encoding in terms of performance. The motion estimation and compensation for shape is similar to the traditional motion estimation and compensation for full-color video frames, but in MPEG-4 it is performed over the binary alpha map.
- **Rate control** is obtained through block level size conversion of all BABs. The conversion ratio (CR) is 1/4, 1/2 of or the original size. Each 16x16 BAB is down-sampled to (16xCR)x(16xCR) and then up-sampled back to 16x16 by means of filters [1].

In this section, we introduce a scalable systolic hardware implementation of the new MPEG-4 "ACcepted Quality" function.

3.2.1 Definition of the ACQ function

Each BAB is divided into 16 4x4 pixel blocks (PB) and this data structure is used by the criterion for an accepted quality. A dedicated function called *accepted quality function* (ACQ) is defined in the MPEG-4 video verification model [1]:

Definition 3.1 *Given the current original binary alpha block i.e. BAB and some approximation of it i.e.* $BAB^{'}$*, it is possible to define a function*

$$ACQ(BAB^{'}) = MIN(acq_0, acq_1, ..., acq_{15}), \tag{3.6}$$

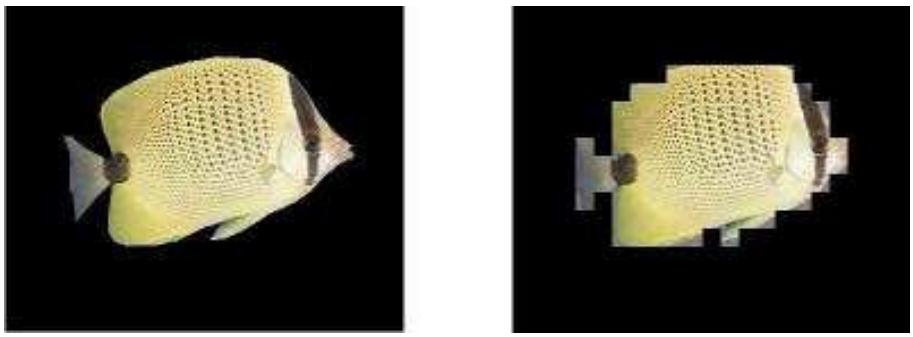


Figure 3.10: Alpha threshold influence on the VOP visual quality: left - alpha_th=0; right - alpha_th=256.

where

$$acq_i = \begin{cases} 0 & if\ SAD\_PB_i > 16 * alpha\_th \\ 1, & otherwise. \end{cases} \tag{3.7}$$

and $SAD\_PB_i(BAB, BAB^{'})$ is defined as the sum of absolute differences for PB_i, where an opaque pixel has value of 255 and a transparent pixel has value of 0.The parameter alpha_th *has values of* {*0,16,32,64,...,256*}.

The ACQ function determines whether the encoding ($BAB^{'}$) of a certain BAB gives an accepted quality result according some specified lossy coding conditions. These conditions are formally included in the alpha threshold parameter. Figure 3.10 depicts the influence of the *alpha_th* parameter on the appearance of an encoded VOP. The higher the *alpha_th* value is, the lower the acceptable quality of the encoding is. If *alpha_th=0*, then encoding will be lossless (with the highest visual quality).

3.2.2 Implementation

To implement the ACQ function we make some mathematical manipulations first. Let us represent SAD_PB_i as follows:

$$SAD\_PB_i = 255 \sum_{j=0}^{15} |P_{i.16+j} - P^{'}_{i.16+j}| \tag{3.8}$$

where $P_{i.16+j}$ and $P^{'}_{i.16+j}$ are the *binarized* values of the j^{-th} pixels from PB_i and $PB^{'}_i$ respectively and a value of *0 represents a transparent pixel* while a

value of *1 - an opaque one*. According to these assumptions, we can substitute the absolute difference in (3.8) with a *xor* operation:

$$SAD\_PB_i = 255 \sum_{j=0}^{15} (P_{i.16+j} \oplus P^{'}_{i.16+j}) =$$

$$= 255(PB_i \oplus PB^{'}_i) = 256(PB_i \oplus PB^{'}_i) - (PB_i \oplus PB^{'}_i) \tag{3.9}$$

where $PB_i \oplus PB^{'}_i$ denotes the bit sum of the bit-by-bit *xor* over the pixel blocks.

According to Definition 3.1 and Equation (3.9):

$$acq_i = (SAD\_PB_i \leq alpha\_th * 16) =$$

$$= [256(PB_i \oplus PB^{'}_i) \leq alpha\_th * 16 + (PB_i \oplus PB^{'}_i)] \tag{3.10}$$

and

$$ACQ(BAB^{'}) = AND_{16}(acq_0, acq_1, ..., acq_{15}) \tag{3.11}$$

According to Definition 3.1, $alpha\_th * 16 = alpha\_th_5 * 256$, where $alpha\_th_5$ denotes the five MSD of *alpha_th*. On the other hand the result of $(PB_i \oplus PB^{'}_i)$ is a five-digit number and we can reduce the acq_i computation to the comparison of two 5-digit numbers as follows: $acq_i = [(PB_i \oplus PB^{'}_i) \leq alpha\_th_5 . \frac{256}{255}]$ and since $\frac{256}{255} \approx 1$:

$$acq_i \approx [(PB_i \oplus PB^{'}_i) \leq alpha\_th_5] \tag{3.12}$$

The implementation of Equation (3.12) is depicted on Figure 3.11. We can assume the discussed structure as a basic processing element (PE) and, taking into account Equation (3.11), we can build the systolic processor shown on Figure 3.12.

3.2.3 Scalability and data bandwidth

The proposed circuit would take two cycles for execution in a real implementation - one cycle for the adder tree and one for the comparator. If pipelined, the structure can produce a valid result every cycle given the data throughput requirements are met. On the other hand, the structure is scalable and can meet any memory bandwidth restrictions. For its efficiency, however, a multiple of 16 bits per cycle bandwidth is recommended, ranging between 16 and 256

bits/cyc for a single BAB. Figures 3.11 and 3.12 show the two extreme cases - a pixel block processor and a BAB processor. These two processors differ in the granularity and the throughput of the processed data. If we use an on-chip memory buffer with a suitable organization for the ACQ engine, we will be able to achieve higher data throughput.

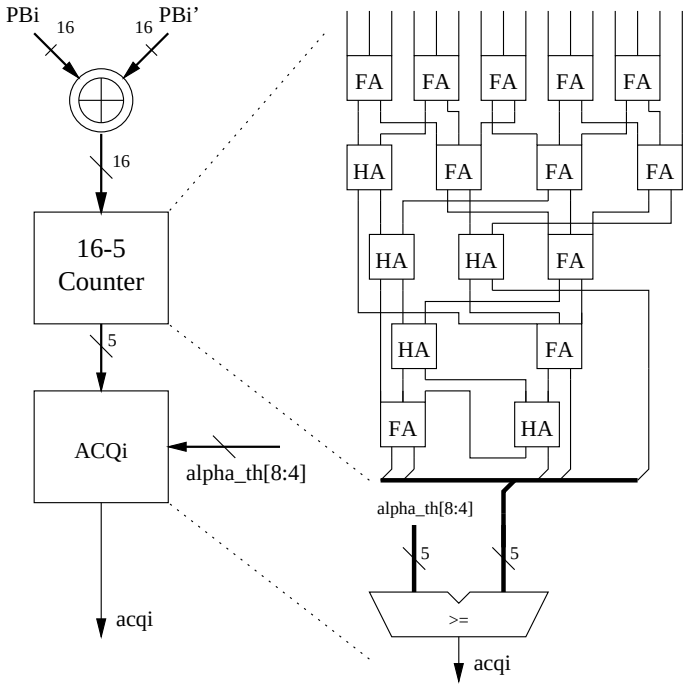


Figure 3.11: Accepted quality single pixel-block processing element.

3.2.4 Evaluation

To evaluate the proposed structure of the ACQ function accelerator, a single processing element and an array of processing elements have been modeled in VHDL and RTL simulations have been run. The VHDL models have been synthesized for Altera FPGA. The reference software for the evaluation of the structure is Altera Max+Plus II. The simulation results indicate that each processing element performs the acq_i function within 60 ns. The evaluation of the MIN function takes about 2 ns. Table 3.8 suggests the processing latency and memory bandwidth, required for different number of processing elements in an Altera FPGA. Besides the operating latency, we use another measurement

for the speed of the engine in terms of processed data units per time unit. In the

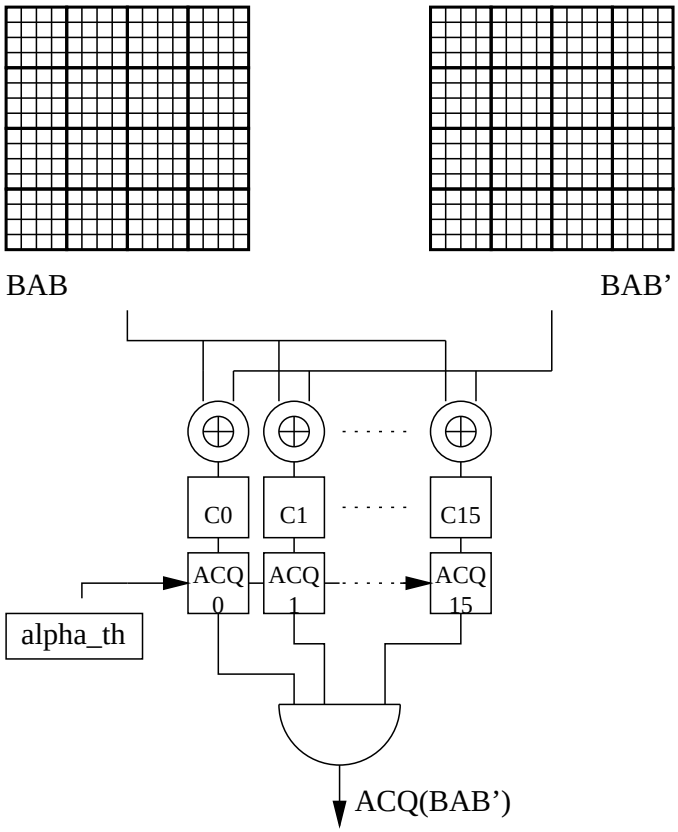


Figure 3.12: The ACcepted Quality processing structure.

proposed engine, the basic data units are BABs and we achieve a speed of up to 16 129 032 BAB/s. Since there is a macroblock corresponding to any BAB and the macroblock processing speed is defined in the MPEG-4 profiles [6], we can use our results to estimate the real-time operating capabilities of the circuit. For the core and main MPEG-4 profiles, the required real-time rates to process 16 and 32 video objects are 23 760 MB/s and 97 200 MB/s (macroblocks per second) respectively (see Table 1.1). These numbers are well below our simulation results and, assuming that a macroblock manipulation involves a BAB processing as well, it is evident that the proposed ACQ engine can easily meet the real-time constrains of a dedicated MPEG-4 shape processor.

Table 3.8: ACQ Processing speed and required data bandwidth according to the number of processing elements (for Altera FPGA).

Number of PE	Processing time in ns	BAB/s	Data bandwidth
1	992	1 008 065	16 bit
2	496	2 016 129	32 bit
8	124	8 064 516	128 bit
16	62	16 129 032	256 bit

3.3 Lifting based discrete wavelet transform

The Discrete Wavelet Transform (DWT) has become a basic encoding technique for recent data compression algorithms. Compared to traditional DCT-based processing, DWT yields higher compression ratios and better visual quality. We consider the so called *lifting scheme* [65], an implementation of the DWT recognized to be fast and extremely cost efficient. In the lifting scheme, half of the data samples are used to predict the other half. The transform process is split into three phases, which are iterated until all samples are processed to a certain level of granularity. Applying the inverse transform in the lifting scheme, as long as the transform coefficients are not quantized, will always result in a perfect reconstruction of the original picture, regardless of the precision of the applied arithmetic. Moreover, it is possible to use integer arithmetic without encountering problems due to finite precision or rounding.

In this section, we introduce an original hardware design of the lifting based Wavelet Transform. To maximize the performance of the design, we utilize different techniques such as pipelining, parallel operating modules and data reusability. Synthesis results for a a basis structure implemented in a Xilinx Virtex II FPGA indicate trivial reconfigurable area utilization of 985 CLB slices, 669 Flip-Flops, 22 Block RAMs and 8 multiplier blocks. The design can achieve high performance for filters factorized in lifting steps.

The remainder of this section is organized as follows. In Subsection 3.3.1, we present some limited background on the DWT and its traditional implementations. The lifting scheme is briefly described in Subsection 3.3.2. We introduce our design in Subsection 3.3.3 and evaluate it in Subsection 3.3.4.

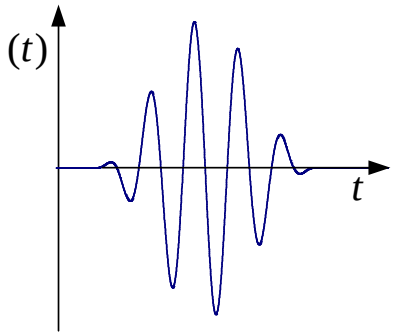


Figure 3.13: Wavelet prototype function - an example.

3.3.1 DWT background

Time-limited signals (space-limited in case of pictures) can be represented efficiently using a basis of block functions (Dirac delta functions), but these block signals do not efficiently handle frequency limitations. Band-limited signals can be represented efficiently using a Fourier basis but Sines and Cosines are not limited in time. Wavelet functions, as a compromise between the pure time-limited and band-limited basis functions, combine the best of both. Wavelets, literally meaning small waves, are mathematical concepts for decomposing a function, say *f*, into sets of other functions known as wavelet bases- $\Psi_{a,b}(t)$:

$$f = \sum_{t} C_{a,b} \cdot \Psi_{a,b}(t) \tag{3.13}$$

For an efficient representation of the signal *f*, using only a few coefficients $C_{a,b}$, it is very important to use a suitable family of functions $\Psi_{a,b}$. The functions $\Psi_{a,b}$ should match the features of the data we want to represent.

In order to get the variable time-frequency localization *(resolution)*, a wavelet called *mother wavelet* or *prototype function* $\Psi(t)$ is defined. One possible mother wavelet is depicted in Figure 3.13. Basis functions $\Psi_{a,b}(t)$ are calculated as scaled and translated versions of the prototype:

$$\Psi_{a,b}(t) = \frac{1}{\sqrt{b}} \cdot \Psi(\frac{t-a}{b}) \ , \tag{3.14}$$

where *b* is the scaling coefficient and *a* is the translating coefficient. In addition, the Discrete Wavelet Transform (DWT) is introduced:

Definition 3.2 *The **DWT** with resolution level j (i.e., scale $b = 2^j$) at time k is defined as:*

$$\Psi_{j,k}(t) = 2^{-\frac{j}{2}} \cdot \Psi(2^{-j} \cdot t - k) \tag{3.15}$$

Considering Equations (3.13) and (3.15), any signal can be represented as the sum of a set of wavelet coefficients at an infinite number of scales:

$$f(t) = \sum_{j,k} C_{j,k} \cdot \Psi_{j,k}(t) \tag{3.16}$$

$$C_{j,k} = \int_{-\infty}^{+\infty} f(t) \cdot \Psi_{j,k}(t) dt \tag{3.17}$$

This equation resembles the Fourier Transform and is called the *classic wavelet transform* [66].

Traditional DWT implementations: DWT can be implemented using different prospects of the transform. For example, the wavelet transform coefficients can be generated using 2 channel *filter banks* called synthesis filters. The input signal is split into two signals using a low-pass filter *h(t)* and its orthogonal or bi-orthogonal high-pass filter *g(t)*. Multiple levels or "scales" of the wavelet transform are made by repeating the filtering and decimation process on the lowpass branch outputs only. The process is typically carried out for a finite number of scales, resulting the wavelet coefficients.

Another implementation-oriented prospect of the DWT is the *Fast Wavelet Transform*. In this case, the DWT can be factorized into a product of a few sparse matrices using similarity properties. When these factors are applied to the multiplication with a vector, the order of operations reduces, thus the transform is called "fast".

3.3.2 The lifting scheme

The Lifting scheme is an efficient implementation of a wavelet transform algorithm. The basic idea behind the lifting scheme is *to use the correlation in the data to remove the redundancy*. It was primarily developed as a method to improve wavelet transform, and then it was extended to a generic method to create so-called second-generation wavelets (i.e. wavelets which do not necessarily use the same function prototype at different levels). Second-generation wavelets are much more flexible and powerful than first generation wavelets [65]. In [67], it is shown that any discrete wavelet transform can be obtained with a finite number of lifting steps. The lifting scheme is an

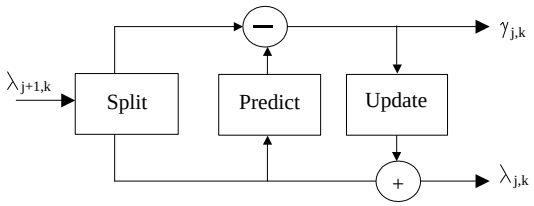


Figure 3.14: The lifting scheme.

implementation of the filtering operations at each level. The algorithm can be described in three phases, namely: *Split phase*, *Predict phase* and *Update phase*, as illustrated in Figure 3.14.

Split Phase: Assume the scheme starts with an input data set of $\lambda_{0,k}$ samples, where *k* represents the data element and zero signifies the original data level. In the first stage, the data set $\lambda_{0,k}$ is split into two subsets of the separated even and odd samples:

$$\lambda_{-1,k} = \lambda_{0,2k} \; ; \;\; \gamma_{-1,k} = \lambda_{0,2k+1} \; . \tag{3.18}$$

The negative indices have been used according to the convention: the smaller the data set, the smaller the index. The subsampling into even and odd samples is also referred to as the *lazy wavelet transform* because it does not de-correlate the data.

Predict Phase or Dual Lifting: In this step the even subset $\lambda_{-1,k}$ is used to predict the odd subset $\gamma_{-1,k}$ by a prediction function $P(\lambda_{-1,k})$. The prediction function should be such that the more correlation in the original data, the closer the predicted value to the original $\lambda_{-1,k}$. Further, each sample of the subset $\gamma_{-1,k}$ is replaced by the difference between itself and its predicted value, namely:

$$\gamma_{-1,k} := \lambda_{0,2k+1} - P(\lambda_{-1,k}) \tag{3.19}$$

Broadly speaking, two types of prediction functions are considered - *piecewise linear* and non-linear or *interpolating*:

Piecewise linear - the odd samples are predicted as the average of its two even neighbors, $\lambda_{-1,k}$ and $\lambda_{0,k+1}$, which is given by:

$$\gamma_{-1,k} := \lambda_{0,2k+1} - \frac{1}{2}(\lambda_{-1,k} + \lambda_{0,k+1}) \tag{3.20}$$

Interpolating subdivision - this model uses the same basic idea as the piecewise linear but uses 2 or more neighbors to either side and an interpolating

function to predict the odd samples. The order of the interpolating subdivision, denote it by *N*, is important because it sets the smoothness of the interpolating function used to find the wavelet coefficients. This function is referred to as the dual wavelet and *N* is referred to as the number of *dual vanishing moments*. Thus *the number of dual vanishing moments defines the degree of the polynomials that can be predicted by the dual wavelet*. Depending on *N*, the wavelet coefficients can measure the failure to predict. For instance, *N =2* indicates the failure to be linear and *N=4* measures the failure to be cubic.

Update Phase or Primal Lifting: The last λ coefficients of the predict phase are samples from the original data and present the coarsest level, which introduces considerable aliasing. We would like some global properties of the original data set to be maintained in the smaller sets $\lambda_{j,k}$. For example, in the case of images, we would like smaller images to have the same overall brightness, i.e. the same average pixel value. This problem is solved by introducing the third stage of the lifting scheme, the *update stage*. In this stage the coefficient $\lambda_{-1,k}$ is lifted with the help of the neighboring wavelet coefficients so that a certain scalar quantity *Q* (e.g., the mean), is preserved:

$$Q(\lambda_{-1,k}) = Q(\lambda_{0,k}) \tag{3.21}$$

Therefore, a new (*update*) operator *U* is applied:

$$\lambda_{-1,k} := \lambda_{-1,k} + U(\gamma_{-1,k}) \tag{3.22}$$

In this phase, also referred to as *primal lifting*, a *scaling function* is calculated from the previously calculated wavelet coefficients to maintain some properties among all the λ coefficients throughout every level. The order of this function is some even value $\tilde{N}$ referred to as *real vanishing moment*, not necessary equal to *N*.

Inverse Transform: The inverse transform of the lifting scheme just follows the reverse data flow in the setup of the forward transform with small changes like switching between additions and subtractions. The following summarizes the forward and the inverse transform lifting steps:

Forward transform:		Inverse transform:	
Split:	$\lambda_{j,k} = \lambda_{j+1,2k}$	Update:	$\lambda_{j,k} = \lambda_{j,k} - U(\gamma_{j,k})$
	$\gamma_{j,k} = \lambda_{j+1,2k+1}$		
Predict:	$\gamma_{j,k} = \gamma_{j,k} - P(\lambda_{j,k})$	Predict:	$\gamma_{j,k} = \gamma_{j,k} + P(\lambda_{j,k})$
Update:	$\lambda_{j,k} = \lambda_{j,k} + U(\gamma_{j,k})$	Merge:	$\lambda_{j+1,2k} = \lambda_{j,k} \bigcup$
			$\lambda_{j+1,2k+1} = \gamma_{j,k}$

A comprehensive method to calculate dual and primal lifting coefficients is described in [68].

Integer transform: The lifting scheme can be easily adjusted to integer arithmetic by scaling and rounding the predict and update filter results. Moreover, as indicated in [69, 70], a perfect reconstruction of the original image is obtained regardless the rounding errors. This is a very important property of the lifting scheme, which means that *we can build an integer version of every wavelet transform and implement it efficiently without losing precision.*

Advantages of the lifting scheme, compared to the classical filter bank algorithm:

1. Lifting leads to a speedup, therefore it is also referred to as *fast lifting wavelet transform* (FLWT). Asymptotically, for long filters, the lifting scheme has half the computational complexity of the classical filter bank wavelet implementation [67].

2. All operations within a lifting step can be done entirely in parallel while the only sequential part is the order of lifting operations.

3. Lifting allows adaptive wavelet transforms. The analysis of a function can start from the coarsest level, followed by data processing at finer levels in the areas of interest.

4. Lifting can be done in-place, which means auxiliary memory is not needed. The new stream at every summation point replaces the old one.

5. It is easy to build non-linear wavelet transform by lifting. For example, the lifting scheme allows integer-to-integer transform, while keeping a perfect reconstruction of the original data set.

3.3.3 The proposed design

The implementation of the lifting algorithm will be explained assuming a signal of length 12 and a polynomial filter with numbers of dual and real vanishing moments, both equal to 4 (L=12, N=4, $\tilde{N}$=4). The proposed design, however, is scalable and can be implemented for arbitrary signal lengths and different numbers of filter coefficients. The filter coefficients can be calculated according to the scheme explained in [68].

A. The predict module

All predict phase calculations for the considered example ($N = 4$, $L = 12$) are illustrated in Figure 3.15. The magnified circle depicts the arithmetic op-

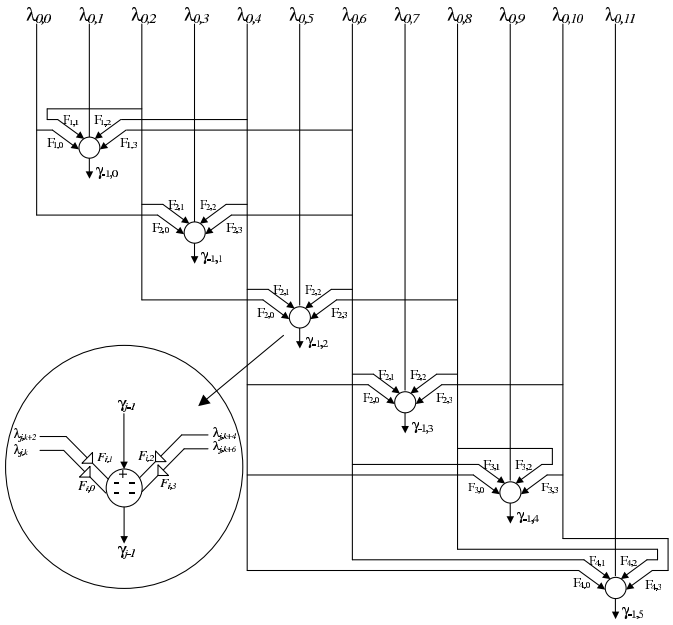


Figure 3.15: Calculations in the predict phase for $N = 4$, $L = 12$.

erations involved in the calculation of each γ:

$$\gamma_{j-1} -= (\lambda_{j,k} \cdot F_{i,0}) + (\lambda_{j,k+2} \cdot F_{i,1}) + (\lambda_{j,k+4} \cdot F_{i,2}) + (\lambda_{j,k+6} \cdot F_{i,3}) \quad (3.23)$$

That is: 4 multiplications, 4 additions/subtractions and 16 memory accesses (4 times read l, read lifting coefficients, read γ, and write γ). These operations consume quite many processor cycles when implemented sequentially. In our design, we introduce parallel processing and data reusability as depicted in Figure 3.16. To meet the requirement for high memory throughput, we break the design problem into four sub-problems, namely:

1. Accessing N λs concurrently;
2. Accessing N filter coefficients concurrently;

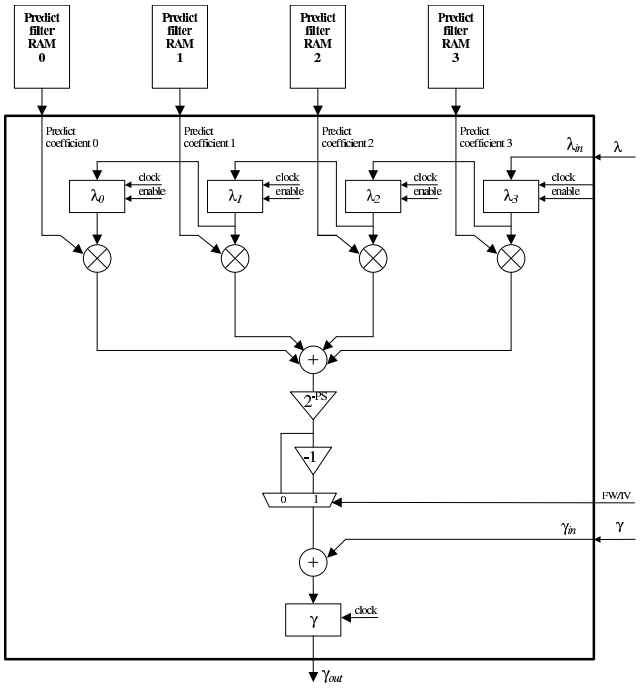


Figure 3.16: The predict module.

3. Reading input γ concurrently;

4. Writing back predicted γ concurrently.

Following, the devised solutions to each of these subproblems are presented.

Accessing $N\lambda$s concurrently: Observing Figure 3.15, it can be noticed that in the calculations of two consecutive γs, at least three out of four λs are common. This implies that reading only one λ from the memory will be sufficient to calculate the following γ, provided that λs from the preceding calculations are temporally stored in buffers for reuse. Therefore, we implemented a pipeline of N stages for λ inputs. During initialization, the pipeline is filled in N cycles. After that, N λs are issued to the multipliers in parallel (see Figure 3.16).

Accessing N filter coefficients concurrently: This is accomplished by using separate banks of RAM for the filter coefficients, as depicted in Figure 3.16.

Reading input γ concurrently: For the parallel processing of the unit, γ inputs have to be read simultaneously with λ inputs. This means that two different

locations of the image storage area have to be accessed at the same moment. To solve this design problem, we utilize true dual port memories with separate, independably addressable input/output ports. Using such memories for picture data, we access γs and λs concurrently (see Figure 3.16). The embedded RAM blocks of the Xilinx Virtex II FPGA chip can be configured as true dual port memories.

Writing back predicted γ concurrently: Reading γ and λ inputs simultaneously occupies both ports of the dual port picture RAM. To write the predicted γ back into memory within the same system cycle, we designed the picture RAM to operate at a frequency two times higher than the rest of the design. In this case four memory locations per system cycle are addressed.

Boundary treatment: For correct processing of the signal around its boundaries, filter coefficients are modified near the beginning and the end of the input sequence. Note in Figure 3.15, that the first two and last three (i.e., the boundary) calculations are performed over identical samples. Therefore, around the signal boundaries the pipeline with the λ coefficients has to be stalled and only the corresponding filter coefficients and γ should be read. This is achieved by the $enable$ signals of the λ pipeline in Figure 3.16.

Scaling: In order to obtain a satisfactory precision of the integer arithmetic and to alleviate the effect of the introduced huge non-linearities, filter coefficients are scaled up first (not explicitly depicted in Figure 3.16) and the result is later scaled down with a factor of 2^{-PS} (the triangle in Figure 3.16).

Forward and inverse predict: The predict phases of the forward and inverse transform differ just in one operation, namely addition is substituted by subtraction (see Subsection 3.3.2). Therefore, the same predict module can be used for both forward and inverse transform by selectively alternating the *FW/IV* signal in Figure 3.16. This signal controls the multiplexing of the positive or negative (two's complement) result of the multiple-input adder to the final adder, thus essentially performing either addition or subtraction.

B. The update module

Figure 3.17 illustrates the calculations during the update phase for the considered example. In each row, one γ updates those λs, which it has affected in the preceding predict phase. This can be described by the following pseudocode:

$$for\ n = 0\ to\ \tilde{N}\ \{\lambda_{j-1,n} += \gamma_i \cdot L_{i,n}\}$$

In the assumed scenario ($N = 4$, $\tilde{N} = 4$, and $L = 12$), each γ updates 4 λs. This results in 4 multiplications, 4 additions/subtractions and 16 memory accesses. Regarding memory throughput, design problems similar to those in the predict phase have been considered:

1. Accessing λs concurrently
2. Accessing filter coefficients concurrently;
3. Reading input γ concurrently;
4. Writing back updated λs concurrently.

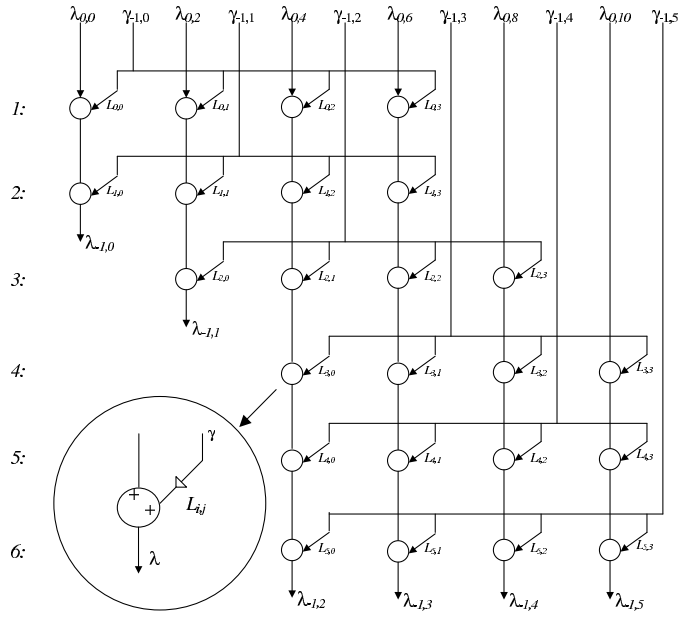


Figure 3.17: Calculations in the update phase for $\tilde{N} = 4$ and $L = 12$.

Figure 3.18 depicts the organization of the update module. The pipeline is filled from different sources to accommodate all computations illustrated in Figure 3.17. The following explains how control signals configure the update module to operate in the four different configurations, namely *initialization*, *boundary processing*, *pipelined processing*, and *empty pipeline*.

Initialization: During initialization, the pipeline registers are filled with the initial data. The control signal $next\lambda$ is set to 1 and the γ input is reset to 0. In

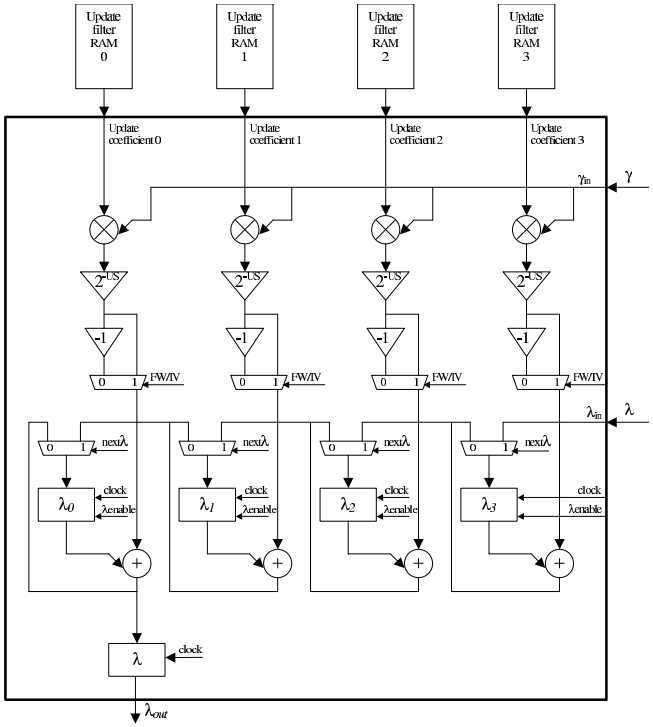


Figure 3.18: The update module.

the first 4 cycles, the contents of the λ registers are shifted to their neighbor at the left side, while $\lambda 3$ keeps reading the data from the picture RAM. Simultaneously with filling the last λ, the corresponding filter coefficients and the first γ value (first left-effected boundary γ) are being loaded via the update coefficient and γ_{in} inputs of the module, respectively. The updated λs are calculated at the output of the adders and are ready to be stored.

Boundary processing: This configuration is used for processing of boundary affected γ samples (except the last right-affected one) and the last non-affected γ. Signal $next\lambda$ is reset to 0, which stalls the pipeline and the loaded (boundary-affecting) λs are not shifted, i.e., they are stored in the same register. Different, boundary-specific filter coefficients are issued, the output sample is calculated and issued via λ_{out}.

Pipelined processing: All boundary non-affected γs (except the last one) are processed with this configuration. The control signal $next\lambda$ is set to 1, which enables the pipeline and all updated λ outputs are shifted. The rightmost λ

register ($\lambda 3$) is loaded with a new λ value, and the updated λ value is available at λ_{out}.

Empty pipeline: This configuration is utilized for the last boundary right-affected γ and is almost identical to the initialization. The only difference is that all λ values have to be written back into the memory, when they are available at λ_{out}.

Scaling: To increase the calculation accuracy, the stored filter data is scaled up. Just like the predict module (Figure 3.16), the update module includes logic for scaling down the result after each multiplication (illustrated by triangular symbols in Figure 3.18).

Forward and inverse update: The update phases of the forward and inverse transform differ in the addition-subtraction operations, just like the predict phases (see Subsection 3.3.2). The control signal FW/IV switches between summation and subtraction, to make the module perform both forward and inverse transform.

C. Parallel operation of the predict and update modules

It has been explained how the predict and update modules calculate one output in every cycle. However, parallel operation of both modules leads to 6 memory operations per cycle (for predict module - the λ and γ inputs and the γ output, and for update module - the λ and γ inputs and the λ output). On the other hand, the dual port memory is accessed twice per system cycle, accommodating a maximum of 4 memory accesses per cycle. Therefore, a straight-forward parallel implementation of both the predict and the update modules would exceed the memory bandwidth. In the forward transform, the three required memory accesses for the predict module and the output of the update module can be directly accommodated to the picture memory. Figure 3.19 illustrates how the two modules can be synchronized by First-In-First-Out (FIFO) buffers for the forward transform. Following, we explain how data are provided to the inputs of the update module for the forward transform.

Providing data to the γ input of the update module: It can be seen in Figure 3.14 that the γ input of the update module can be fed with the output of the predict module. Introducing a FIFO buffer between the output of predict module and the γ input of the update module, absorbs the unequal delivery and consumption rate of data at the beginning and at the end of the predict and update phases (see Figure 3.19).

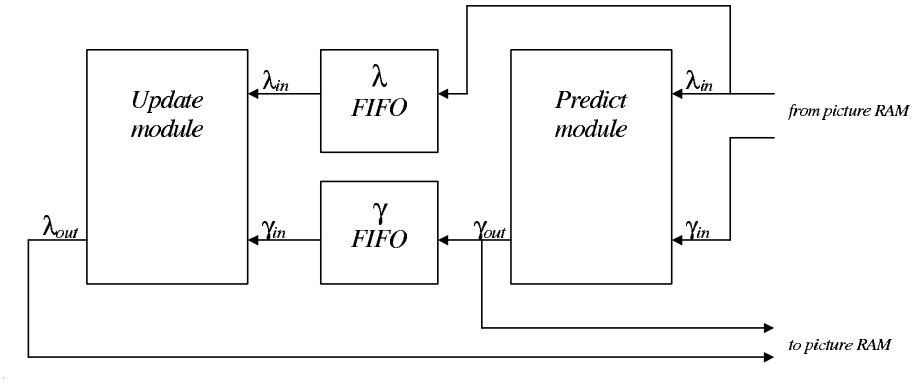


Figure 3.19: Synchronizing FIFO buffers for forward transform.

Providing data to the λ input of the update module: Figure 3.14 shows that the update phase uses the same λs as the predict phase, but at a different instant of time. Consequently, it is not possible to fill the λ pipelines of both predict and update modules with the output of the picture RAM. Placing a FIFO buffer before the λ input of the update module is a solution to this timing problem. This buffer, in fact, solves the synchronization problems and allows the update module to reuse the λs (see Figure 3.19).

In the inverse transform, the synchronization scheme is identical with the only difference that the FIFOs provide data to the inputs of the predict module and the update module receives data from the picture memory.

3.3.4 Design evaluation

Figure 3.20 illustrates the top-level design organization of the complete lifting-based DWT unit. Following, we present hardware synthesis results and performance evaluations.

Synthesis results: The VHDL description of the entire lifting-based DWT unit has been synthesized using Synplify Pro synthesis package (version 7.0) for the Xilinx Virtex II FPGA technology. Table 3.9 presents the hardware resource utilization for the Xilinx x2v1000-5 chip considering a test picture of 64x32 pixels (8 bits gray-scale) and a 4-4 polynomial filter. Except for the number of BRAM blocks (required for the picture memory), the values in Table 3.9 also apply for larger picture dimensions.

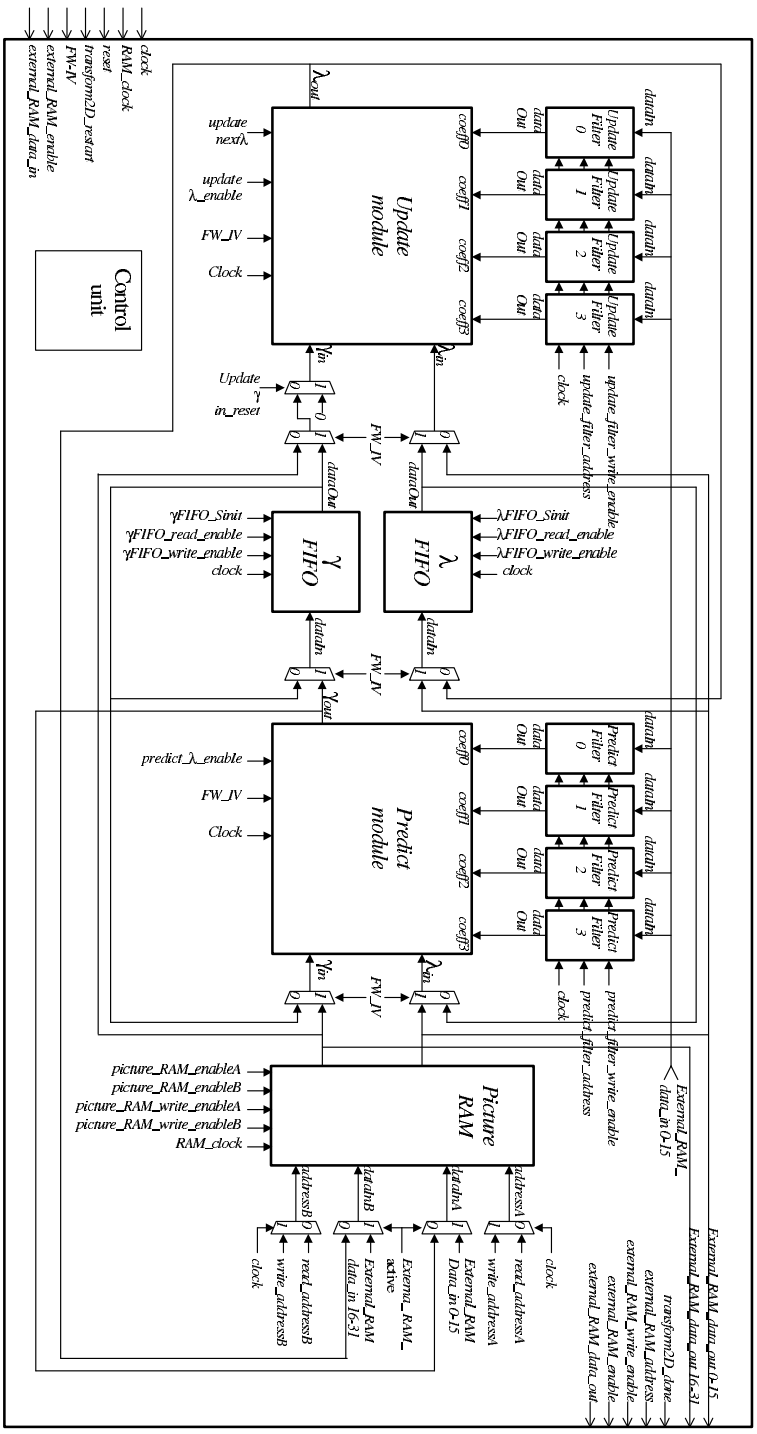


Figure 3.20: Top-level organization of the lifting-based DWT unit.

Table 3.9: Synthesis results for the lifting based DWT unit, 4-4 polynomial filter and a 64x32 picture.

Device	Xilinx x2v1000-5	%
Number of Slices	985 out of 5 120	19%
Number of Slice Flip Flops	669 out of 10 240	6%
Number of 4 input LUTs	1 637 out of 10 240	15%
Number of BRAMs:	22 out of 40	55%
Number of 18x18 multipliers	8 out of 40	20%
Maximum Frequency	$> 5x10^7$ Hz	N.A.

Performance evaluation: We consider two performance evaluation scenarios. First, we evaluate the design for different degrees of the polynomial filters and fixed picture size. Second, we fix the polynomial degrees and evaluate the performance for different picture sizes. The evaluation criteria for the standalone hardware implementation are *transform time* (in clock cycles and in μseconds), *pictures per second*, and *cycles per pixel*. Additionally, we consider a Molen processor implementation. We implement the proposed DWT in the reconfigurable processor, assuming a core MIPS processor operating at 1 GHz. In the considered Molen setup, we estimate the time to perform the DWT on an entire image including all transfers to/from the main memory. Then, we compare the Molen performance results to a pure software sequential execution on a general-purpose MIPS processor, operating at 1 GHz. An existing software application, called LIFTPACK [68], has been used as a reference. The performance has been evaluated by a cycle-accurate simulator of the assumed MIPS processor (more precisely, *sim-outorder* from the SimpeScalar toolset [71]).

The performance results for different polynomial degrees are reported in Table 3.10. Obviously, the hardware transform time increases with the increase of the polynomial degree. What should be noted, however, is that this increase is not as fast as the transform time increase of the pure software implementation. Since we consider constant image size, the data transfer time from the main memory does not depend on the polynomial degree and the Molen execution time follows the pure hardware performance curve. Therefore, as the last row of Table 3.10 suggests, *the higher polynomial degrees of the lifting filters, the more dramatically the Molen processor outperforms the corresponding pure software implementation of the DWT.*

The influence of the picture size on the performance of the proposed unit can be evaluated through Table 3.11, where polynomial filter degrees of 4-4 are assumed. To perform the following evaluations we introduce the term *process-*

Table 3.10: Performance evaluation for polynomial filters of different degrees and a constant picture size of 352x288 pixels.

Clock frequency for the DWT unit is 50 MHz, for the MIPS - 1GHz.

Category	unit	Filter polynomial degree		
Filter degree	-	2-2	4-4	8-8
HW transform time	FPGA cycles	152000	160000	175000
(DWT unit, 50MHz)	μsec	3040	3200	3500
Pictures per sec on FPGA	pic/sec	329	313	286
FPGA cycles per pixel	FPGA cyc/pix	1.5	1.6	1.7
Molen execution*	μsec	3210	3370	3670
Pure SW execution	MIPS cycles	9831292	13395008	22061284
(MIPS 1GHz)	μsec	9831	13395	22061
SW/Molen speedup	-	3.06	3.97	6.01

* Transform time and main memory data transfers included.

Table 3.11: Performance evaluation for different picture sizes and constant polynomial filter degrees of 4-4.

Clock frequency for the DWT unit is 50 MHz, for the MIPS - 1GHz.

Category	unit	Picture size		
Picture format	pixels	176x144	352x288	720x560
HW transform time	FPGA cycles	46000	160000	586000
(DWT unit, 50MHz)	μsec	920	3200	11720
Pictures per sec on FPGA	pic/sec	1087	313	85
FPGA cycles per pixel	FPGA cyc/pix	1.8	1.6	1.5
Molen execution*	μsec	962	3370	12577
Pure SW execution	MIPS cycles	3314925	13395008	63300925
(MIPS 1GHz)	μsec	3315	13395	63301
SW/Molen speedup	-	3.44	3.97	5.03

* Transform time and main memory data transfers included.

ing efficiency as the number of cycles spent per pixel. The lower this number, the higher the efficiency. Filling the pipelines in the beginning of each one-dimensional transform (i.e., transforming a single row or a single column) and updating left- and right- affected boundary γs, leads to cycles in which no output is generated (see Subsection 3.3.3). Furthermore, the parallel execution of the predict and update modules cannot start immediately at the beginning of each one-dimensional transform, but after a number of cycles, when both of the predict outputs become available for the update inputs. Therefore, a better processing efficiency is observed for larger pictures, where the influence of

these constant overhead delays diminishes. This is confirmed by the results in Table 3.11 where the number of hardware cycles spent per pixel is lower for higher picture dimensions (see row 'FPGA cycles per pixel'). Therefore, *for larger pictures, the processing efficiency of the proposed hardware DWT unit grows*. Regarding the Molen organization, the data transfer time from the main memory is proportional to the picture size. However, the relative part of this data transfers is negligible, compared to the DWT transform time, therefore the Molen implementation has virtually the same processing efficiency as the standalone DWT unit. The result is, as suggested in the last row of Table 3.11: *the larger picture dimensions, the more severely the Molen processor (embedding the proposed hardware DWT unit) outperforms the corresponding pure software DWT implementation.*

3.4 Conclusions

In this chapter, we considered three computationally intensive kernels for hardware implementation to accelerate data processing in the higher MPEG-4 profiles. Two hardware approaches to realize the MPEG-4 repetitive padding algorithm in real-time were discussed. The first approach proposed a design of a simple dedicated systolic structure. It is shown that an operating frequency of up to 13.4 MHz allows a processing speed of up to 280 K macroblocks per second to be achieved by only 16 processing elements (PE), mapped on a relatively small Altera FPGA. Evaluations for a Xilinx FPGA indicate that a systolic padding unit with 64 PEs can process approximately 950 K macroblocks per second, thus achieving performance well above the requirements of the the most-demanding MPEG-4 visual profiles and levels. The second approach for a hardware repetitive padding implementation described a scheme for general purpose ALU augmentation, which could accelerate the MPEG-4 padding algorithm by orders of magnitude. We proposed a pipelined implementation of the idea, thus preserving the original functionality and timing scheme of the target ALU. At trivial hardware costs, we could easily achieve real-time performance at the most-demanding profile levels of MPEG-4. We proved that the proposed design is scalable by applying it on ALUs with different operand widths. Another performance demanding part of MPEG-4, the so-called accepted quality (ACQ) function, was also considered in this chapter. A possible systolic hardware implementation was proposed. Evaluations indicate capabilities of processing speeds far beyond the MPEG-4 real-time requirements. More precisely, results indicate capabilities of processing up to 16 129 032

BAB/s. Finally, we introduced a hardware accelerator of the Discrete Wavelet Transform based on the so called lifting scheme. Different performance enhancing design techniques were introduced in the unit, like pipelining, parallel module operation and data reuse. The performance evaluation of the unit suggested that for larger picture sizes its processing efficiency grew. The DWT unit was evaluated as a part of the reconfigurable processor in a Molen organization. Simulation results clearly indicated that for large picture dimensions and long polynomial lifting filters, a Molen organization embedding the DWT unit severely outperforms a pure software DWT implementation (by 3-6 times in the considered scenarios). For the DWT unit alone, an operating frequency of 50 MHz was reported by the Xilinx FPGA synthesis tools and assumed for the simulations. The core Molen processor was chosen to be an out-of-order, MIPS running at 1GHz. All three considered functional accelerators could be implemented as stand-alone units or embedded in a reconfigurable processor. In Chapter 6, we resynthesize the padding and the ACQ custom computing units for the Xilinx Virtex II Pro technology and evaluate their contribution to the performance speedup of a proposed Molen processor prototype. The padding unit and the ACQ function are processing block-organized visual data. The efficient processing of this specific data organization requires large memory throughput, impossible to be obtained by traditional linearly addressable memories. In the chapter to follow, we propose a scalable memory organization capable to deliver block organized data to the relevant processing units at the required throughput speeds.

Chapter 4

Visual Data Rectangular Memory

In this Chapter, we focus on the parallel access of randomly aligned rectangular blocks of visual data, common for various media applications. As an alternative of traditional linearly addressable memories, we suggest a memory organization based on a rectangular array of memory modules. A highly scalable data alignment scheme incorporating module assignment functions and a new generic addressing function are proposed. The addressing function implicitly embeds the module assignment functions and is separable, which potentially enables short critical paths and saves hardware resources. We also discuss the interface between the proposed memory organization and a linearly addressable memory accompanied with comprehensive examples. An implementation, suitable for MPEG-4 is presented and mapped onto an FPGA technology as a case study. Synthesis results indicate reasonably small hardware costs for an exemplary 512×1024 2D addressable space and a range of access pattern dimensions. The design is envisioned to be more cost-effective compared to related works. Regarding performance, our experiments suggest that a speedup of 8X can be expected.

The chapter starts with Section 4.1 giving a short introduction to the problem. Section 4.2 motivates the presented research and formally introduces the particular addressing problem. In Section 4.3, the addressing scheme is described and the corresponding memory organization with a possible implementation are discussed. Case study synthesis results for FPGA technology are reported and related work is compared to our design in Section 4.4. Finally, the chapter is concluded with Section 4.5.

4.1 Introduction

The problems of conflict-free parallel accesses of different data patterns out of a two-dimensional storage have been extensively explored for long time in several research areas. Vector processors designers have been interested in memory systems that are capable of delivering data at the demanding bandwidths of the increasing number of pipelines, see for example [72–75]. Different approaches have been proposed for optimal alignment of data in multiple memory modules [72, 74–78]. Module assignment and addressing functions have been utilized in various interleaved memory organizations to improve the performance. In graphical display systems, researchers have been investigating efficient accesses of different data patterns: blocks (rectangles), horizontal and vertical lines, forward and backward diagonals [78, 79]. While all these patterns are of interest in general purpose vector machines and graphical display systems, rectangular blocks are the basic data structures in visual data compression. The most computationally intensive algorithms, like motion estimation and the discrete cosine transform, operate on square pixel blocks, requiring a significant data throughput. Therefore, the visual data compression standards have narrowed the problems towards block (rectangularly) accessible memories with emphasis on high-performance implementations. Furthermore, to utilize the available bandwidth of a particular machine efficiently, new scalable memory organizations, capable of accessing rectangular pixel patterns are needed. In this chapter, we propose an addressing function for rectangularly addressable systems, with the following characteristics:

- Rectangular sub-arrays out of a two-dimensional data storage can be accessed with high scalability. The addressing is separable, which saves addressing hardware. We also introduce implicit module assignment functions to further improve the designs. Finally, we propose a conflict free data routing circuitry avoiding large critical path penalties.

- Reasonably small hardware costs are shown by an FPGA case study implementation. In our experiments, we consider the maximum available on-chip memory of the Xilinx Virtex II Pro 2vp50ff1152 device, which is sufficient to implement a 512×1024-byte data storage. The proposed implementation requires from as little as 534 slices for 2×4-pixel patterns up to 3287 slices for 8×8 ones, which is between 1% and 13% of the today's reconfigurable device resources considered. Speedups around 8x are estimated for the case study FPGA implementations versus traditional linearly accessible memories.

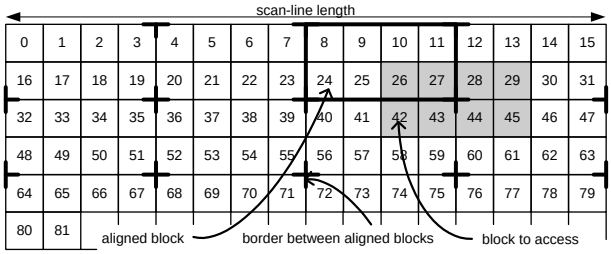
4.2 Motivation

In this section, we consider the memory addressing and accessing problem by considering the MPEG standards. The problems described here, as well as the solutions described later are, however, of a general nature regarding vector rectangular data accessing.

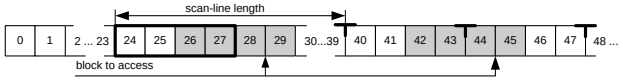
The addressing problem - a motivating example. Most of data processing in MPEG standards is not performed over separate pixels, but over certain regions (blocks of pixels) from a frame. This generates problems with data alignment and access in system memory. To illustrate these problems, let us consider the following *motivating example*. Assume a single port Linearly Addressable Memory (LAM) and a pixel plane divided into blocks with dimensions 4x2, with each pixel represented by a byte. Further, assume that the video information is stored as a scan-line (see Figure 4.1(a)) and that the system is capable of accessing 8 consecutive bytes per cycle. Obviously, neither of the blocks containing pixels {8, 9, 10, 11, 24, 25, 26, 27} and {26, 27, 28, 29, 42, 43, 44, 45} is accessible by a single memory transfer. This is because these blocks are not aligned into consecutive memory locations (see Figure 4.1(b)). Even though the memory system could be accessing all data, because it can access linearly 8 bytes in a single memory cycle, in fact it can access, for example, either bytes {26, 27, 28, 29} or bytes {42, 43, 44, 45}, but not all 8 {26, 27, 28, 29, 42, 43, 44, 45}. Consequently, even though an 8-byte memory bandwidth is available, redundant data fetches can not be avoided.

Another approach to process block-organized data may be to reorder data into the LAM. If we position blocks into consecutive bytes (Figure 4.1(c)), we will be able to access such blocks in a single memory cycle (e.g., pixels {8, 9, 10, 11, 24, 25, 26, 27}). In MPEG, however, some of the most demanding algorithms (e.g., motion estimation) require accessing block data at an arbitrary position in the frame, thus in memory. In the Figure 4.1(c) example, accessing block {26, 27, 28, 29, 42, 43, 44, 45} requires 4 cycles, even though the bandwidth is 8 bytes. This is because only two of its bytes can be accessed in one memory access cycle (i.e., either {26, 27}, or {28, 29}, or {42, 43}, or {44, 45}). Figure 4.1(c) suggests that in such cases data fetching may become even less effective than the scan-line alignment scheme.

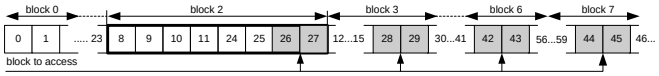
In the rest of the presentation, for conciseness, we will refer to blocks like {8, 9, 10, 11, 24, 25, 26, 27} in Figure 4.1(a) as aligned, and to the remaining blocks (like {26, 27, 28, 29, 42, 43, 44, 45}) as non-aligned. The borders between aligned blocks in the Figure are marked with thick line crosses.



(a) Pixels in a video frame;



(b) Scan-line alignment;



(c) Block-based alignment.

Figure 4.1: Addressing problem in LAM.

Formal problem introduction and proposed solution. Let us assume a LAM with word length of *w* bits (*w* = 8, 16, 32, 64, 128) and the time for linear memory access to be T_{LAM}. The time to access a single $a \times b$ sub-array of 8-bit pixels, depending on its alignment in the LAM will be:
1.) Aligned sub-array: $\frac{8 \cdot a \cdot b}{w} \cdot T_{LAM}$;
2.) Not aligned sub-array: $(\frac{8 \cdot a}{w} + 1) \cdot b \cdot T_{LAM}$.
The time to access *N* $a \times b$ blocks with respect to their alignment will be:
1.) All *N* blocks aligned: $N \cdot \frac{8 \cdot a \cdot b}{w} \cdot T_{LAM}$;
2.) None of the blocks aligned: $N \cdot (\frac{8 \cdot a}{w} + 1) \cdot b \cdot T_{LAM}$;
3.) Mixed: $N \cdot [\frac{1}{a} \cdot \frac{8 \cdot a}{w} + \frac{a-1}{a}(\frac{8 \cdot a}{w} + 1)] \cdot b \cdot T_{LAM} =$

$$= N \cdot (\frac{8 \cdot a}{w} + 1 - \frac{1}{a}) \cdot b \cdot T_{LAM} \tag{4.1}$$

By *mixed* access scenario we mean accessing both aligned and non-aligned blocks. In (4.1), we assume that the probability to access an aligned block is $\frac{1}{a}$, while for a non-aligned block it is $\frac{a-1}{a}$. For simplicity, but without losing generality, assume square blocks of $n \times n$, (i.e., *a*=*b*=*n*). Further assuming N blocks to access, we can estimate the number of LAM cycles as indicated in Table 4.1. Obviously, the number of cycles to access an $n \times n$ block in a LAM is a square function of *n*, i.e., $O(n^2)$.

Table 4.1: Number of LAM cycles in different access scenarios.

all aligned	none aligned	mixed
$\frac{8 \cdot n^2}{w} \cdot N$	$(\frac{8 \cdot n^2}{w} + n) \cdot N$	$(\frac{8 \cdot n^2}{w} + n - 1) \cdot N$

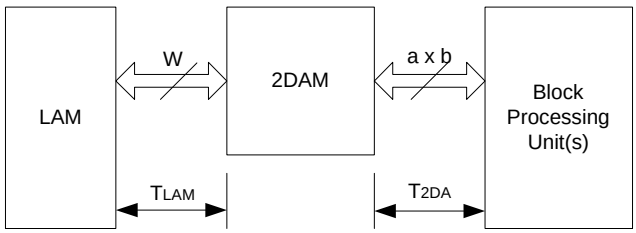


Figure 4.2: Memory hierarchy with 2DAM.

An appropriate memory organization may speed-up the data accesses. Consider the memory hierarchy in Figure 4.2 with time to access an entire $n \times n$ block from the 2D Accessible Memory (2DAM) to be T_{2DA}. In such a case, the time to access *N* $n \times n$ sub-blocks in the mixed access scenario will be:

$$\frac{N}{n} \cdot \frac{8 \cdot n^2}{w} \cdot T_{LAM} + N \cdot T_{2DA}, \quad [sec] \Leftrightarrow$$

$$(\frac{8 \cdot n}{w} + \frac{T_{2DA}}{T_{LAM}}) \cdot N, \text{ [LAM cycles].}$$

That is the sum of the time to access the appropriate number of aligned blocks (i.e., $\frac{N}{n}$) from LAM plus the time to access all *N* blocks from the 2DAM. It is evident that in a mixed access scenario, the number of cycles to access an $n \times n$ block in the hierarchy from Figure 4.2 is a linear function of *n*, i.e., *O(n)* and depends on the implementation of the 2D memory array. Table 4.2 presents access times per single $n \times n$ block. Time is reported in LAM cycles for some typical values of *n* and *w*. Three cases are assumed for LAM:

1. Neither of the *N* blocks is aligned - worst case (WC);
2. Mixed block alignment (Mix.);
3. All blocks are aligned - best case (BC).

The last two columns contain cycle estimations for the organization from Figure 4.2. In this case, both mixed and best case scenarios assume that

Table 4.2: Access time per $n \times n$ block in LAM cycles. $t = \frac{T_{2DA}}{T_{LAM}}$.

n	w	LAM			2DAM	
		WC	Mix.	BC	Mix./BC	WC
8	8	72	71	64	8+t	64+t
	16	40	39	32	4+t	32+t
	32	24	23	16	2+t	16+t
16	8	272	271	256	32+t	256+t
	16	144	143	128	16+t	128+t
	32	80	79	64	8+t	64+t

aligned blocks are loaded from the LAM to the 2DAM first and then non-aligned blocks are accessed from the 2DAM. The 2DAM worst case (contrary to LAM) assumes that all blocks to be accessed are aligned. Even in this worst case, the 2DAM-enabled hierarchy may be better than LAM best case if the same aligned block should be accessed more than once. For example, assume accessing *k* times the same aligned block. In LAM, this would take $k \cdot \frac{8 \cdot n^2}{w} = [\frac{8 \cdot n^2}{w} + (k-1) \cdot \frac{8 \cdot n^2}{w}]$, while with 2DAM, it would cost $[\frac{8 \cdot n^2}{w} + (k-1) \cdot \frac{T_{2DA}}{T_{LAM}}]$ LAM cycles per block. Obviously, to have a 2DAM enabled memory hierarchy, faster than pure LAM, it would be enough if $\frac{8 \cdot n^2}{w} > \frac{T_{2DA}}{T_{LAM}}$. All estimations above strongly suggest that *a 2DAM with certain organization may dramatically reduce the number of accesses to the (main) LAM, thus considerably speeding-up related applications.*

4.3 Block addressable memory

In this Section, we present the proposed mechanism by describing its addressing scheme, the corresponding memory organization and a potential implementation.

Addressing scheme: Assume $M \times N$ image data stored in $k = a \times b$ memory modules ($1 \leq a \leq M; 1 \leq b \leq N$). Furthermore, assume that each module is linearly addressable. We are interested in parallel, conflict-free access of $a \times b$ blocks at any (i,j) location, defined as:

$$B(i,j) = \{I(i+p, j+q) | 0 \leq p < a, 0 \leq q < b\},$$
$$0 \leq i \leq M-a, 0 \leq j \leq N-b.$$

To align data in *k* modules without data replication, we organize these modules in a two-dimensional $a \times b$ matrix. A module assignment function, which maps

a piece of data with 2D coordinates *(i,j)* in memory module $(p, q) : 0 \leq p < a, 0 \leq q < b$, is required. We separate the function denoted as $m_{p,q}(i, j)$, into two mutually orthogonal assignment functions $m_p(i)$ and $m_q(j)$. We define the following module assignment functions for each module at position *(p,q)*:

$$m_p(i) = (i - p)\ mod\ a \tag{4.2}$$

$$m_q(j) = (j - q)\ mod\ b \tag{4.3}$$

The addressing function for module *(p,q)* with respect to coordinates *(i,j)* is defined as:

$$A_{p,q}(i, j) = (i\ div\ a + c_i) \cdot \frac{N}{b} + j\ div\ b + c_j \tag{4.4}$$

$$c_i = \begin{cases} 1, i\ mod\ a > p \\ 0, otherwise. \end{cases} \qquad c_j = \begin{cases} 1, j\ mod\ b > q \\ 0, otherwise. \end{cases}$$

Obviously, if $p = a - 1 \Rightarrow c_i = 0$ for $\forall i$; if $q = b - 1 \Rightarrow c_j = 0$ for $\forall j$, respectively. In essence, c_i and c_j are the module assignment functions, implicitly embedded into the linear address $A_{p,q}(i, j)$.

Example: Consider the motivating example of Section 4.2 and the pixel area from Figure 4.1(a). The same pixel area is mapped into a 2D addressing space with N=16 as depicted in Figure 4.3. In this new mapping, we address data by columns and rows, as 2D addressing is the actual addressing performed at algorithmic level. That is, byte 27 is referred to as *(1,11)*. Consequently, we have to perform the physical memory partitioning and assignment of data. Assume that data will be stored into linearly byte addressable memory modules, organized in a 2×4 matrix. Because in our example we have $5 \times 16 = 80$-byte memory, we subdivide the physical memory into 8 modules in total, 10 bytes each. Each pixel has to be allocated in a specific module by the assignment function. The memory module assignments of all pixels from the considered pixel area for a=2, b=4 are depicted in Figure 4.4(a). In the Figure, the pixel with 2D address *(1,11)* from Figure 4.3 is allocated by the module assignment function in module *(1,3)*. At the second addressing level, the linear address of each individual pixel within the module (*intra-module address*), has to be determined. The addressing function (4.4) generates a unique intra-module address within an uniquely assigned memory module, for each and every byte from the 2D addressing space. The intra-module address of pixel *(1,11)* determined by (4.4) is 2, denoted as A2 in module *(1,3)*, see Figure 4.4(b). Consequently, the proposed addressing scheme is in fact performed at two levels-module assignment and intra-module addressing.

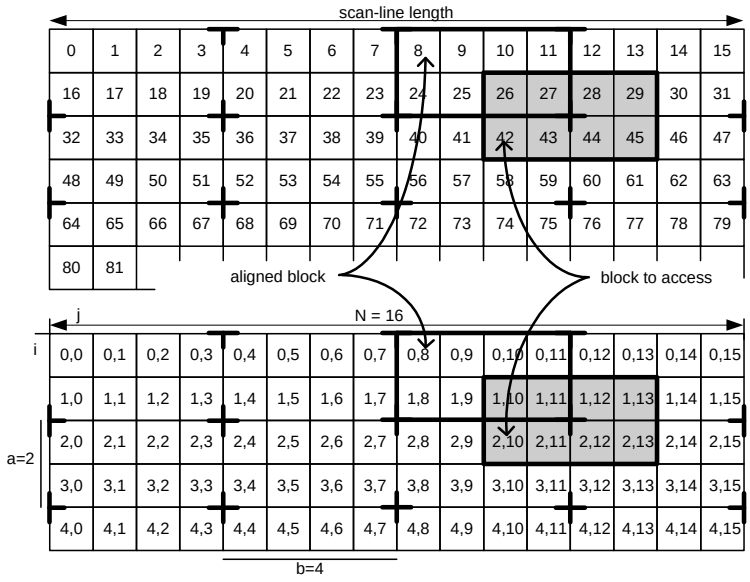
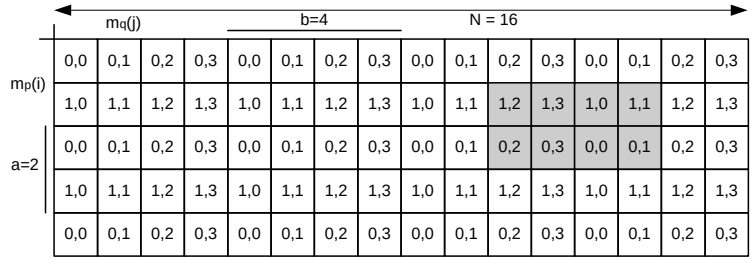


Figure 4.3: Mapping of scan-line organized pixels into a 2D addressing space.

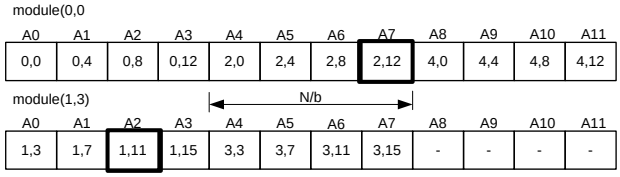
As it has been stated, our scheme addresses and simultaneously accesses entire blocks rather than individual bytes. In the presented example, blocks are of dimension 2×4 bytes. By our definition, blocks are addressed by the 2D coordinates of their upper-left pixels. Consider the shaded non-aligned block $\{26\text{-}45\}$ from the motivating example. This block will be addressed as *B(1,10)*, see Figure 4.3. Note that the pixels of a block are accessed from all 8 modules simultaneously, in parallel. Using (4.2)-(4.4), we can calculate the linear address of the pixels from the considered block for each module *(p,q)* with respect to 2D address *i,j=(1,10)*:

- **module *(p,q)=(0,0)***
$$\left.\begin{array}{l} i \bmod a = 1 > p \Rightarrow c_i = 1 \\ j \bmod b = 2 > q \Rightarrow c_j = 1 \end{array}\right\} \Rightarrow A_{0,0}(1,10) = 7$$
- **module *(p,q)=(1,3)***
$$\left.\begin{array}{l} i \bmod a = 1 = p \Rightarrow c_i = 0 \\ j \bmod b = 2 = q \Rightarrow c_j = 0 \end{array}\right\} \Rightarrow A_{1,3}(1,10) = 2$$

That is, the pixels of block *i,j=(1,10)* will be allocated at address 7 in module *(p,q)=(0,0)* and at address 2 in module *(p,q)=(1,3)*. Identically, the intra-module addresses of the remaining 6 pixels of the considered block can



(a) Module assignments of the 2D pixel area;



(b) 2D addresses and linear addressing within modules.

Figure 4.4: Modules assignment and internal addressing for *a=2*, *b=4*, *N=16*.

be calculated for each of the remaining 6 modules to be $A_{0,1}(1,10) = 7$, $A_{0,2}(1,10) = 6$, $A_{0,3}(1,10) = 6$, $A_{1,0}(1,10) = 3$, $A_{1,1}(1,10) = 3$, $A_{1,2}(1,10) = 2$. Figure 4.4(b) illustrates the internal linear addressing and data alignment within the considered two memory modules. Note that having the intra-module addresses of all pixels in the considered block, we only need to know which module contains the upper-left pixel *(i,j)=(1,10)* to reorder the data properly. The upper-left pixel of block *B(1,10)* is calculated (from the zeroes of (4.2) and (4.3)) to be located in module *(p,q)=(1,2)*. *Thus, having each and every of the 8 block pixels localized in each and every of the 8 modules, we can access the entire block in one cycle by accessing all the modules in parallel.* Yet identically, it can be shown that any 2×4 block, regardless its position (thus including aligned blocks), can be accessed in a single cycle. Recall that block *B(1,10)* is the 2D notation of block $\{26\text{-}45\}$ from the motivating example. This block was accessible in 2 or 4 cycles from a conventional 8-byte LAM, thus 2 to 4 times slower than the proposed scheme at the same bandwidth of 8 bytes per cycle.

Memory organization and implementation: The key purpose of the proposed addressing scheme is to enable performance-effective memory implementations optimized for algorithms requiring the access of rectangular blocks. Designs with shortest critical paths are to be considered with the highest priority, as they dictate machine performance. Equations (4.2)-(4.4) are generally

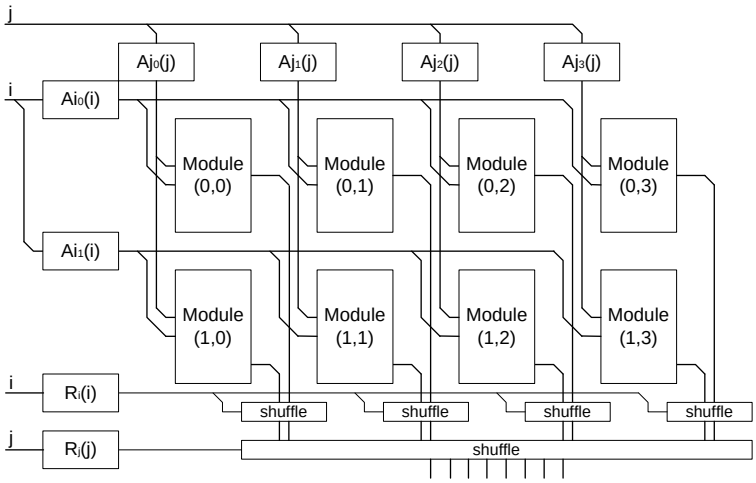
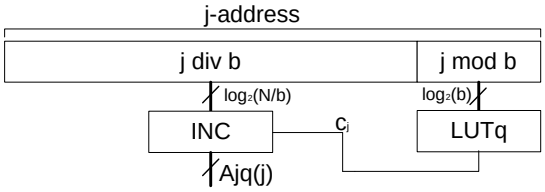


Figure 4.5: 2DAM for *a=2*, *b=4*, and $N = 2^n \geq 16$.

valid for any natural value of parameters *a*, *b* and *N*. To implement the proposed addressing and module assignment functions, however, we will consider practical values of these parameters. Since pixel blocks processed in MPEG algorithms have dimensions up to 16×16, values of practical significance for parameters *a* and *b* are the powers of two up to 16 (i.e., 1, 2, 4, 8, 16). For the particular implementation example we will consider the discussed block size - $a \times b = 2 \times 4$.

Module addressing: An important property of the proposed module addressing function is its *separability*. It means that the function can be represented as a sum of two functions of a *single* and *unique* variable each (i.e., variables i and j). The separability of $A_{p,q}(i,j) = Ai_p(i) + Aj_q(j)$ allows the address generators to be implemented per column and per row (see Figure 4.5) instead of implemented as individual addressing circuits for each of the memory modules. Taking into account the separability of $A_{p,q}(i,j)$ and considering an arbitrary range of picture dimensions to be stored, we can define $C_h = N = 2^n, n \geq 4$ as "horizontal capacity" of the 2DAM (to be discussed later). The requirements for the frame sizes of all MPEG standards and for Video Object Planes (VOPs) [1] in MPEG-4 are constituted to be multiples of 16, thus, *N* is a multiple of 2^4 by definition. Assuming the discussed practical values of N and b, further analysis of Equation (4.4) suggests that $j \; div \; b + c_j < \frac{N}{b}$ and $(j \; div \; b + c_j)_{max} = \frac{N}{b} - 1$, i.e., no carry can be ever generated between $Ai_p(i)$ and $Aj_q(j)$. Therefore, we can implement $A_{p,q}(i,j)$



(a) Generation circuit of q-addresses for $1 \leq q < b$;

j mod b		c_j			i mod a	c_i p=0
		q=0	q=1	q=2		
0	0	0	0	0	0	0
0	1	1	0	0	1	1
1	0	1	1	0	-	-
1	1	1	1	1	-	-

(b) LUTs contents for *a=2*, *b=4*.

Figure 4.6: Module address generation.

for every module *(p,q)* by simply routing signals to the corresponding address generation blocks without actually summating $Ai_p(i) + Aj_q(j)$. Figure 4.6(a) illustrates address generation circuitry of q-addresses ($Aj_q(j)$) for all modules except the first ($1 \leq q < b$). With respect to (4.4), if c_j is 1 the quotient *j div b* should be incremented by one, otherwise it should not be changed. To determine the value of c_j, a Look-Up-Table (LUT) with *j mod b* inputs can be used. For the assumed practical values of *a* and *b* (≤ 16), such a LUT would have at most 4 inputs, i.e., *c_j is a binary function of at most 4 binary digits*. Row p-addresses are generated identically. For *p=1* or *q=3*, $c_i = 0$, $c_j = 0$ respectively. Therefore, address generation in these cases does not require a LUT and an incrementor. Instead, it is just routing *i div a* and *j div b* to the corresponding memory ports, i.e., blocks $Ai_1(i)$ and $Aj_3(j)$ in Figure 4.5 are empty. Figure 4.6(b) depicts all 4 LUTs for the case $a \times b = 2 \times 4$. The usage of LUTs to determine c_i and c_j is not mandatory, fast pure logic can be utilized instead. However, we use LUTs for two main purposes: 1.) to illustrate the design concept; and 2.) LUTs fit better in the FPGA implementation considered further in this chapter.

Data routing circuitry: In Figure 4.5, the shuffle blocks, together with blocks $R_p(i)$ and $R_q(j)$, illustrate the data routing circuitry. The shuffle blocks are in essence circular barrel shifters, i.e. having the complexity of a network of multiplexors. An $n \times n$ shuffle is actually an $n \rightarrow 1$ n-way multiplexor. In

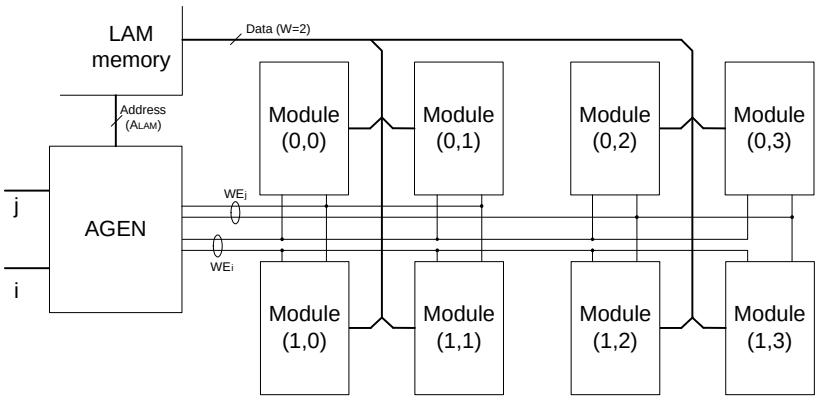


Figure 4.7: LAM interface for *W=2, a=2, b=4*.

the example from Figure 4.5, the *i*-level shuffle blocks are four ($2 \rightarrow 1$) 16-bit multiplexors and the *j*-level one is ($4 \rightarrow 1$) 64-bit. To control the shuffle blocks, we can use the module assignment functions for $p = q = 0$, i.e., $R_i(i) = i \ mod \ a$ and $R_j(j) = j \ mod \ b$. These functions calculate the *(p,q)*-coordinates of the "upper-left" pixel of the desired block, i.e., pixel *(i,j)*. For the assumed practical values of *a* and *b* being powers of two, the implementation of $R_i(i)$ and $R_j(j)$ is simple routing of the least-significant $log_2(a)$ -bits (resp. $log_2(b)$) to the corresponding shuffle level.

2DAM capacity: Earlier, we have defined the "horizontal capacity" of 2DAM as $C_h = N = 2^n, n \geq 4$. C_h is the *maximal scanline length in bytes (pixels), the 2DAM can store without addressing conflicts*. The "vertical capacity" of 2DAM is denoted as C_v and defined as the *maximal number of C_h-byte (C_h-pixel) scanlines the 2DAM can store*. Finally, *the capacity C_{2DAM} of a 2DM is defined as the couple ($C_h \times C_v$)-bytes (pixels)*, rather than as a single number.

LAM interface: Figure 4.7 depicts the organization of the interface between LAM and 2DAM (recall Figure 4.2) for the modules considered in Figure 4.5. Data bus width of the LAM is denoted by W (in number of bytes). In the particular example, W is assumed to be 2, therefore modules have coupled data busses. For each *(i,j)* address, the AGEN block sequentially generates addresses to the LAM and distributes write enable (WE) signals to a corresponding module couple. Two module WE signals (WE_i, WE_j) are assumed for easier row and column selection. In the general case, the AGEN block should sequentially generate $\frac{a \cdot b}{W}$ LAM addresses for each *(i,j)* address. Provided that pixel data is stored into LAM in scan-line manner, the set of LAM addresses to be generated is defined as follows:

$$A_{LAM}(i,j) = \{a \cdot (i\ div\ a) + k\} \cdot N + b \cdot (j\ div\ b) + l \cdot W$$

Which, assuming that only aligned blocks will be accessed from the LAM (i.e., *(i,j)* are aligned), can be simplified:

$$A_{LAM}(i,j) = (i+k) \cdot N + j + l \cdot W \tag{4.5}$$

$$k = 0, 1, ..., a-1; l = 0, 1, ..., \tfrac{b}{W} - 1.$$

In the 2DAM, the data words should be simultaneously written in modules:

$$(p,q) = (k, l \cdot W), (k, l \cdot W + 1), ..., (k, l \cdot W + W - 1) \tag{4.6}$$

at local module address:

$$A_{p,q}^{LAM}(i,j) = (i\ div\ a) \cdot \frac{N}{b} + j\ div\ b. \tag{4.7}$$

Note, that accessing only aligned blocks from the LAM enables thorough bandwidth utilization. When only aligned blocks are addressed, all address generators issue the same address, due to (4.4). Therefore, during write operations into 2DAM, the same addressing circuitry can be used as for reading. If the modules are true dual port, the write port addressing can be simplified to just proper wiring of both *i* and *j* address lines because the incrementor and the LUTs from Figure 4.6(a) are not required. Therefore, module addressing circuitry is not depicted in Figure 4.7.

Addressing consistency: In the following, we will prove that the described scheme provides a consistent LAM and 2DAM addressing. It means that each and every byte is allocated in the same memory module and at the same intra-module address by both LAM and 2DAM addressing schemes.

Lemma 4.1 $x\ mod\ z = x - n \cdot z\ \ iff\ 0 \le x - n \cdot z < z;\ \forall x, n, z \in N.$

Proof. 1. If $x\ mod\ z = x - n \cdot z\ \Rightarrow\ 0 \le x - n \cdot z < z;\ \forall x, n, z \in N$ is true by the definition of *mod* operation. 2. If $0 \le x - n \cdot z < z \Rightarrow x\ mod\ z = x - n \cdot z;\ \forall x, n, z \in N$. Let $x\ mod\ z = x - p \cdot z$. Then, by definition $0 \le x - p \cdot z < z$. Assume $p \ne n \Rightarrow |p - n| \ge 1$. We derive the system:

$$\left\| \begin{array}{l} 0 \le x - n \cdot z < z \\ 0 \le x - p \cdot z < z \end{array} \right.$$

The only solution of the system $p = n$ contradicts to the assumption ■

Lemma 4.2 $(x-y)\ mod\ z = (x\ mod\ z - y)\ mod\ z; \forall y < z; \forall x, y, z \in N$

Proof. By definition $x\ mod\ z = x - n1 \cdot z$ and $(x\ mod\ z - y)\ mod\ z = (x\ mod\ z - y) - n2 \cdot z$. $\Rightarrow$ By substitution and based on Lemma 4.1, we derive: $(x\ mod\ z - y)\ mod\ z = (x - n1 \cdot z - y) - n2 \cdot z = (x-y) - (n1+n2) \cdot z = (x-y)\ mod\ z$ ■

Lemma 4.3 $(x\ div\ y) \cdot\ y = x - x\ mod\ y$

Proof. $\left\| \begin{array}{l} x\ mod\ y = p \\ x\ div\ y = k \\ k \cdot y + p = x \end{array} \right. \Rightarrow \begin{array}{l} (x\ div\ y) \cdot y = \\ = k \cdot y = x - p = \\ = x - x\ mod\ y \end{array}$ ■

Theorem 4.1 (Consistency between the 2DAM and the LAM addressing schemes). *Assume the 2DAM and LAM addressing interface schemes defined by (4.2)-(4.4) and (4.5)-(4.7), respectively. Any byte $(i^{'}, j^{'})$ is allocated in the same memory module at the same intra-module address by both addressing schemes.*

Proof. Consistency of module assignments. Consider byte $(i^{'}, j^{'})$. In consistence with (4.5), we define $k = i^{'}\ mod\ a$ and $l = (j^{'}\ mod\ b)\ div\ W$. Considering the LAM interface and Lemma 4.3, the module, where byte $(i^{'}, j^{'})$ should be stored is calculated as follows:

$$\begin{array}{c} (p,q) = (k, l \cdot W + (j^{'}\ mod\ b)\ mod\ W) = \\ = (k, \{(j^{'}\ mod\ b)\ div\ W\} \cdot W + (j^{'}\ mod\ b)\ mod\ W) = \\ = (k, (j^{'} mod\ b) - (j^{'} mod\ b) mod W + (j^{'} mod\ b) mod W) \end{array}$$

$$\Rightarrow (p,q) = (k, j^{'}\ mod\ b) \tag{4.8}$$

Consider (4.2)-(4.3) for the 2DAM module allocation and Lemma 4.2, then:

$$\begin{array}{c|c} m_p(i^{'}) = & m_q(j^{'}) = \\ = (i^{'} - p)\ mod\ a = 0 & = (j^{'} - q)\ mod\ b = 0 \\ (i^{'} mod\ a - p)\ mod\ a = 0 & (j^{'} mod\ b - q)\ mod\ b = 0; \\ (k-p) mod\ a = 0; k < a & j^{'} mod\ b\ < b \end{array}$$

$$\Rightarrow p = k;\ q = j^{'} mod\ b \tag{4.9}$$

Equations (4.8) and (4.9) indicate that any byte $(i^{'}, j^{'})$ will be allocated in the same memory module both by the LAM interface and by the 2DAM read circuitry.

Consistency of intra-module addresses. Assume *(i,j)* is the aligned block, containing byte $(i^{`}, j^{`})$, i.e., $i\ div\ a = i^{`}\ div\ a$, $j\ div\ b = j^{`}\ div\ b$.
Consider (4.4): $A_{p,q}(i^{`}, j^{`}) = (i^{`}\ div\ a + c_i) \cdot \frac{N}{b} + j^{`}\ div\ b + c_j$,
from (4.9): $p = i^{`}\ mod\ a$ and $q = j^{`}\ mod\ b \Rightarrow c_i = c_j = 0, \Rightarrow$
$A_{p,q}(i^{`}, j^{`}) = (i^{`}\ div\ a) \cdot \frac{N}{b} + j\ `\ div\ b\ \Rightarrow$(Recall the assumption)
$A_{p,q}(i^{`}, j^{`}) = (i\ div\ a) \cdot \frac{N}{b} + j\ div\ b$, identical to (4.7) ■

Example: We consider a single (arbitrary chosen) byte and show that it is allocated in the same memory module and at the same intra-module address both by the LAM and by the 2DAM addressing schemes.

Assume that visual data is scan-line aligned in LAM with word length of 2 bytes and big-endian convention. Consider the byte with 2D address *(1,11)*, see Figure 4.3. The memory hierarchy of Figure 4.2 indicates that byte *(1,11)* has to be loaded from the LAM into the 2DAM by means of the proposed LAM interface. Assuming that the 2DAM is first loaded in its entirety, all aligned blocks of the considered 5×16-byte area are to be loaded from the LAM into the 2DAM. Byte *(1,11)* is assigned in the LAM as part of aligned block *(0,8)*. The LAM addresses of the four 2-byte words containing the pixels of the block are $A_{LAM} = 8, 10, 24, 26$, see Figure 4.3. The LAM address of the 2-byte word, containing the considered pixel *(1,11)* is calculated from (4.5) to be: $A_{LAM}(0,8)_{k=1,l=1} = (0+1) \cdot 16 + 8 + 1 \cdot 2 = 26$. Recall Figure 4.3, where byte *(1,11)* had LAM address 27. Thus, in the assumed big-endian LAM convention, the considered byte 27 is the most significant byte of the 2-byte memory word aligned at address 26. Considering (4.6), this 2-byte word should be stored into modules *(1,2)* and *(1,3)*, see Figure 4.7. The most significant byte, i.e., byte 27, should be stored into module $(p,q)_{k=1,l=1} = (k, l \cdot W + W - 1) = (1,3)$. Its intra-module address with respect to the LAM interface is calculated from (4.7) to be:

$$A_{1,3}^{LAM}(0,8) = (0\ div\ 2) \cdot \frac{16}{4} + 8\ div\ 4 = 2$$

That is, *byte* (1,11) *with LAM address 27, will be stored by the LAM-to-2DAM interface into module (1,3) at intra-module address 2.*

Consider the 2DAM addressing scheme, the shaded non-aligned block *(1,10)* in Figure 4.3 and Figure 4.4, and (4.2)-(4.4). Indeed, *considering the 2DAM addressing scheme, byte* (1,11) *can be read from address location 2 of module* (1,3), as it was shown in the previous example.

Critical paths: Regarding the performance of the proposed design, we should consider the created critical path penalty. Assuming generic synchronous memories where addresses are generated in one cycle and data are available

in another, we separate the critical paths into two: address generation and data routing. For the proposed circuit implementation, the address generation critical path (CP_A) is determined by:

$$CP_A = max(CP_{add\frac{M}{a}}, CP_{add\frac{N}{b}}) + CP_{LUT}.$$

That is the critical path of either a $log_2(\frac{M}{a})$-bit or a $log_2(\frac{N}{b})$-bit adder, whichever is longer, and the critical path of one (max. 4-input) LUT. The data routing critical path (CP_D) is:

$$CP_D = CP_{mux_a} + CP_{mux_b}.$$

That is, the sum of the critical paths of one $a \rightarrow 1$ multiplexor and one $b \rightarrow 1$ multiplexor.

4.4 Experimental results and related work

In the previous sections, we have considered the theoretical aspects of our proposal illustrated by simplified examples. In this section, an experimental case study for a number of real-world FPGA-based designs is presented, followed by a comparison to other related works reported in literature.

Case study: A generic VHDL model of the memory organization has been developed and synthesized for the recent Virtex II Pro FPGA technology of Xilinx. We assume reconfigurable technology for two reasons. First, showing the viability of the organization in reconfigurable technology also proves its viability to all other current and near future technologies. Second, we envision, for cost-efficiency, that assuming MPEG specific requirements, the organization may be incorporated in a reconfigurable augmented processor [39]. Table 4.3 contains synthesis results for the 2vp50ff1152 FPGA device (the last column displays some of the resources available on the chip). The on-chip memory volume allows frames or VOPs sized up-to *512x1024* pixels to be stored. It should be noted that more than one frame can be stored in the memory and accessed, depending on the particular frame format. For example, up-to fourteen CIF frames (*144x176*) can be stored into the implemented *512x1024* storage. This issue is much more beneficial in MPEG-4, where the arbitrary shaped VOPs to be stored vary both in size and number for each particular codec session. Synthesis data for practical MPEG pattern sizes of *2x4, 4x8, 8x8 and 16x16-pixels* indicate that respective structures can be efficiently

Table 4.3: Synthesis for frames up-to 512x1024 (device 2vp50ff1152).

$a \times b$	2 x 4	4 x 8	8 x 8	16 x 16	Avail.
2-1mux	192	1280	3072	16384	N.A.
Adders:	4	10	14	30	N.A.
bits/#	8/1	7/3	6/7	5/15	N.A.
bits/#	8/3	7/7	7/7	6/15	N.A.
# Slices	534	1512	3287	15408	24640
%	1	6	13	63	100
# LUT4	928	2630	5723	26805	49280
%	1	5	11	54	100
IOs	100	292	548	2084	756
BRAM	8x 64K	32x 16K	64x 8K	256x 2K	522K

implemented with a fraction of the available FPGA resources. Only the *16x16* pattern creates a resource conflict with regard to the available IO pins of the chip. This conflict, however, should not be considered as a problem, since structures with bandwidth of that magnitude are usually intended for on-chip implementations. In the 'Adders' rows of Table 4.3, the notation 'bits/#' denotes the number of bits in an adder and the corresponding number of such adders, respectively. Results indicate that in the most common case of 8×8 block patterns, 3287 Virtex II Pro slices are required, which is 13% of the 2vp50ff1152 FPGA device resources.

In Table 4.4, transfer speedup estimations are presented, assuming $T_{LAM} = 10ns$. Calculations are made according to the figures and notations presented in Table 4.2. In BC, all blocks are assumed to be non-aligned, while in WC the very unlikely scenario that all blocks are aligned and accessed only once is considered. T_{2DA} values are derived from the synthesis reports for the designs considered in Table 4.3. Figures in Table 4.4 indicate that even in the unfavorable case when 2DAM is slower than the LAM, considerable transfer speedups of up to 8x can be achieved, due to the proposed memory organization.

Related work: Accessing blocks of memory has been in the hearts of vector (array) processors researchers and developers for long time. Two major groups of memory organizations for parallel data access have been reported in literature - organizations with and without data replication (redundancy). We are interested only in those without data replication. Another division is made with respect to the number of memory modules - equal to the number of accessed data points and exceeding this number. Organizations with a prime number of

Table 4.4: Estimated transfer speedups for $T_{LAM} = 10ns$.

$a \times b$	T_{2DA}	t= $\frac{T_{2DA}}{T_{LAM}}$	w	Transfer speedup		
				BC	Mix.	WC
8x8	16,7ns	1,67	8	7,45	7,34	0,97
			16	7,05	6,88	0,95
			32	6,54	6,27	0,91
16x16	18,8ns	1,88	8	8,03	8,00	0,99
			16	8,05	8,00	0,99
			32	8,10	8,00	0,97

memory modules can be considered as a subset of the latter. An essential implementation drawback of such organizations is that their addressing functions are non-separable and more complex, thus slower and costly to implement. We have organized our comparison with respect to block accesses, discarding other data patterns, due to the specific requirements of visual data compression. It should be noted, however, that our design can be easily augmented to accesses horizontal and vertical $a \times b$ lines, just by slightly modifying the module assignment functions and preserving the same addressing function. To compare designs, two basic criteria have been established: scalability and implementation drawbacks in terms of speed and/or complexity. Comparison results are reported in Table 4.5.

Budnik and Kuck [72] described a scheme for conflict free access of $\sqrt{N} \times \sqrt{N}$ square blocks out of $N \times N$ arrays, utilizing $m > N = 2^n$ memory modules, where *M* is a prime number. Their scheme allows the complicated full crossbar switch as the only possibility for data alignment circuitry and many costly *modulo(M)* operations with M not a power of two. In a publication, related to the development of the Burroughs Scientific Processor, Lawrie [74] proposes

Table 4.5: Comparison to other proposed schemes.

Related Work	scalability	# modules	implementation drawbacks or limitations
Budnik, Kuck [72]	$\sqrt{N} \times \sqrt{N}$ from $N \times N$	prime $m > N = 2^n$	*mod(m)*, crossbar, no addressing
Lawrie [74]	$\sqrt{N} \times \sqrt{N}$	$m = 2.N; N = 2^{2n+1}$	*mod(m)*, no addressing
Voorhis, Morin [75]	$p \times q$ from $M \times N$	$m \geq p \times q$	not separable,*mod(pq)*,*mod(pq+1)*,
Kim, Prasanna [76]	$\sqrt{N} \times \sqrt{N}$ from $N \times N$	*m = N*	certain blocks are inaccessible
De-lei Lee [77]	$\sqrt{N} \times \sqrt{N}$ from $N \times N$	*m = N*	many modules for higher *N*
Sproull et al. [79]	8×8	8×8	time-space multiplexing, not general
Park [78]	$p \times q$ from $M \times N$	prime $m > p \times q$	not separable, many adders, big LUTs
HiPAR-DSP [80, 81]	$N \times N$	$m = (1 + N)^2$	$2 \times N + 1$ additional modules, *mod(m)*
HiPAR-DSP16 [82]	$p \times q$ from $M \times N$	$m >> p \times q$	big number of modules, *mod(m)*
This proposal	$p \times q$ from $M \times N$	$m = p \times q$	none of the above, rectangular patterns only

an alignment scheme with data switching, simpler than a crossbar switch, but still capable to handle only $\sqrt{N} \times \sqrt{N}$ square blocks out of *m=2N* modules, where $N = 2^{2n+1}$. Both schemes in [72] and [74] require larger number of modules than the number of simultaneously accessed (image) points (*N*). Furthermore, in both papers authors do not describe the necessary addressing circuitries for their schemes. Voorhis and Morin [75] suggest various addressing functions considering $p \times q$ subarray accesses and different number of memory modules M: both $m = p \times q$ and $m > p \times q$. Neither of the functions proposed in [75] is separable, which leads to an extensive number of address generation and module assignment logic blocks. In [76] authors propose a module assignment scheme based on Latin squares, which is capable of accessing $\sqrt{N} \times \sqrt{N}$ square blocks out of $N \times N$ arrays, but not from random positions. Similar drawbacks has the scheme proposed in [77]. One early graphical display system, described in [79], can be considered a partial case of our scheme, since authors describe square 8×8 submatrix accesses and memory alignment similar to the proposed in our scheme. The authors in [79] did not consider rectangular subarray accesses, which are not directly deducible from the proposed reading. No formalization of the addressing functions was presented either. A more recent display system memory, capable of simultaneous access of $p \times q$ rectangular subarrays is described in [78]. The design, proposed there, utilizes a prime number of memory modules, which enables accesses to numerous data patterns, but disallows separable addressing functions. Therefore, regarding block accesses, it is slower and requires more memory modules than our proposal. Large LUTs (in size and number) and yet longer critical path with consecutive additions can be considered as other drawbacks of [78]. A memory organization, capable of accessing $N \times N$ square blocks, aligned into $(1 + N)^2$ memory modules was described in [80]. The same scheme was used for the implementation of the matrix memory of the first version of HiPAR-DSP [81]. Besides the restriction to square accesses only, that memory system uses a redundant number of modules, due to additional DSP-specific access patterns considered. A definition of rectangular $p \times q$ block random addressing scheme from the architectural point of view dedicated for multimedia systems was introduced in [83], but no particular organization was presented there. In the latest version of HiPAR16 [82], the matrix memory was improved so that a restricted number of rectangular patterns could also be accessed. This design, however, still uses excessive number of memory modules as *p* and *M* respectively *q* and *N* should not have common divisors. E.g., to access the example 2×4 pattern, the HiPAR16 memory requires $3 \times 5 = 15$ memory modules, instead of 8 for our proposal. The memory

of [82] would require more-complicated circuitry. Similarly to [79], [81, 82] assume separability, however, the number of utilized modules is even higher than the closest prime number to $p \times q$. Compared to [72, 74, 76–82], our scheme enables higher scalability and lower number of memory modules. This reflects directly to the design complexity, which has been proven to be very low in our case. Address function separability reduces the number of address generation logic and critical path penalties, thus enables faster implementations. Regarding address separability, we differentiate from [72, 74–78], where address separability is not supported. As a result, *our design is envisioned to have the shortest critical path penalties among all referenced works.*

4.5 Conclusions

We presented a scalable memory organization capable of addressing randomly aligned rectangular data patterns out of a virtual 2D data storage. High performance is achieved by reduced number of data transfers between memory hierarchy levels, efficient bandwidth utilization, and short hardware critical paths. In the proposed design, data are located in an array of byte addressable memory modules by an addressing function, implicitly containing module assignment functions. An interface to a linearly addressable memory has been provided to load the array of modules. Theoretical analysis proving the consistency and efficiency of the linear and the two-dimensional addressing schemes has been also presented. The implementation of the organization was evaluated by experimental synthesis. Results indicate that a scalable range of such organizations can be efficiently mapped on recent FPGA technologies. At reasonably small hardware costs, we achieved considerable speedups of up to 8X for an experimental case study design versus traditional linearly addressable memories. The design is envisioned to be more cost-effective compared to related works reported in literature. The proposed organization is intended for specific data intensive algorithms in visual data processing applications, but can also be adopted by other general purpose applications with high data throughput requirements including vector processing. In the chapter to follow, we present A Virtex II Pro Molen prototype, which can embed the proposed memory organization at microarchitectural level, thus supporting some of the block processing CCUs implemented.

Chapter 5

The Xilinx Virtex II Pro Prototype

The original Molen proposal [39] assumes that the arbiter is actually the standard instruction decoding unit of the core processor, augmented with additional functionality. This assumption, however, implies that the Molen designer can access the internal structure of the core processor, which in many cases may become an implementation prohibitive requirement. Given that current reconfigurable technology does not allow such a design, we investigate in this dissertation whether it is possible to implement the Molen paradigm without changing the design of the core processor. In this chapter, referring to the core processor as to a ”black box”, we implement the Molen paradigm by emulating reconfigurable operations with the ISA of the targeted core processor. An important challenge is to preserve the closely coupled GPP-RP organization and to achieve performance efficient processing. We have chosen the Virtex II Pro platform of Xilinx [84] as a target FPGA technology for our prototype design. An implementation of the minimal functionally complete Molen πISA is proposed, comprising the instructions: **set**, **execute**, **movtx**, and **movfx**. The discussion is focused on the microarchitectural support for the implemented πISA, emulated on the embedded PowerPC 405 processor in the Virtex II Pro FPGA. Some important considerations regarding the software support of the Molen prototype are discussed as well.

The prototype designs of the key Molen units are presented in separate sections of this chapter. Section 5.1 deals with the design aspects of the prototype arbiter and its implementation description. The $\rho\mu$-code unit prototype implementation is presented in Section 5.2 including some discussion on important

$\rho\mu$-code manipulations. The design considerations behind the implementation of the exchange registers, the memory organization, and the clock domains are introduced in Section 5.3. Section 5.4 describes the interface, a designer must implement, to embed a CCU in the prototype organization. The software code annotations required to support the reconfigurable functionality of Molen are explained in Section 5.6. The chapter is concluded in Section 5.7.

5.1 The arbiter

In this section, we introduce the design of an arbiter, which is a potentially performance limiting unit of the Molen $\rho\mu$-coded processor. An analysis of the general requirements to a Molen arbiter is presented, followed by specific software considerations, architectural solutions, implementation issues and a functional testing routine for the proposed prototype arbiter. The complete arbiter design has been described in VHDL and synthesized for the Xilinx VirtexII Pro technology. Synthesis results indicate a very low hardware utilization, namely: *less than 1% of the reconfigurable hardware resources available on the selected prototyping VirtexII Pro FPGA chip (xc2vp20).*

The section is organized as follows. We start with Subsection 5.1.1 describing the design requirements to a general arbiter. In Subsection 5.1.2, software and hardware considerations for the particular arbiter design for PowerPC and Virtex II Pro FPGA are presented. Subsection 5.1.3 discusses functional testing and presents hardware synthesis results for the specific implementation of the arbiter.

5.1.1 General requirements to the arbiter.

The arbiter is a unit, which directs instructions either to the GPP or to the RP, controls their proper co-processing, and manages the main memory access. Thus, the arbiter is closely connected to three major units of the $\rho\mu$-coded processor, namely the GPP, the memory and the RP (more precisely to the $\rho\mu$-code unit). Each of these parts of the organization has its own requirements that should be considered when an arbiter is designed. Regarding the core processor the arbiter should:

- Preserve the original behavior of the core processor when no reconfigurable instruction is executed. Create the shortest possible critical path penalties on these executions.

- Emulate reconfigurable instruction execution behavior on the core processor using its original instruction set and/or other architectural features.
- Reconfigurable instructions should be encoded in consistence with the instruction encoding of the targeted general purpose architecture.

Regarding the $\rho\mu$-code unit the arbiter should:

- Distribute control signals and the address of the microcode to be loaded or executed to the $\rho\mu$-unit.
- Consume minimal hardware resources. This is crucial if the arbiter is mapped together with the $\rho\mu$-unit on the same FPGA. Thus more reconfigurable resources will be available for the CCU.

For proper memory management, the arbiter should be designed to:

- Arbitrate the data access between the $\rho\mu$-unit and the core processor.
- Allow speeds within the capabilities of the utilized memory technology, i.e., not creating performance bottlenecks in memory transfers.

The arbiter should also provide proper timing for reconfigurable instruction execution to all units referred above.

πISA emulation: We already noted that in the original Molen proposal it was assumed that the GPP instruction decoding unit is augmented with arbitrating capabilities to support the Molen architecture. This assumption, however, limits the applicability of the approach only to designers who have access to the internal structure of the targeted GPP. We propose another design approach, namely: *emulate πISA instructions with the original ISA of the targeted GPP architecture*. In Figure 5.1, a general view of an arbiter organization supporting such πISA emulation is presented. The operation of such an arbiter is entirely based on decoding the input instruction flow. The unit either redirects these instructions, or generates a dedicated sequence of instructions to control the state of the core processor during reconfigurable operations. In such an organization, the critical path penalty to the original instruction flow can be reduced to just one 2-1 multiplexer, thus negligible. Once either of the reconfigurable instructions has been decoded, the following actions are initiated:

1. Arbiter emulation instructions are multiplexed to the processor instruction bus. These instructions emulate reconfigurable instruction execution by driving the processor into wait or halt state.

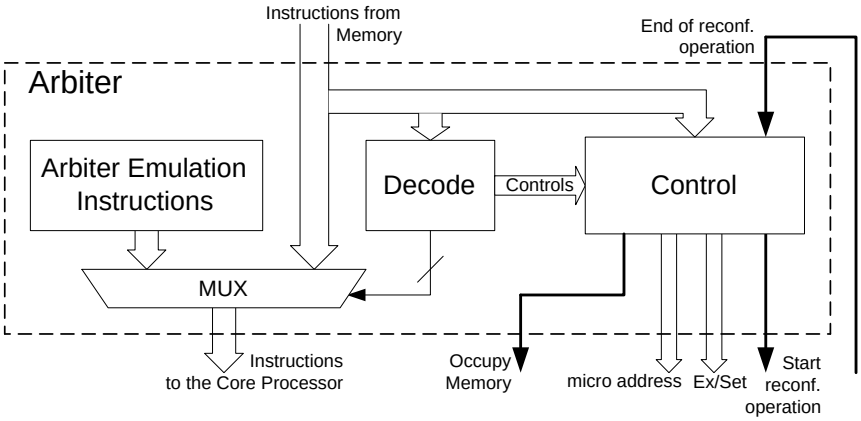


Figure 5.1: General organization of the proposed πISA emulating arbiter.

2. Control signals from the decoder are generated to the control block in Figure 5.1. Based on these controls, the control block performs the following:

   - Redirects the microcode location address of the corresponding reconfigurable instruction to the $\rho\mu$-unit.
   - Generates an internal code signal (Ex/Set) for the decoded reconfigurable instruction and delivers it to the $\rho\mu$-unit.
   - Initiates reconfigurable operation by generating *'start reconf. operation'* signal to the $\rho\mu$-unit.
   - Reserves the data memory control for the $\rho\mu$-unit by generating *memory occupy* signal to the (data) memory controller.
   - Enters a wait state until signal *'end of reconf. operation'* arrives.

An active *'end of reconf. operation'* signal initiates the following actions:

1. Data memory control is released back to the core processor.
2. An instruction sequence is generated to ensure proper exiting of the core processor from the wait state.
3. After exiting the wait state, the control of the program flow is transferred back to the instruction immediately after the reconfigurable instruction, which has been executed last.

5.1.2 Arbiter implementation

Software considerations: In the proposed prototype design, the core processor is driven into a wait state, while a reconfigurable operation is executing. This wait state is initiated, maintained, and eventually discontinued by arbiter emulation instructions. Due to performance reasons, however, we decided instructions related to the special operating modes of PowerPC not to be used for emulation. We motivate this decision with the argument that exiting such special operating modes is usually performed by interrupt, which would slow down our implementation. Furthermore, we may lose generality if we implement the design with instructions strictly specific for the chosen architecture. Therefore, we decided to emulate the wait state of the GPP during reconfigurable operations by unconditional branch instructions. Such instructions are implemented in almost all popular GPP architectures. From the PowerPC instruction set we employed the *'branch to link register'* (**blr**) and *'branch to link register and link'* (**blrl**) to emulate a wait state and to get the processor out of this state. The difference between these instructions is that **blrl** modifies the link register (LR), while **blr** does not. For both instructions, the next instruction address is the effective address of the branch target, stored in LR. When **blrl** is executed, the new value loaded into LR is the address of the instruction following the branch instruction. To drive the processor into a wait state we utilize instruction **blr**, while to 'wake it up' we use **blrl**. Thus the emulation instructions, stored into the corresponding block in Figure 5.1 are reduced to only one instruction for wait and one for 'wake-up' emulation.

Let us assume the following mnemonics for the three microcode related reconfigurable instructions: *'**pset** rm_addr'*, *'**cset** rm_addr'* and *'**exec** rm_addr'*. To implement the proposed mechanism, we only need to initialize LR with a proper value, i.e. the address of the reconfigurable instruction. This should be done by the compiler with the *'branch and link'* (**bl**) instruction of PowerPC. The assembly code of an application program containing the *'complete set'* instruction should be similar to the following:

	bl	*label1*	→	***bl***	— branch to *label1*	; LR = *label1*
label1:	**cset**	*rm_addr*	→	***blr***	— branch to *label1*	; LR = *label1*

To the right of the arrows, the actual instructions driven by the arbiter to the PowerPC instruction bus are denoted, LR values are also presented. Obviously, the processor will execute branch instruction to the same address, because LR remains unchanged and points to an address containing **blr** instruction. Thus we drive the processor into an eternal loop. It is the responsibility of the arbiter to get the processor out of this state. When the reconfigurable instruction

is complete, an *'end_op'* signal is generated by the $\rho\mu$-unit to the arbiter, which initiates the execution of **blrl** exactly twice. Thus, the effective address of the next instruction is loaded into the LR, which points to the address of the instruction immediately following the reconfigurable one and the processor exits the eternal loop. Below, the instructions generated by the arbiter to finalize a reconfigurable operation are displayed (instruction alignment is at 4 bytes):

label1:	**cset**	*rm_addr*	→	***blr***	— branch to *label1*
			→	***blr***	— branch to *label1*
				.....	
			→	***blrl***	— branch to *label1*
			→	***blrl***	— branch to *label1+4*
label1+4:	**next**	**instr**	→	.....	— next instruction

The proposed implementation allows also the execution of *blocks of reconfigurable instructions* (BRI) defined below:

Definition 5.1 *We define **BRI** as any sequence of reconfigurable instructions starting with the instruction '**bl**' and containing arbitrary number of consecutive reconfigurable instructions. No other instructions can be utilized within a BRI.*

Utilizing BRI saves the necessity to initialize LR every time a reconfigurable instruction is invoked, thus saving a couple of **bl** instructions. In this scheme only one **bl** instruction is used to initialize LR in the beginning of the BRI. The time spent for executing a single reconfigurable operation (T_ρ) is estimated to be the time for the *reconfigurable execution* ($T_{\rho E}$), consumed by the $\rho\mu$-unit, plus the time for three *unconditional taken branch instructions* (T_{UTB}) :

$$T_\rho = 3 \times T_{UTB} + T_{\rho E} \tag{5.1}$$

Assuming the number of reconfigurable instructions in the BRI to be N_{BRI}, the execution time of a reconfigurable instruction within a BRI costs:

$$T_\rho = 2 \times T_{UTB} + T_{\rho E} + \frac{T_{UTB}}{N_{BRI}} \tag{5.2}$$

In other words, *the time penalty for single reconfigurable instruction execution is* $3 \times T_{UTB}$ *and within a BRI execution - between* $2 \times T_{UTB}$ *and* $3 \times T_{UTB}$.

Optionally, the *'instruction synchronization'* instruction (**isync**) can be added before a BRI to avoid out-of-order executions of previous instructions during reconfigurable operation. This choice, however, depends on the particular program, being implemented.

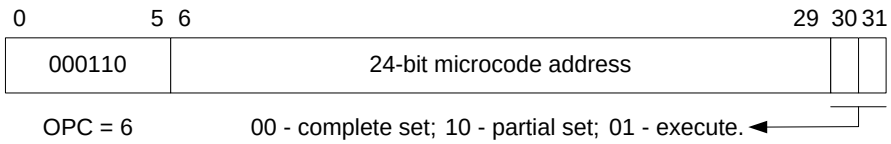


Figure 5.2: Reconfigurable instruction encoding: ρ-form.

Instruction encoding: To perform the Molen processor reconfigurations, the PowerPC Instruction Set Architecture (ISA) should be extended. Here, we will consider three microcode related instructions from the πISA, namely: **execute**, p-**set** and c-**set**. To encode these three instructions with respect to their efficient implementation and utilization, we have considered the following:

- The encoding scheme should be consistent with the PowerPC instruction format with opcodes (OPC) encoded in the six most-significant bits of the instruction word (see Figure 5.2).
- All three instructions have the same OPC field and same instruction form, which is similar to the I-form. Let us call the new form of the reconfigurable instructions *ρ-form*.
- The OPCodes of the instructions are as close to the OPC of the emulation instructions as possible (shortest Hamming distance), i.e. **blr** and **blrl**. From the free opcodes of the PowerPC architecture, such is opcode '6' ("000110_b").
- Instruction modifiers are implemented in the two least-significant fields of the instruction word, to distinguish the three reconfigurable instructions.
- A 24-bit address, embedded into the instruction word, specifies the location of the microcode in memory. A modifier bit R/P (Resident/Pageable), assumed to be a part of the address field, specifies where the microcode is located and how to interpret the address field. If R/P=1 a memory address is specified, otherwise an address of the on-chip storage in the $\rho\mu$-code unit is referred. The address always points to the location of the first microcode instruction. This first address should contain the microcode length or its final address. A microprogram is terminated by an *end_op* microinstruction.

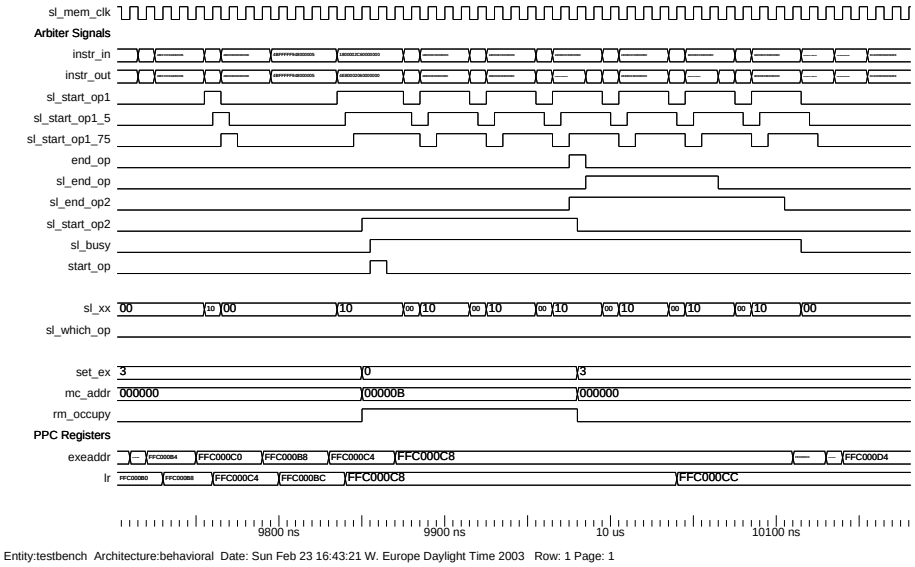


Figure 5.3: Reconfigurable instruction execution timing.

Hardware requirements: To implement the arbitration of the PowerPC instruction bus, the following design considerations have to be taken into account:

- The only input information, related to instruction decoding, arbitration and timing is obtained through the instruction bus.
- PowerPC instruction bus is 64-bit wide and instructions are fetched in couples (doublewords).
- Speculative (dummy) prefetches are performed, which should not disturb the right timing of a reconfigurable instruction execution.
- Both the arbiter and the $\rho\mu$-code unit strobe input signals on rising clock edges and generate output controls on falling clock edges.

The $\rho\mu$-code arbiter for PowerPC has been described in synthesizable VHDL and mapped on the Virtex II Pro FPGA. Figure 5.3 displays the timing of this implementation. The unit uses the same clock signal (*'sl_mem_clk'*) as the utilized instruction memory. The only inputs of the arbiter unit are the *input instruction* bus (*'instr_in'*) and *end of (reconfigurable) operation* (*'end_op'*).

The decode unit of the arbiter (see Figure 5.1) decodes the OPCodes of both fetched instructions simultaneously. Non-reconfigurable instructions are redirected (via the MUX) to output *'instr_out'*, directly driving the instruction bus of PowerPC. Alternatively, when either of the decoded two instructions is reconfigurable, the instruction code of **blr** is multiplexed via *'instr_out'* from the *'emulation instructions'* block. Obviously, the critical path penalty to the original instruction flow is just one 2-1 multiplexer and the decoding logic for a 6-bit value. In any case, this is the equivalent delay of a single logical 2-2 AND-OR gate or two 4-input FPGA Look-Up-Tables (LUT), thus negligible. The decode block generates two internal signals to the control block - *sl_start_op1* (explained later) and *sl_xx*. The latter signal indicates the alignment of the fetched instructions with respect to the reconfigurable ones. A one represents a reconfigurable instruction, a zero - any other instruction. For example, assuming big endian alignment: *"sl_xx=10"* means a reconfigurable instruction at the least-significant and a non-reconfigurable instruction at the most-significant address.

The control block generates signal *start (reconfigurable) operation* (*'start_op'*) for one clock cycle delayed with two cycles after the moment a reconfigurable operation is prefetched and decoded, thus filtering short (dummy) prefetches. In Figure 5.3 the rising edge of the internal signal *sl_start_op1* indicates the moment a reconfigurable operation is decoded. One can see that signal (*'start_op'*) is generated only when the reconfigurable instruction is really fetched, i.e. when *sl_start_op1* takes longer than one clock cycle. Dummy prefetch filtration has been implemented by two flip-flops, connected in series and clocked by complementary clock edges. The outputs of these flip-flops are denoted by signals *sl_start_op1_5* and *sl_start_op1_75*. The output control to the $\rho\mu$-unit, *sl_start_op* is generated between two falling clock edges.

Synchronously with the decoding of a reconfigurable instruction, the two instruction modifier fields (output signal *set_ex*) and *microcode address* (24-bit output *mc_addr*) are registered on rising clock edge (recall Figure 5.2). The internal flip-flop *sl_which_op* is used only when both of the fetched instructions are reconfigurable (*sl_xx="11"*) to ensure the proper timely distribution of *set_ex, mc_addr* and controls. In addition, two internal signals (flip-flops) are set when reconfigurable instruction is decoded. These two signals denote that the $\rho\mu$-unit is performing an operation (*sl_start_op2*) and that the arbiter is busy (*sl_busy*) with such an operation, therefore another reconfigurable execution can not be executed. To multiplex the data memory ports to the $\rho\mu$-unit during reconfigurable operations, signal *sl_start_op2* is driven out of the arbiter, via external signal *rm_occupy* to the data memory controller.

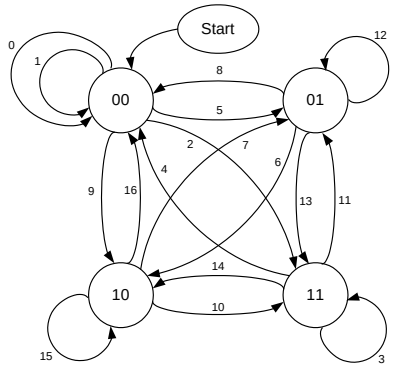
When a reconfigurable instruction is over, *'end_op'* is generated by the $\rho\mu$-unit and strobed on rising clock edge. At this moment, the *sl_start_op2* flip-flop is reset, thus releasing the data memory (via *rm_occupy*) for access by other units. Now, the control logic should guarantee that the **blrl** instruction is decoded exactly twice. This is done by a counter issuing active *sl_end_op* for precisely two **blrl** cycles, i.e., eight clocks. Instruction codes of **blr** and **blrl** differ only in one bit position. Therefore, redirecting *sl_end_op* via the MUX to this exact position of *'instr_out'* while **blr** is issued, drives **blrl** to the PowerPC. When *'end_op'* is strobed by the arbiter, another counter generates the *sl_end_op2* signal to prevent other reconfigurable operations from starting executions before the current reconfigurable operation has finished properly. The falling edge of signal *sl_end_op2* synchronously resets signal *busy*, thus enabling the execution of reconfigurable operations coming next.

5.1.3 Arbiter testing and hardware complexity

The reliability of the arbiter operation is important, therefore we propose a test program algorithm to validate its functionality for all possible instruction sequence scenarios.

Testing: To test the operation of the arbiter, we need an assembly program, strictly aligned into memory and testing all possible sequences of instruction couple (doubleword) alignments. Figure 5.4(a) depicts the transition graph of such a test sequence. A bubble in this graph represents an instruction couple alignment with respect to the reconfigurable instructions: one represents a reconfigurable instruction, a zero - any other instruction. Arrows indicate fetching of the next aligned instruction couple (we refer to these fetches as transitions). The minimal number of such transitions to cover all possible situations is 16 and the numbers next to each of the arrows indicate the position of the transition in the sequence of the test program. An extra 00 to 00 transition (transition 0) is performed to test the dummy prefetch filtration, the arbiter should be able to perform. Its corresponding assembly code is presented in Figure 5.4(b) and is in the beginning of the test program. The waveforms of the whole test program execution can be observed in Figure 5.5.

FPGA synthesis results: The VHDL code of the Arbiter has been simulated with Modeltech's ModelSim and synthesized with Project Navigator ISE 5.2 SP3 of Xilnx. The target FPGA chip was XC2VP20-5 (speed grade 5). Hardware costs reported by the synthesis tools are presented in Table 5.1. These results strongly suggest that at trivial hardware costs the $\rho\mu$-arbiter design can arbitrate the PowerPC instruction bus without causing severe critical path



(a) Transition graph;

```
                ...
                bl      labelA
labelB:         bl      labelC
                nop

//dummy prefetch here
labelA:         bl      labelB
labelC:         bl      label1

label1:         cset    rm_addr
                .......
```

(b) Dummy prefetch test.

Figure 5.4: Test program.

penalties and frequency decreases. Moreover, virtually all reconfigurable resources of the FPGA are available for designing reconfigurable microcoded custom computing units. Regarding the reported total number of flip-flops in the arbiter design (69), the majority of them (52) are used for registering *mc_addr* and *set_ex* outputs. Thus only 17 flip-flops are spent for the control

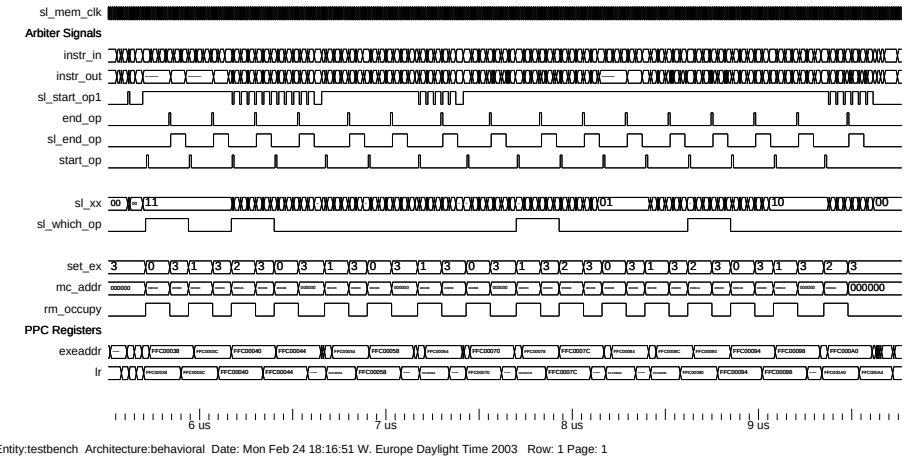


Figure 5.5: Test program waveforms.

block, including the two embedded counters (2×4 flip-flops). We estimate that the number of the control flip-flops may be further reduced and the frequency enhanced if required.

Table 5.1: Arbiter synthesis results for xc2vp20, speed grade-5.

Number of Slices	84 out of 10304	$< 1\%$
Number of Slice Flip Flops	69 out of 20608	$< 1\%$
Number of 4 input LUTs	150 out of 20608	$< 1\%$
Minimum clock period	7.004ns	
Maximum Frequency	142.776MHz	

5.2 The $\rho\mu$-code unit

In this section, we describe the $\rho\mu$-code unit design. In Subsection 5.2.1 we first discuss some problems regarding $\rho\mu$-code loading and propose their solutions by introducing $\rho\mu$-code manipulations. Subsection 5.2.2 adds details about the actual implementation of the $\rho\mu$-code unit.

5.2.1 Manipulations on the $\rho\mu$-code

In the original Molen architectural description [39], the end of the $\rho\mu$-code is marked by an end_op microinstruction. Conceptually, this is correct, however it creates implementation drawbacks with respect to whether the $\rho\mu$-code, stored into memory, is a sequence of microinstructions ($execute$ instruction) or a configuration bitstream (set instruction). The following discussion enlightens the related problems and proposes possible solutions.

$\rho\mu$-code termination: Considering the $execute$ $\rho\mu$-code, a single end_op microinstruction at the end of the $\rho\mu$-code segment suffices for the proper termination of the reconfigurable operation, provided the $\rho\mu$-code is properly aligned into memory. This technique, however, would not work with set $\rho\mu$-code, because the reconfiguration bitstreams are arbitrary bit sequences. Moreover, a pre-defined and widely accepted standard for such bitstreams does not exist, i.e., the same high-level hardware description file will result in completely different configuration bitstreams, which will vary per vendor and per device. Therefore, it is impossible to find a unique bit pattern not presented in the reconfiguration bitstreams, and use it as an end_op microinstruction.

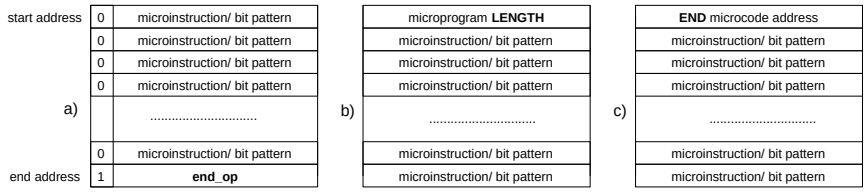


Figure 5.6: Microcode termination techniques.

On the other hand, if the end_op bit pattern is not unique for the entire $\rho\mu$-code segment, a $\rho\mu$-code loading may be terminated earlier by a fault end_op microcode and the operation of the system will be compromised. Obviously, additional techniques should be developed for proper $\rho\mu$-code termination.

Figure 5.6 depicts three possible solutions that can be utilized to solve the $\rho\mu$-code termination problem. In Figure 5.6a), a flag bit is utilized to indicate whether the memory word is an end_op (1), or any other microinstruction/bit pattern (0). This approach has been proposed in [85] and is applicable for both the *set* and the *execute* $\rho\mu$-codes. Its severe drawbacks are the redundant memory space utilization and the severity of the microprogram (resp. reconfigurable bitstream) alignment into the main memory. An alternative approach is illustrated with the examples in Figure 5.6 b) and c). In both cases, an additional microcode word is aligned at the starting address of the microprogram segment. The microcode word may contain either the length of the microprogram (Figure 5.6 b) or its final address (Figure 5.6 c). This approach is more efficient in terms of memory space because just a single extra microinstruction word is required. The latter two examples are functionally equivalent to each other and differ only in their underlying hardware implementations.

$\rho\mu$-code finalization: In all three cases of $\rho\mu$-code termination, explicit binary information should be added to the $\rho\mu$-code. For instance, in the case of end_op attached at the end of the set $\rho\mu$-code, additional flag bits should be inserted into the 'raw' bit pattern (see Figure 5.6a)) and the expanded $\rho\mu$-code bit patterns should be properly aligned into the targeted memory organization. Therefore, a process transforming a 'raw' configuration bitstream into a properly memory aligned $\rho\mu$-code is required.

Definition 5.2 *The process of preparing the $\rho\mu$-code for its final alignment into the targeted main memory of a Molen processor is called $\rho\mu$-code finalization.*

The position of the (automated) $\rho\mu$-code finalization tool in a generalized Molen CCU design process is depicted in Figure 5.7. A CCU algorithm is

described in a hardware description language (HDL) and targeted to a particular FPGA technology via logic synthesis, place and route tools. The bitstream generation tool, typically supplied by the FPGA vendor, generates the complete configuration bitstream file. This file is ready to be loaded into the targeted FPGA via any of the configuration paths supported (e.g., JTAG or dedicated configuration controllers). The Molen processor paradigm, however, requires a CCU configuration bitstream, i.e., a set $\rho\mu$-code, to be stored in the system main memory similarly to the software modules (see Chapter 2). A CCU is configured (i.e., configuration bitstream is loaded into the FPGA) via the $\rho\mu$-code unit, which should be capable of identifying the end of the set $\rho\mu$-code as discussed earlier. Therefore, the 'raw' configuration bitstream should be 'wrapped' according to the selected $\rho\mu$-code termination technique. This is performed by the $\rho\mu$-code finalization tool. This tool utilizes additional information for the specific Molen configuration, stored in configuration files ($conf$ in Figure 5.7). Such configuration files, for instance, should contain information about the targeted system memory organization, so that the set $\rho\mu$-code be aligned accordingly. The set $\rho\mu$-code endianess is transparent for the proposed process and does not require special consideration.

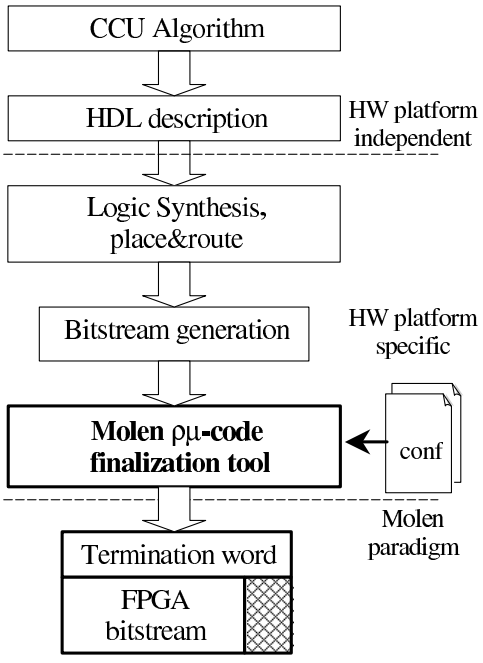


Figure 5.7: Molen finalization.

The discussed $\rho\mu$-code finalization process can be fully automated by the Molen $\rho\mu$-code finalization tool. The product of this tool is a binary sequence, compliant with the Molen programming paradigm. The finalized binary sequence can be a file, a linkable object, or a high-level data structure incorporating the binary information (e.g., included directly in a C project before compilation).

5.2.2 $\rho\mu$-code unit implementation

The proposed design of an $\rho\mu$-code unit utilizes the $\rho\mu$-code termination mechanism from Figure 5.6c), i.e., assuming the end microprogram address value is stored at the starting microcode location. A general view of the design is depicted in Figure 5.8.

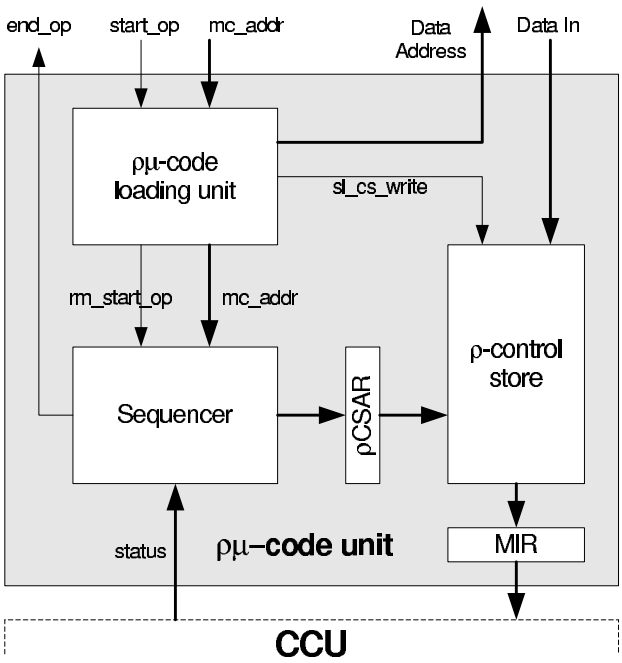


Figure 5.8: General view of the $\rho\mu$-code unit.

Operation: The $\rho\mu$-code loading unit, as its name suggests, loads microprograms from the external memory. It also initiates the operation of the sequencer, once the desired microprogram is transferred to or has been available in the ρ-control store. The $\rho\mu$-code unit operates as follows:

1. When a πISA instruction is decoded, the arbiter generates signal *start_op*, which initiates a reconfigurable operation.

2. The $\rho\mu$-code loading unit sequentially generates the addresses of the microprogram (through the 'Data Address' bus in Figure 5.8) to the main memory starting with address *mc_addr*. During the address generation, the ρ-control store is write enabled by signal *sl_cs_write*. Thus, the desired microprogram is stored into the ρ-control store via the write-only port *Data_In*.

3. Once the desired microprogram is available in the ρ-control store, i.e., the end address of the microprogram in the external memory has been reached, signal *rm_start_op* activates the sequencer.

4. The sequencer starts to generate $\rho\mu$-code addresses towards the ρCSAR (reconfigurable Control Store Address Register). The microinstruction to be executed by the CCU is loaded into the microinstruction register (MIR).

5. *Status* signals from the CCU are directed to the sequencer to determine the next microcode address. Once the status signals indicate that the CCU has completed the operation, the sequencer generates signal *end_op* to the arbiter.

The *end_op* signal indicates that the reconfigurable operation has completed and the arbiter initiates the execution of the next instruction from the application program. The next instruction can be either from the standard ISA of the core processor, or an instruction from the implemented πISA. If the latter is the case, the operation flow described above is repeated.

FPGA mapping: The Virtex II Pro FPGA has been used as a target reconfigurable technology. For the particular prototype, we assumed a microcode word length of 64 bits and a logical main memory segment of 4Mx64-bits (22-bit address) for microprograms. The ρ-control store has been designed to handle up-to 8KBytes in 64-bit microcode words. The primary microcode storage units of the ρ-control store have been implemented into the BRAM memory blocks of the FPGA fabrics. The ρ-control store BRAMs are configured as a monolithic dual port memory. Each of the two ports is unidirectional - one read-only and one write only. The read-only port is used to feed the MIR, while the write-only one loads microcodes from the external memory into the pageable section of the ρ-control store. The VHDL code of the $\rho\mu-$code unit has been synthesized with Project Navigator ISE 5.2 SP3 of Xilinx. The target

Table 5.2: $\rho\mu$-code unit synthesis results for xc2vp20, speed grade-5.

Number of Slices	71 out of 10 304	1%
Number of Slice Flip Flops	78 out of 20 608	$< 1\%$
Number of 4 input LUTs	171 out of 20 608	1%
Number of BRAMs:	4 out of 112	3%
Maximum Frequency [MHz]	≈ 130	

FPGA chip was xc2vp20-5 (speed grade 5). Reconfigurable hardware utilization, as reported by the synthesis tools, is presented in Table 5.2.

5.3 XREGs, memory organization, and clocks

The exchange registers (XREGs): The XREGs provide the interface for the communication between the GPP and the reconfigurable processor (RP). From the programmer's (GPP) point of view, the XREGs are considered as variables, used to communicate with the functions mapped on the CCUs. These variables correspond to physically implemented registers, accessible both by the GPP and the CCUs of the RP. It is not advised to exchange large pieces of data via the XREGs, because this would decrease the performance of the machine dramatically. Typical exchanged values should be small pieces of data. By *small* pieces of data we mean data comprising a single register up to a few registers, e.g., function parameters and results, memory addresses, CCU configuration parameters, etc. Since parameters and results are passed to or written from the XREGs both by the GPP and the RP, both co-processors should support the same register allocation mechanism. An example of such mechanism is described below and has been implemented by our prototype.

XREG allocation: The XREG allocation mechanism, proposed hereafter, is not mandatory for the implemented XREG organization of our prototype. It is just one possible implementation, used in the experiments and by the CCUs considered in this thesis. We note that other allocation mechanisms can be implemented without changing the XREGs design of our prototype. Figure 5.9 illustrates the proposed register allocation for two CCUs. In the proposed mechanism, the first register of the XREG register file (XR0) is reserved for an offset address within the register file. Each of the CCUs uses this offset address to allocate its own registers. The allocated registers of the parameters and/or results of a CCU will be referred to as *XREG blocks*. Though not mandatory, it is advisable an XREG block to contain sequentially allocated

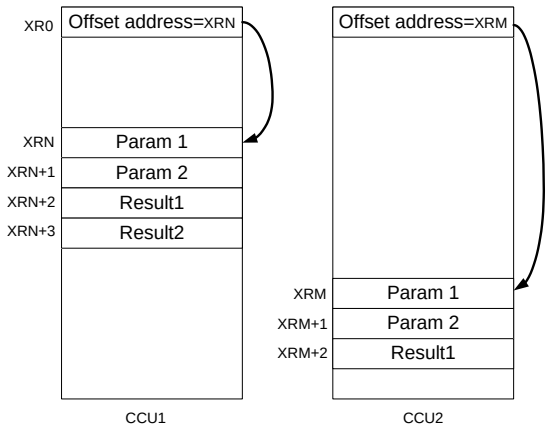


Figure 5.9: An example of XREGs allocation for two CCUs.

registers. There is no specific limit on the size of an XREG block other than the size of the entire implemented XREG file. In the example of Figure 5.9, an XREG block of four registers with offset XRN is allocated within the XREG file for CCU1. For CCU2, XR0 contains the value XRM, which allocates a XREG block of three registers at address XRM. Generally, both XREG blocks should not be overlapping, unless common registers are used by both CCUs, e.g., for communications. Though feasible, the assumption of explicit communication between two or more CCUs via common XREGs will not be considered further in this thesis. The offset value may be loaded in XR0 every time a CCU functionality is called by the software, but may be loaded once, as well. For example, every time before an **execute** or a **set** instruction is executed, an XREG offset can be loaded in XR0. Or, alternatively, only the first time a **c-set** or **p-set** instruction is executed (i.e., upon initialization), a preceding offset value loading in XR0 can be required. The choice of the offset loading moment depends on the CCU implementation, as well. The following example of a program piece illustrates in pseudocode the communication between the software part of an algorithm and a CCU:

movtx XR0, OFFSET // Load the offset address in XR0
for (All Param_i)
{**movtx** [OFFSET+i], Param_i}; // Load all parameters of the CCU
synchronize; // Synchronize parameter loading
set/execute ADDRESS; // A **set** or an **execute** instruction
for (All Result_j)
{**movfx** Result_j, [OFFSET+i_{max}+j]}; // Read the results from the CCU

The programmer should be aware of the XREGs structure of each particular CCU. For automatic allocation of XREG blocks, the CCU XREGs information can be provided by CCU specification files. It is a programmer's responsibility to allocate the XREG blocks correctly, so that no register allocation conflicts between the CCUs appear.

XREGs prototype implementation: The XREGs have been implemented as a dual-port register file. One port is connected to the GPP, the other - to the RP, more precisely to the CCUs. Both of the XREG file ports are bidirectional (i.e., read-write ports) and with separate and independent address busses. In the Virtex II Pro FPGA, the GPP is a PowerPC core. We decided to utilize the Device Control Registers (DCR) interface of PowerPC for the GPP interface of the XREG file. The DCR ports are easily connected to the GPP-side ports of the XREG. Moreover, the DCR transfers are supported by two dedicated PowerPC instructions: **mtdcr** and **mfdcr**. Thus, the πISA instructions **movtx** and **movfx** have been mapped to **mtdcr** and **mfdcr**, respectively. The RP port of the XREG file has been connected to the CCUs via an interface, described in Section 5.4. In the particular prototype, a single BRAM block (2KBytes) organized as 512×32-bit storage has been utilized for the XREG file.

Memory organization: For transferring large amounts of data, e.g., image data arrays, the XREG mechanism is not efficient. A huge performance penalty has to be paid if every piece of data is moved from the PowerPC allocated memory, via the XREGs, to the CCU allocated memory. To avoid such performance draw-backs, a shared memory is implemented. Thus, the core processor and the CCUs process the same pieces of data without having to transfer them from one location to another. The arbiter determines the access to the memory by the dedicated signal 'occupy memory'(see Figure 5.1) based on the performed instruction. In case the instruction is from the standard ISA, the memory control is directed to the PowerPC, if a πISA instruction is executed, the memory control is transferred to the RP, more precisely - to the active CCU.

For the memory design of the prototype, we considered the on-chip memory blocks of the utilized FPGA. The available BRAM blocks in xc2vp20 allow the implementation of 128 KBytes memory for both data and instructions. The PowerPC has a Harvard architecture with separated instruction and data addressing spaces. Therefore, for better performance, we separated the main memory into two equal segments - 64 KBytes for instructions and other 64 KBytes for application data. Both the instruction and the data memory are organized with 64-bit word memories. In this case, we note that the amount of

memory is limited only by the available on-chip memory. By utilizing external memories, it is possible to extend the memory volume up to the entire memory space addressable by PowerPC (i.e., 32-bit addresses). The later option, however, has not been considered in the described prototype.

Regarding the proposed two-dimensionally addressable memory, discussed with details in Chapter 4, its usage is application specific, therefore it is considered optional. Such a memory is not part of the common ”back-bone” memory architecture. It is implemented as a part of the RP and controlled by microcode, i.e., this memory is considered as a part of the reconfigurable microarchitecture. A particular CCU is responsible to access the two-dimensional data correctly, thus making the two-dimensional memory transparent for the Molen programmer.

Clock domains: Due to the polymorphic nature of the Molen processor and for performance efficiency, three clock domains have been implemented in our prototype:

- **PPC_clk**- clock signal to the core processor. The frequency of this signal has been set to 250 MHz, the maximum recommended for the PowerPCs in xc2vp20-5;
- **mem_clk**- clock signal to the main memory. This signal has been set to be three times lower than the PPC_clk, i.e., 83 MHz;
- **CCU_clk**- clock signal to the CCU driven by an external pin. It may be utilized by any CCU, which requires frequencies, different from the PPC_clk and mem_clk.

5.4 The polymorphic interface

An important advantage of the Molen paradigm is that a new application specific functionality can be embedded without changing the architecture of the processor, nor its organization. Ideally, no architecture- or organization-specific requirements should be imposed to the CCU design. It is desirable almost any third-party design of an application-specific accelerator to be implemented as a CCU easily. Such an accelerator may be designed either by hand or generated automatically. Therefore, the interface between the ’back-bone’ Molen organization and its potential CCUs should be as unrestrictive as possible. At architectural level, the interface between the core processor and the CCUs is determined by the πISA, the exchange registers, and the

shared data memory space. Consequently, the underlying hardware organization should support the interface, defined by the architecture. Figure 5.10 depicts the hardware interface signals between a single CCU and the rest of the Molen organization. In accordance with the architectural definition, each

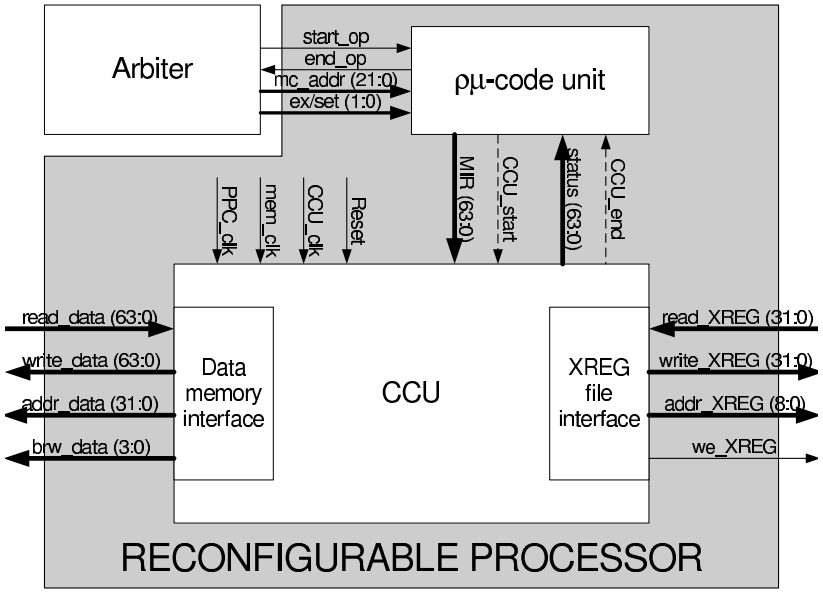


Figure 5.10: The CCU polymorphic interface.

CCU can interface with three parts of the Molen organization: the $\rho\mu$-code unit, the XREGs, and the data memory.

Microprogrammable and self-controlled CCUs: The interface with the $\rho\mu$-code unit is based on the microprogrammable control paradigm. A microinstruction is directed to the CCU via the bus MIR, and status signals are generated to the sequencer of the $\rho\mu$-code unit. The proposed control interface is general and supports not only microprogrammable CCUs, but also CCUs with embedded hardwired control units. For such self-controlled CCUs the two synchronizing signals *CCU_start* and *CCU_end* are explicitly depicted in Figure 5.10. The *CCU_start* signal initiates the operation of the CCU and the *CCU_end* indicates that the CCU has completed its task. In order a hardwired control CCU to be embedded in the Molen processor, only the requirement to support such a start-stop synchronization should be met. It is sufficient that a CCU has an input starting signal, which initiates its operation and a flag output signal indicating when an operation has completed. In essence, the *CCU_start*

signal is a part of the MIR bus and the *CCU_end* signal- a part of the status bus. Furthermore, any wire of the MIR bus can be utilized as a *CCU_start* signal to an individual CCU and identically, any wire of the status bus can be utilized as a *CCU_end* signal. Thus, allowing start-stop operations of the CCUs we leave the interface open not only for microprogrammable units, but also for self-controlled accelerators.

Data memory and XREG file interfaces: To allow data communications between the core processor and a CCU, two separate interfaces are provided. For exchanging parameters and results, which do not exceed a few register-wide words, the XREG interface can be utilized. We mentioned that in the Virtex II Pro prototype a single BRAM block has been used to implement the XREG file. Therefore, the implemented CCU interface to the XREGs assumes a BRAM organized as 512×32-bit memory, i.e. 32-bit data busses and a 9-bit address bus (See Figure 5.10). Large amounts of data are not exchanged via the XREGs. Instead, a shared memory is implemented allowing the PowerPC and the CCUs to allocate and process data in the same (shared) memory locations. Regarding the data memory interface of the CCUs, we considered the same interface format as it is for the PowerPC GPP, i.e., 64-bit data busses and 32-bit addresses. In addition, a byte-enabled writing into the data memory can be implemented in a CCU utilizing the bus *brw_data*.

Additional synchronizing signals: The three clock signals of the prototype design described earlier, *PPC_clk*, *mem_clk*, and *CCU_clk*, are available for any CCU design, as well as a *reset* signal is presented.

5.5 Overall synthesis results

As a platform FPGA, we used a Xilinx xc2vp20-5 device from the Virtex II Pro™ family. The Molen organization has been described in VHDL and synthesized by the Xilinx XST tool of ISE 5.2, SP3. In this section, we consider only the reconfigurable hardware overhead caused by the Molen-specific "backbone" infrastructure described above, i.e., the arbiter, the $\rho\mu$-code unit and the associated infrastructure. No CCU implementations are considered in this hardware evaluation as they are application specific and vary per configuration. Data memory has not been considered in the hardware cost estimation, as well, because arbitrary memory volumes can be implemented, without influencing the overall utilization of the other hardware categories. A top level view of the implemented prototype is depicted in Figure 5.11. The arbiter, the $\rho\mu$-code unit (rm_unit), the XREG file (xreg_file) and the clocks generator are

Table 5.3: Molen organization synthesis results (* RP infrastructure only, without any CCU implemented).

Device xc2vp20 Speed Grade -5	RP*	Arbiter	Total incl. XREGs	Available Resources	%
Number of Slices	71	84	156	10304	1
Number of Slice Flip Flops	78	69	147	20608	1
Number of 4 input LUTs	171	150	322	20608	1
Number of BRAMs:	4	N.A.	5	112	4
Maximum Frequency [MHz]	130	143	130	N.A.	N.A.

presented. The molen_core block wraps the PowerPC processor and the main memory. We recall that a microcode word length of 64 bits and a ρ-control store of 8KBytes have been considered. The XREGs have been implemented in a single 2KBytes BRAM organized as 512×32-bit storage.

Synthesis results: Hardware costs reported by the synthesis tools are presented in Table 5.3. The first column displays the particular FPGA resources considered. Column two reports the actual values of these resources, consumed by the reconfigurable processor, without considering any CCU implementation. This includes the $\rho\mu$-code loading unit, the sequencer and the ρ-control store. Column three presents resource utilization of the arbiter. In column four, the overall resource consumption of the reconfigurable processor infrastructure, the arbiter and the XREGs is presented. Finally, columns five and six respectively present the available FPGA resources in the xc2vp20 chip and the utilized part of these resources by the Molen organization (in %). Synthesis results strongly suggest that the Molen infrastructure consumes trivial hardware resources, thus leaving virtually all FPGA resources available for CCU implementations.

5.6 Program code annotation

This section can be considered as a guideline for the implementation of a backend PowerPC compiler supporting the Molen paradigm. In the following discussion, we explain by an example how a program code is annotated to support the new architectural extension. For our experiments with the Virtex II Pro prototype, we utilized the publicly available compiler GCC for PowerPC. Some considerations, specific for the particular implemented design are explained. The following example describes a code annotation for an **execute** operation

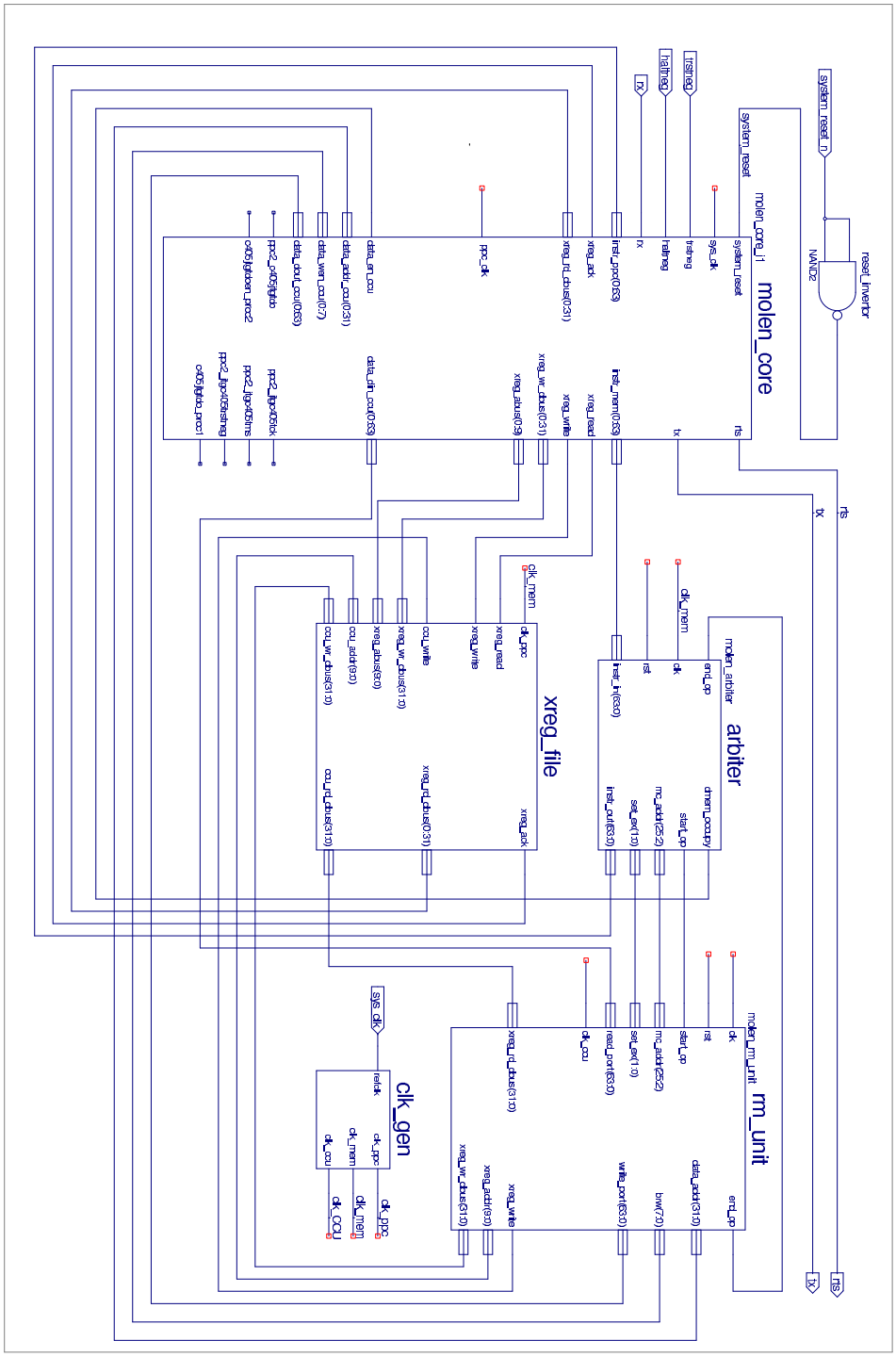


Figure 5.11: Top-level schematic of the Virtex II Pro Molen prototype.

with two input parameters and one result. The input parameters are the addresses of memory aligned data arrays with input and output data, the result is a scalar value.

Example:

```
/*Declarations*/
long long int inp_block[];
long long int outp_block[];
long long int result;
long unsigned XREGvalue;

/*Load a CCU XREG block offset of 0x56 into XR0*/
__asm("mtdcr 0x0, 0x56");
/*Load the first parameter (the input array address) in XR56*/
XREGvalue = (long unsigned) inp_block;
__asm("mtdcr 0x56, XREGvalue");
/*Load the second parameter (the output array address) in XR57*/
XREGvalue = (long unsigned) outp_block;
__asm("mtdcr 0x57, XREGvalue");

/*Start the execute operation*/
__asm("sync");                    /*Synchronize*/
__asm("nop");
__asm("nop");
__asm("nop");
__asm("bl label1");
__asm("label1: execute ρμc_addr");/*Here a BRI can be included, as well*/
__asm("nop");                     /*This line must not be a branch target*/
/*End the execute operation*/

/*Read the result from XR58*/
__asm("mfdcr XREGvalue, 0x58");
```

In the above example, a CCU XREG block offset of 0x56 is assumed and the value is loaded in the first register (XR0) of the XREG file. The address of the input array *inp_block* is loaded in the first position (XR56) of the CCU XREG block as the first parameter of the execute operation. The second parameter is loaded in the next position of the CCU XREG block (XR57). Once the parameters of the CCU function have been loaded into the corresponding XREG

block, the **execute** operation can be invoked. Note, that the same example can be considered for the **set** operations as well.

Before proceeding with the actual **execute** operation, the programmer has to make sure that all possible preceding out-of-order operations have completed. Therefore, the PowerPC *sync* instruction is included in the example code (also explained in Section 5.1). For example, out-of-order operations in the considered prototype are the memory accesses. We assume that between the *sync* instruction and the Molen instruction (**execute**) execution, all instructions, including the Molen one, execute in-order. The three *nop* instructions immediately after *sync* are required due to the prefetch pipeline of the PowerPC, which is 3 stages deep. Thus, we make sure that the **execute** operation is fetched and decoded by the arbiter after *sync* has completed. In essence, the three *nop* instructions can be replaced by some other in-order executing PowerPC instructions (i.e., no memory transfers), which may be considered by the compiler optimization process. The **execute** instruction itself is invoked, utilizing the branch-to-link-register mechanism, described in details in Section 5.1. At this place a block of reconfigurable instructions (recall BRI from Section 5.1) can be implemented, as well. In the example, after the **execute** instruction a *nop* is included. In essence, this can be any other instruction, with only one limitation: this instruction should not be a branch target. If the instruction, immediately after the **execute** (or after the last BRI instruction) is a branch target, the **execute** will be erroneously fetched and decoded by the arbiter, thus erroneously executed. This can happen if the instruction after the **execute** is fetched, being targeted by a branch. In such a case, due to the memory alignment, both 32-bit instructions can be fetched through the 64-bit instruction bus simultaneously.

5.7 Conclusions

In this chapter, we proved in practice that the Molen concept can be implemented by emulation of reconfigurable operations without changing the design of the core processor. Thus, we also prove the practical feasibility of the Molen paradigm in general. More specifically, we utilized the original PowerPC ISA to emulate the execution of the minimal functionally complete πISA on the Virtex II Pro platform FPGA. We proposed efficient designs of the potentially performance limiting parts of the $\rho\mu$-coded processor, namely: the arbiter, the $\rho\mu$-coded unit, the exchange registers, the memory organization, and the clock domains and distribution. The arbitration between πISA

and original PowerPC ISA instructions was investigated. All design aspects of an arbiter have been described, including software considerations, architectural solutions, implementation issues and functional testability. We also addressed several specific problems regarding $\rho\mu$-code related manipulations. More specifically, the generation, processing, memory alignment and loading of $\rho\mu$-codes were investigated and alternative design considerations were proposed with an emphasis on the set $\rho\mu$-code. A $\rho\mu$-code unit design was described and its prototype-specific synthesis data were presented. The considered organization of the exchange registers (XREG) file was introduced in details and a possible XREG allocation discipline was proposed. The implemented memory organization and the available clock domains were also described. Additionaly, it was emphasized on the importance of the CCU designers' interface for the practical implementation of new application-specific functionalities. In this direction, we defined an open, non-restrictive, CCU interface based on a general microcode control, and supporting general (exchange) register and memory accesses. The proposed interface can be utilized to embed various third-party designed CCUs in the Molen concept with minimal (if any) design changes of these CCUs. Synthesis results for the back-bone Molen infrastructure, i.e., all described units excluding data memory and any particular CCU, indicate a very low utilization of the reconfigurable resources of the selected Virtex II Pro XC2VP20 device. Utilization figures, as a part of all available resources on the chip, vary between 1% and 4% per particular resource category. Thus, virtually the entire FPGA area remains available for CCU implementations. And last, but not least, by means of an example of an annotated C-code segment, we explained the high-level software support for the Molen (application-specifically) augmented architecture. We also gave practical tips how to program for the proposed Molen prototype and explained why certain programming considerations have to be observed. As far as the GPP and the RP in the proposed prototype remain as closely coupled as in the original Molen proposal, we expect and in the remainder of this thesis prove its high performance benefits. Theoretical and experimental performance evaluations of the Virtex II Pro Molen prototype implementation with some embedded CCUs are presented and analyzed in the following Chapter 6.

Chapter 6

Performance Evaluation

In this chapter, we evaluate the performance of the proposed Virtex II Pro prototype experimentally. The evaluation methodology comprises three approaches, considered with respect to the requirements of the prototype and the application. These approaches are referred to as *real experimental evaluation*, *experimentally projected evaluation*, and *theoretical estimation*. The chapter establishes theoretical grounds to support the proposed methodology and to analyze the prototype performance for the considered applications, which are MJPEG, MPEG-2, and MPEG-4. A discussion on the manual versus automated custom computing unit (CCU) design for the Molen polymorphic processor outlines the boundaries of the implementability of both methods. Automatically and manually generated CCU designs are utilized throughout the evaluation. The obtained experimental results can be considered as an indication for the potentials of a general Molen processor for speeding up various computationally demanding applications.

The chapter is organized as follows. In Section 6.1, the evaluation methodology, its constituent approaches and some theoretical grounds are described. Section 6.2 discusses the manual vs. automated CCU design considerations and presents synthesis data on the CCUs, considered for the experiments but not discussed in Chapter 3. Experiments and experimental results are reported in Section 6.3, which is divided into three subsections, each of them dealing with the MJPEG, MPEG-2, and MPEG-4 applications. Finally, the chapter is concluded with Section 6.4

6.1 Performance evaluation methodology

For the experimental performance evaluation we have considered three evaluation approaches, which have been established with respect to two implementation limitations that apply. The first limitation is the availability of an established or a reliable *benchmark program*. The other limitation is the amount of the available *prototype program memory*, which is currently 64 KBytes. The three evaluation approaches are: *real experimental*, *projected*, and *theoretical estimation*.

A real experimental *evaluation is considered for reliable benchmark applications, which fit into the prototype memory*, i.e., when neither of the mentioned limitations applies. Such an application is the MJPEG software implementation, considered in our experiments. In the real experimental scenario, the original benchmark program is compiled and run on the prototype processor. The duration of the data processing for the entire program is measured in number of PowerPC clock cycles. In the following step, the benchmark program is annotated (as described in Section 5.6) to support application specific kernels and those kernels are loaded into the FPGA as CCUs. The annotated benchmark program is executed again on the prototype with the considered CCUs configured. PowerPC cycles for the entire data processing are measured again. The ratio between the execution cycle numbers before and after the program code annotation gives the actual speedup of the benchmark.

A projected *evaluation is required when the considered benchmark program code exceeds the amount of program memory implemented on the prototype.* With this approach, we project the entire application speedup assuming the implementability of a larger program memory. We measure locally the speedup for each of the interesting kernels using the real experimental approach on program segments that can fit into the prototype memory. Using these local speedups per kernel and a generalized formulation of the Amdahl's law [86], the overall speedup of the entire application is projected. We considered this approach for the popular Berkeley MPEG-2 encoder and decoder benchmarks.

A theoretical estimation *is employed to evaluate the potential speedups of unreliable or unestablished benchmarks, where only the general trends and functions of the application are defined.* Though any prediction on the overall speedup of such applications will be highly speculative, the speedups of some kernels can be precisely measured if the real experimental approach is applied locally. In our experiments, we consider the highest profiles of MPEG-4. This was indicated in Chapter 3 where we introduced some hardware accelerators

of high profile MPEG-4 kernels. These accelerators are considered in our evaluation of the MPEG-4 kernel speedups. Their impact on the overall MPEG-4 speedup can be theoretically estimated by means of the Amdahl's law. Since there are no reliable applications of the Core and Main MPEG-4 profiles that can be used for benchmarks, we can only speculate on how much such applications would be speeded up when they become available. Note that for these theoretical estimations real experiments are carried out when the local speedups of the considered MPEG-4 kernels are measured on the Molen prototype.

More details on each of the utilized performance evaluation approaches are presented along with the reported experimental results for each of the benchmark applications, considered further in this chapter.

Amdahl's law and reconfigurable computing: As we already indicated, the Amdahl's law can be utilized for the evaluation of the projected speedup of an application running on a Molen processor. In fact, this law can be used not only for evaluations with different degree of speculation, but it can be also used to evaluate the theoretical boundaries of the projected speedups, including the real experimental evaluation approach. Following, we do not utilize the original notations of the Amdahl's law [86], but introduce our own, instead. Our notations intuitively correspond to the specifics of this thesis and contribute to its readability and better understanding.

Assume a sequential software benchmark, where T is the execution time of the original program and T_{SEi} - the time to execute its kernel *i* in software. Further assume $T_{\rho i}$ is the execution time for a reconfigurable hardware implementation of the same kernel *i* (notations are consistent with Equations (5.1) and (5.2)). The overall speed-up of the benchmark program with respect to the reconfigurable hardware implementation of kernel *i* is:

$$S_i = \frac{T}{T - T_{SEi} + T_{\rho i}} = \frac{1}{(1 - a_i) + \frac{a_i}{s_i}} \tag{6.1}$$

Where a_i is the fraction taken by kernel i from the total time of the sequential software execution, and s_i is the (local) speedup of kernel i in reconfigurable execution scenario. That is, $a_i = \frac{T_{SEi}}{T}$, $0 < a_i \leq 1$ and $s_i = \frac{T_{SEi}}{T_{\rho i}}$, $s_i > 0$. Identically, assume that totally all considered kernels take a fraction of a from the application execution time, i.e., $a = \sum_i a_i$, $0 < a \leq 1$. Then, all considered kernels will speedup the benchmark by:

$$S = \frac{T}{T - \sum_i T_{SEi} + \sum_i T_{\rho i}} = \frac{1}{(1 - a) + \sum_i \frac{a_i}{s_i}} \tag{6.2}$$

Theoretical analysis: Consider Equation (6.2). We are interested to establish the theoretical boundaries of the execution speedup that can be achieved. Clearly, if we assume that a reconfigurable kernel implementation executes for time 0, the local speedup with respect to its software execution will approach its maximum, i.e., infinity:

$$s_i^{max} = \lim_{T_{\rho i} \to 0} \frac{T_{SEi}}{T_{\rho i}} = \infty \tag{6.3}$$

Assume that all considered kernels execute for time 0 in reconfigurable scenario. That is, $\forall i, s_i \to \infty$ (from Equation (6.3)) and considering Equation (6.2), we devise:

$$S_{max} = \lim_{\forall i, s_i \to \infty} S = \frac{1}{1-a} \tag{6.4}$$

Where S_{max} is the theoretical maximum of the achievable speed-up, given the fraction a, which corresponds to a certain kernel partitioning of the benchmark application. Obviously, the bigger a, the bigger the speedup. That is, the more sequential portions of the execution time of a program we accelerate in reconfigurable hardware, the faster the overall speedup. A careful analysis of Equation (6.4) suggests that an order of magnitude acceleration is attainable only if more than 90% (a=0.9) of the application are accelerated. Therefore, *speedups of sequential algorithms in orders of magnitude are impractical to achieve, unless virtually the entire application execution time is accelerated by dedicated hardware*. One more support for this statement is the fact that two orders of magnitude accelerations can be achieved only if 99% (a=0.99) of the execution time is reduced to 0. Obviously, for orders of magnitude speedups, a GPP is virtually unnecessary and the entire application should be implemented (if possible) in dedicated hardware. Generally, such a solution will increase the silicon cost of a design and will bring all disadvantages related to the hardwired designs, mainly losing flexibility and even implementability. On the other hand, in the general purpose computing society, even accelerations of 10% are considered spectacular. In this field, if we are capable to achieve accelerations between 50% and 1000% by implementing only a few selected kernels in (reconfigurable) hardware, we can safely claim a considerable speedup.

Some convenient graphical interpretations of the Amdahl's law are illustrated in Appendix A along with examples with practical parameter values. The proposed analysis will be used as a guideline and will be referenced when we evaluate our experimental results.

Experimental testbench: Our experiments have been carried out on an Alpha-Data development board ADM-XPL (for details check `http://www.alpha-data.com/`) equipped with the Xilinx Virtex II Pro chip xc2vp20-5. The FPGA chip is an engineering silicon with a speed grade 5 recommending PowerPC clock frequency of 250 MHz. In our experiments, we run the hardware at this frequency, but in the evaluations we tried to discard its absolute value. Wherever possible, our evaluations are made on relative basis and, instead of absolute time, PowerPC cycle numbers are considered. The Xilinx standard development tools embedded in the ISE 5.2. (Integrated Software Environment) SP3 have been utilized both for the synthesis and FPGA mapping. Behavioral hardware simulations have been performed with ModelSim SE 5.7c of Modeltech. Programs have been compiled with the public domain compiler GCC included in the EDK 3.2 (Embedded Development Kit) of Xilinx.

6.2 Reconfigurable units considered

In this section, we discuss the two conceptually different approaches to design a reconfigurable unit (CCU) supporting a specific kernel functionality, namely the *manual* and the *automated* hardware design generation. We also report the synthesis data for those units considered in our experiments, the internal organization details of which are beyond the scope of this dissertation.

Automatically generated vs. manual designs: During the discussion related to Figure A.2 in the Appendix 6.1, we argue that in order to obtain 90% of the top theoretical speedup (S_{max}), a kernel that consumes between 50% and 90% of the entire application execution time, must be accelerated locally between 10 and 80 times, respectively. This conclusion gives practical boundaries of the feasibility and the design requirements of a potential reconfigurable implementation. Additionally, experimental results, reported later in this chapter, prove that beyond certain point, growing local kernel accelerations do not contribute efficiently to the overall application speedups. The implication is that in many practical cases, severe kernel speedups of, say, several orders of magnitude, are not required for their supporting hardware accelerators. Thus, not necessarily the most efficient kernel accelerator is the most efficient reconfigurable implementation application-wide. This conclusion opens a design gap, which can be filled by automatically generated hardware.

Generally speaking, automatically generated designs are far from the optimal hardware as they are neither faster, nor more cost-effective in silicon area com-

pared to the manually designed units. In the reconfigurable devices, however, the customizable hardware gates are constantly increasing in number, which releases the requirement for cost-effective area designs. Moreover, as far as power consumption is not concerned, expanding the hardware within the available reconfigurable resources may not be considered as implementation prohibitive. On the other hand, our analysis suggests that a Molen machine organization can speed up application processing significantly, without implementing the most time efficient CCU designs.

Yet, in certain cases, where some severe design requirements have to be met, the manual design of CCUs is indispensable. Very often, the manual design can make the difference between implementability and non-implementability within a given set of requirements, e.g., limited availability of reconfigurable resources. Adding the low maturity of recent tools for automated hardware generation as well as the vast number of non-trivial and irregular hardware operations, often makes the manual approach the only option for a CCU design. In the experiments to follow, we employed both the automated and the manual approaches for the designs of the considered CCUs.

Synthesis data for the designs considered: For the experiments reported in Subsection 6.3.1, we utilized the Compaan [32, 87] and Laura [33] tools to generate automatically a CCU, which supports four computationally demanding operations of the MJPEG encoding algorithm altogether. The obtained synthesis results are reported together with the experiment description in Subsection 6.3.1. Experimental results presented in Subsection 6.3.2 regard the MPEG-2 application, where the Sum-of-Absolute-Differences (SAD) operation, the Discrete-Cosine Transform (DCT) and its inverse transform (IDCT) are considered for CCU implementations. Since these three operations have been extensively investigated in the literature, we are not focusing on their organizational details. Interested readers are referred to the sources, describing the devices we implemented for our experiments. Following, synthesis data for the considered designs are presented. Table 6.1 displays synthesis results considering the Xilinx xc2vp50 FPGA device. For the SAD function, we implemented the organization proposed in [88]. The super-pipelined 16-byte version of this SAD organization (SAD16) is capable of processing one 16-pixel line (1 pixel is 1 byte) of a macroblock in 17 cycles at over 300 MHz. The 128-byte version (SAD128) processes eight macroblock lines in 23 cycles, and the 256-byte version (SAD256), processes an entire 16x16-pixel macroblock in 25 cycles. SAD256 requires more resources than available in the xc2vp20 chip, therefore we consider it for an implementation on a larger FPGA (e.g., xc2vp50). To support the DCT and IDCT kernels, we synthe-

Table 6.1: Synthesis results per CCU implementation.

Device xc2vp20 Speed Grade -5	SAD16	SAD128	SAD256 (xc2vp50)	DCT	IDCT	Available Resources
Slices	831	6807	13613*	4314	5436	10304
Slice Flip Flops	1448	11862	23724*	7964	9876	20608
4 input LUTs	1390	11379	22757*	6832	8624	20608
BRAMs	N.A.	N.A.	N.A. *	2	2	112
F_{max} [MHz]	310	310	310*	96	96	N.A.

* Results for xc2vp50 FPGA

Table 6.2: Synthesis parameters for the Core Generator™ IPs.

Parameter	2-D DCT	2-D IDCT
Data width [bits]	16 (signed)	16 (signed)
Coeff. width [bits]	24	24
Result width [bits]	16 (rounded)	16 (rounded)
cycles/input sample	6	8
Internal latency [cyc]	94	97

sized the 2-D DCT and 2D-IDCT v.2.0 cores available as IPs in the Xilinx Core Generator Tool. The parameters for their synthesis are presented in Table 6.2. Considering the implemented clock domains and synthesis results (from Table 6.1) in our experiments, we have run the DCT and IDCT functions at mem_clk frequency (83MHz). The SAD designs were clocked by PPC_clk, i.e., at 250MHz.

Synthesis data on the hardware supporting the considered MPEG-4 kernels have been reported in Chapter 3. The speedups of these kernels are evaluated experimentally in Subsection 6.3.3.

6.3 Experimental results

In this section, we consider three benchmark applications to evaluate the performance gains attained by the Molen Virtex II Pro prototype. The considered three applications are MJPEG, MPEG-2, and MPEG-4. Each of these benchmarks is evaluated with one of the three evaluation approaches, proposed in Section 6.1 and results are reported in three separate subsections.

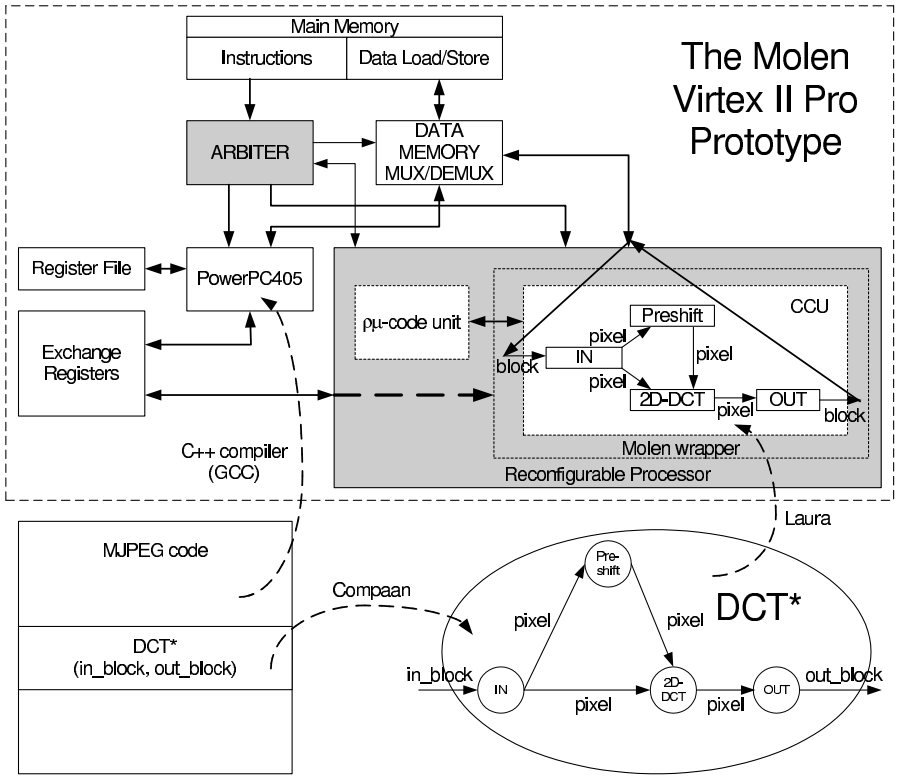


Figure 6.1: Mapping MJPEG onto the Virtex II Pro Molen prototype.

6.3.1 MJPEG real experimental evaluation

An MJPEG (Motion JPEG) encoder processes the frames in a video sequence as a series of JPEG images. We consider an MJPEG object-oriented source code written in C++. The compiled MJPEG application fits into the implemented memory of our Virtex II Pro Molen prototype, therefore we utilize the real experimental approach to evaluate the performance gains. An automated process is employed to synthesize a CCU, supporting four computationally demanding operations of the MJPEG encoding algorithm altogether.

Mapping MJPEG on the Molen Prototype: Figure 6.1 illustrates how the MJPEG encoder is mapped on the Molen Virtex II Pro prototype. The four operations, considered for automatic hardware generation are *block input*, *preshift*, *2D-DCT* and *block output*. These three operations are embedded in a single hardware design generated with the Compaan [32, 87] and the Laura [33]

Table 6.3: Synthesis results for the automatically generated DCT* CCU.

Device xc2vp20 Speed Grade -5	CCU	CCU+ wrapper	Available Resources
Slices	1804	1975	10304
Slice Flip Flops	2271	2388	20608
4 input LUTs	2014	2228	20608
BRAMs	4	4	112
Multipliers18x18	8	8	112
F_{max} [MHz]	100	100	N.A.

tool-sets, which utilize Khan Process Networks (KPN) [89] for intermediate modelling format. The output of the Laura tool-set is a synthesizable VHDL code. Since the Laura tool-set does not consider the Molen CCU interface (see Chapter 5), we manually designed a Molen consistent wrapping interface to embed the generated hardware unit as a CCU. The core functionality among the considered four is the 2D-DCT, therefore, we refer to all four operations as to a single kernel denoted by DCT*. The DCT* CCU is synthesized with the Xilinx tools and mapped on the Virtex II Pro FPGA. Synthesis results are reported in Table 6.3. Column 2 contains the resource utilization for the automatically generated unit without the Molen specific interface implemented. In Column 3, synthesis data for the Molen interface wrapper (see Figure 6.1) is also considered.

The Molen program code supporting the DCT* CCU (see Section 5.6), as well as the rest of the MJPEG C++ code (i.e., not including the DCT* kernel) are considered for PowerPC execution, more precisely they are mapped into the main memory. For the experiments, we considered an image size of 48×48 and 4:2:2 YUV macroblock format, i.e., a 16×16-pixel macroblock comprises four 8×8 Y (luminance) blocks and four 8×8 chrominance blocks (two U and two V blocks). The DCT* kernel processes "half" macroblocks at a time, i.e., two luminance and two (U and V) chrominance blocks.

Speedup evaluation: In the real experimental evaluation approach, the original benchmark program is compiled and run on the prototype processor first. The duration of the data processing for the entire program is measured in number of PowerPC clock cycles. Separately, the benchmark program is annotated (as described in Section 5.6) to support the considered CCUs, which are loaded into the FPGA. The annotated benchmark program is executed on the Molen prototype. The new CCU configuration and the PowerPC cycles for the entire data processing are counted again. The ratio between the execution cycle num-

Table 6.4: Overall MJPEG speedup by the DCT* Molen CCU implementation.

sequence	frame No	Total MJPEG execution [cycles]		Overall Speedup S_i for $a_i = 0.61, s_i = 6.22$		
		Software	DCT* CCU	exper.	S_{max}	% of S_{max}
tennis	1	84556800	40307208	2.10	2.57	81.69
	2	84615272	40393200	2.09	2.56	81.67
	3	84689544	40462000	2.09	2.56	81.69
	4	84629288	40439904	2.09	2.56	81.59
	5	84615808	40436592	2.09	2.57	81.57
	6	84594184	40409512	2.09	2.57	81.58
	7	84471640	40308680	2.10	2.57	81.50
	8	84434216	40263576	2.10	2.57	81.49
barbara	1	85371112	41131512	2.08	2.53	81.94
artemis	1	85577112	41354208	2.07	2.52	82.01

bers before and after the program code annotation gives the actual speedup of the benchmark. For the experiments we considered three picture sequences: *tennis*, *barbara*, and *artemis*, the first one comprising eight 48×48 frames, the latter two comprising only a single frame. Table 6.4 contains the experimental data on these three sequences. Column 3, labelled "Software", contains the total number of cycles, required by the entire MJPEG application, when running as a pure software. The next column, labelled "DCT* CCU", indicates the total number of cycles, required by the MJPEG execution on the Molen prototype configured with the automatically generated DCT* CCU. In column 6 ("exper."), the experimentally attained speedups are presented, as the numbers from column 2 are divided by the numbers in column 4, thus straightforwardly calculating the overall MJPEG speedup. We are also interested how close we are to the theoretically maximum attainable speedups S_{max}, as devised in Equation (6.4), therefore, we carried out additional experiments, to obtain parameters a_i and s_i. For parameter a_i, we simply measured the number of cycles required to execute the DCT* kernel in software and divided it by the total number of MJPEG execution cycles. Thus, we calculated, that the DCT* software kernel constitutes roughly 61% of the total execution time of the pure software MJPEG encoding algorithm. Employing Equation (6.4) with the calculated exact values of a_i results to the theoretical speedups, reported in column 6 of Table 6.4. Finally, in column 7, we estimate how close the experimentally measured speedups are to the theoretical maximum.

Table 6.5: MPEG-2 profiling results for the considered functions.

MPEG-2 application		encoder				decoder
sequence	# frames@Res.	SAD	DCT	IDCT	Total	IDCT
carphone	96@176x144	51.1 %	12.5 %	1.3 %	64.9 %	50.4 %
claire	168@360x288	53.8 %	11.8 %	1.0 %	66.6 %	37.6 %
container	300@352x288	56.2 %	10.7 %	1.0 %	67.9 %	40.4 %
tennis	112@352x240	60.0 %	9.5 %	0.8 %	70.3 %	40.5 %

6.3.2 MPEG-2 experimentally projected evaluation

We target the Berkeley implementation of the MPEG-2 encoder and decoder included in libmpeg2. Following, we describe the experiments that have been carried out and report the obtained results. To calculate the projected speedup of the entire MPEG-2 application, we employed the experimentally projected evaluation approach, because the compiled application code is too large to fit into the implemented program memory of the Molen prototype. We profile the application after running it on a larger system with a PowerPC processor. The profiling data are used to identify and design performance critical kernels as CCU implementations. We run the extracted kernels on the prototype Molen processor and directly measure the performance gains. Using these measurements, the profiling data, and the Amdahl's law interpretation from Section 6.1, we estimate the projected overall speedup, rather than directly run the entire MPEG-2 application on the Molen prototype.

Software profiling results. The first phase of the experimentation is to identify the kernels, which consume most of the application execution time. These kernels will be considered as candidates for reconfigurable hardware implementations. Due to the memory limitations of the Molen prototype implementation, the compiled MPEG-2 code did not fit in the available program memory. Therefore, we profiled the application on a separate system with a PowerPC 970 processor running at 1600 MHz. The input data comprised a set of four popular video sequences, namely *carphone*, *claire*, *container* and *tennis*. Profiling results for each considered function and its descendants (obtained with the GNU profiler **gprof**) are presented in Table 6.5 per sequence. For the MPEG-2 encoder, the total execution time spent in SAD, DCT and IDCT operations (Table 6.5, column 6) emphasizes that these functions require around 2/3 of the total application time. Note, that although the IDCT function in MPEG-2 encoder takes only around 1% of the total encoding time

(Table 6.5, column 5), in the MPEG-2 decoder it requires on average around 42% of the total decoding. Also note, that the profiling results are data dependent and slightly vary per data sequence. Consequently, all three considered functions are good candidates for hardware implementations although their individual part of the total execution time vary per sequence and per (encoder or decoder) application.

Local kernel speedups. We have embedded the considered CCU implementations of SAD, DCT and IDCT within the Virtex II Pro Molen prototype and carried out experiments in two stages:

Stage 1. Extract the kernels of interest from the original MPEG-2 application source code used in the profiling phase without any further code modifications. Compile these software kernels for the original PowerPC ISA and run them on one of the embedded PowerPC405 processors. Obtain the number of PowerPC cycles consumed per kernel execution.

Stage 2. Substitute the kernel software code with a new piece of code to support πISA. Compile the new code. Load the CCU configuration supporting the corresponding kernel into the reconfigurable processor. Run the newly compiled CCU-enabled kernel on the Virtex II Pro Molen prototype and obtain the number of PowerPC cycles consumed during its execution.

For our experiments, we considered the same data sequences as used in the profiling phase. In both stages, the PowerPC timers are initialized before a kernel is executed and are read immediately after the kernel execution has completed. Thus, the exact number of PowerPC cycles, required for the entire kernel execution is obtained. Figure 6.2 depicts in logarithmic scale the measured cycles obtained in the two experimentation stages for each of the three kernels, considered in the experiments. It has been noted that the SAD, the DCT and the IDCT software implementations are slightly data dependent. Therefore, there are four chart groups illustrated in Figure 6.2, which depict the cycle numbers consumed in the software execution of each testbench sequence. On the contrary, the CCU implementations of all three kernels are data independent. This implies that the same processing cycles are required for the same amount of data, regardless the contents of the benchmark data sequence. Therefore, only one chart group (the last one in Figure 6.2) presents the cycle numbers, consumed by the Molen prototype. In Figure 6.2, only results for fixed $\rho\mu$-code CCU implementations are depicted. Additionally, we have considered both fixed and pageable $\rho\mu$-code implementations for SAD16 and SAD128 and the obtained execution cycle numbers are reported in Table 6.6. The cycle numbers of the right-most chart group and in Table 6.6 include all kernel related

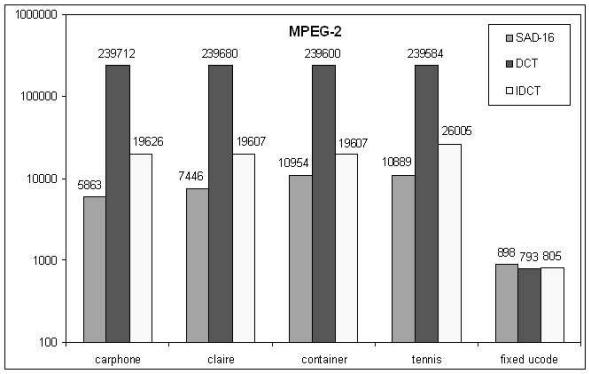


Figure 6.2: Kernels execution cycles for PowerPC ISA and fixed $\rho\mu$-code.

Table 6.6: Cycle numbers for different SAD implementations.

	SAD16	SAD128	SAD256
fixed $\rho\mu$-code	898	311	264
peageable $\rho\mu$-code	914	331	284

XREG transfers, memory transfers and data processing. The pagable $\rho\mu$-code total cycle numbers in Table 6.6 include the transfers from the main memory to the ρ-control store, as well. After obtaining the execution cycle numbers for each kernel both on PowerPC and on the Molen prototype, the kernel speedup is calculated for all data sequences with respect to each CCU implementation. Table 6.7 presents the calculated kernel speedups.

Projected application speedup: Consider Equation 6.1. Further consider parameters a_i to be the profiling results reported in Table 6.5 and parameters s_i- the local kernel speedups in Table 6.7. Thus, employing Equation 6.1, the overall speedup figures for the entire MPEG-2 encoder and MPEG-2 decoder

Table 6.7: Local speedup for the MPEG-2 kernels considered ($s_i = \frac{T_{SEi}}{T_{\rho i}}$).

	SAD16		SAD128		SAD256		DCT	IDCT
	fixed	pag.	fixed	pag.	fixed	pag.	fixed	fixed
carphone	6.5	6.4	18.9	17.7	22.2	20.6	302.3	24.4
claire	8.3	8.1	23.9	22.5	28.2	26.2	302.2	24.4
container	12.2	12.0	35.2	33.1	41.5	38.6	302.1	24.4
tennis	12.1	11.9	35.0	32.9	41.2	38.3	302.1	32.3

Table 6.8: Projected overall MPEG-2 speedup per kernel ($Si = \frac{1}{1-(a_i-\frac{a_i}{s_i})}$).

	encode								decode
	SAD16		SAD128		SAD256		DCT	IDCT	IDCT
	fixed	pag.	fixed	pag.	fixed	pag.	fixed	fixed	fixed
carphone	1.76	1.76	1.94	1.93	1.95	1.95	1.14	1.01	1.94
claire	1.90	1.89	2.06	2.06	2.08	2.07	1.13	1.01	1.56
container	2.07	2.06	2.20	2.20	2.21	2.21	1.12	1.01	1.63
tennis	2.22	2.22	2.40	2.39	2.41	2.41	1.10	1.01	1.65

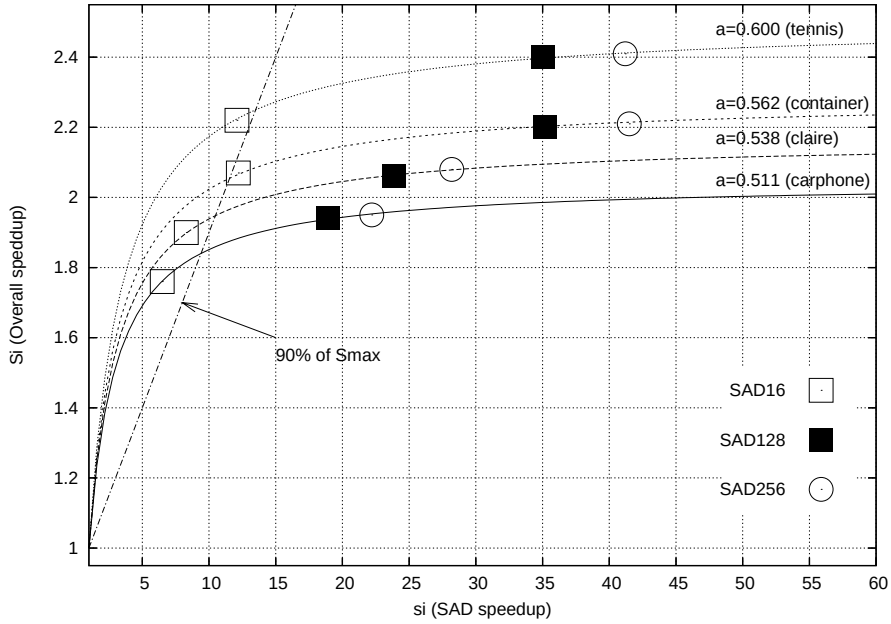


Figure 6.3: Overall MPEG-2 encoder speedup with three SAD configurations.

are projected per kernel contribution and results are reported in Table 6.8. The data related to the three considered SAD implementations and reported in Tables 6.5, 6.7, and 6.8 are summarized and illustrated in Figure 6.3. The figure depicts the overall MPEG-2 encoder speedup depending on the local speedup of the SAD operation for the four considered data sequences. The speedup, attained by each of the proposed SAD configurations is marked with different symbols. Figure 6.3 suggests that the SAD16 configuration alone (empty squares) can speedup the entire MPEG-2 encoder by less or around 90% of the maximum theoretically attainable speedup. Obviously, SAD128 (solid

Table 6.9: Overall speedup estimations for the entire MPEG-2.

	MPEG2 encoder*			MPEG2 decoder		
	theory	impl.	impl./th.	theory	impl.	impl./th.
carphone	2.85	2.64	93%	2.02	1.94	96%
claire	2.99	2.80	94%	1.60	1.56	98%
container	3.12	2.96	95%	1.68	1.63	97%
tennis	3.37	3.18	94%	1.68	1.65	98%

* fixed $\rho\mu$-code SAD128 + DCT + IDCT

squares) and SAD256 (circles) CCU implementations outperform SAD16 allowing more than 90% of the theoretical limit to be attained in speedup, which is due to their parallel processing organization. Though SAD256 clearly outperforms SAD128 in processing the SAD kernel alone, the overall impact of this processing superiority over the entire MPEG-2 decoder is negligible. Due to the fact that both SAD128 and SAD256 configurations are in the saturation zone of the overall performance curves in Figure 6.3, both of them perform with almost identical overall efficiency. Therefore, we can conclude that from the three proposed SAD configurations, SAD128 is the optimal one, because it severely outperforms SAD16. On the other hand, the hardware complexity of SAD128 is dramatically (twice) lower compared to SAD256 at the cost of a slight performance decrease. Similar analysis can be carried out for the DCT and IDCT implementations, as well as for an arbitrary design, considered for a CCU implementation.

That is, *the overall performance speedup of the Molen processor depends non-linearly on the individual performance of the implemented reconfigurable kernels. Experiments suggest that a saturation point is reached, beyond which further local kernel accelerations are inefficient application-wide.*

So far, we have analyzed the individual impact of a reconfigurable kernel on the overall performance of an application. Let us focus on the combined influence of several reconfigurable kernels on the overall performance. Consider Equation (6.2) and the experimental results in Tables 6.5 and 6.7. We calculated the projected overall speedup figures for the entire MPEG-2 encoder and MPEG-2 decoder applications and report them in Table 6.9. Columns labelled "theory" present the theoretically attainable maximum speedup (S_{max}) calculated with respect to Equation (6.4) and illustrated in Figure A.1. Columns labelled with "impl." contain data for the projected speedups with respect to the considered Molen implementation and Equation (6.2). For the MPEG-2 encoder, the simultaneous configuration of the SAD128, DCT, and IDCT

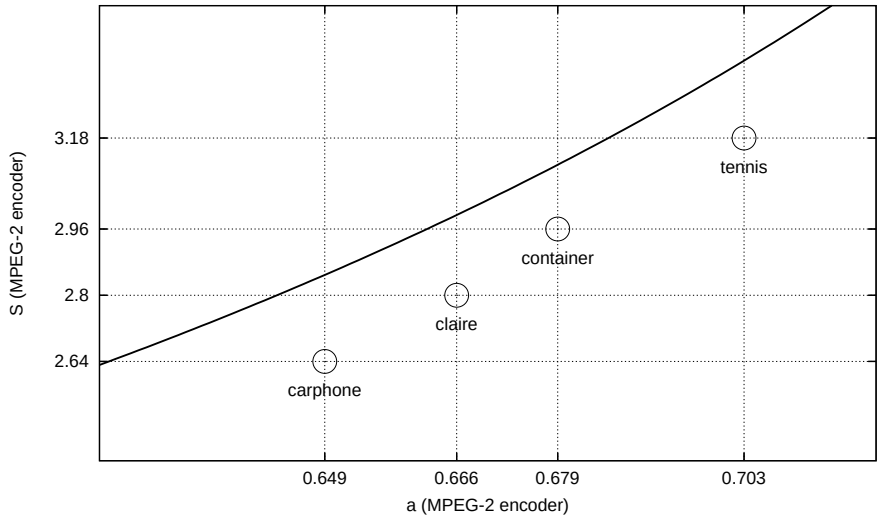


Figure 6.4: Experimental versus theoretical speedups.

operations employing fixed microcode implementations has been considered. For the MPEG-2 decoder, only the IDCT reconfigurable implementation has been employed. Columns with label ”imp./th.” in Table 6.9 indicate (in %) how close the real speedup is to the theoretically attainable one. Reported results strongly suggest that the actual speedups of the MPEG-2 encoder and decoder obtained during our practical experimentation very closely approach the theoretically estimated maximum possible speedups, which is graphically illustrated in Figure 6.4.

Speedup amplification effect: Figure 6.5 illustrates how the nonlinearity of the speedup curve influences the overall MPEG-2 encoder speedup. On the left-hand side of the figure, experimental results for the individual CCU implementations of IDCT, DCT and SAD are depicted. Note that when DCT is implemented alone, just 12,5% from the application are accelerated yielding a total of 14% overall acceleration. When SAD is implemented alone, the speedup is 1.94×, i.e., an acceleration of 94%. If both SAD and DCT are implemented altogether, the speedup is 2.55, which is 31.4% acceleration with respect to the SAD alone implementation. Thus, the contribution of the DCT CCU implementation to the overall speedup is amplified more than twice in the SAD+DCT configuration (i.e., 14% vs. 31.4% DCT-affected overall acceleration). The more to the right on the overall speedup curve a unit operates, the stronger the *speedup amplification effect*. This is proved once again by the

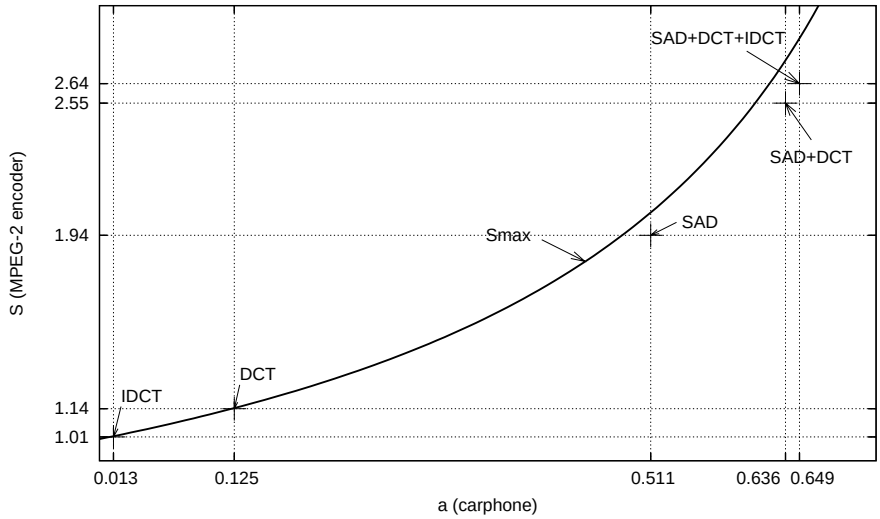


Figure 6.5: Influence of nonlinearity on the overall MPEG-2 encoder speedup.

SAD + DCT + IDCT configuration in Figure 6.5 where a speedup of 2.64 is attained versus 2.55 for the SAD+DCT configuration. It is a 3.5% acceleration caused by the IDCT which, due to its small part of the total execution time, contributes with only around 1% to the overall acceleration if implemented alone (i.e., when operating in the left-most part of the overall speedup curve).

Summary: The MPEG-2 application was accelerated very closely to its theoretical limits by implementing SAD, DCT and IDCT as reconfigurable coprocessors in the Molen Virtex II Pro prototype. The MPEG-2 encoder overall speedup was in the range between 2.64 and 3.18 while the speedup of the MPEG-2 decoder varies between 1.65 and 1.94.

6.3.3 MPEG-4 theoretically estimated speedup

Experimental results for the MJPEG and MPEG-2 applications, reported in the previous two subsections, indicate that the Virtex II Pro Molen processor can approach application speedups very close to the theoretically attainable maximum speedups. The two example applications considered so far, however, are already established benchmarks in the media domain. Therefore, we have been able to evaluate their performance in entirety on the prototype Molen platform. As already mentioned, in this dissertation we are also addressing future media applications and more specifically, the applications related to the high-

est MPEG-4 visual profiles and levels, which are not available in a mature and efficient software code to date. Therefore, we consider only separate kernels of the Core and Main MPEG-4 profiles as we implement them in the Molen context and evaluate local speedups against their pure software implementation. For the impact of these local kernel speedups on the overall MPEG-4 execution, we can only speculate until a mature, performance efficient MPEG-4 testbench becomes available. These speculations can be founded on the established Amdahl's low based methodology, which proved to be applicable in the previous two examples.

Experimental framework: We considered the publicly available source code of the MPEG-4 MoMuSys project. The application is written in strict ANSI C language and presents an implementation of the MPEG-4 video verification model [1]. The code is far from efficient performance-wise, it is full with redundant pieces of code, which execute identical functionalities. Therefore, this application can not be considered as an established benchmark and it is only a functional verification of the standard. Many separate subroutines of this application, however, are programmed efficiently and can be considered for evaluation. Such are the routines implementing the repetitive padding algorithm and the ACQ function, considered in Chapter 3. We extracted these kernels, compiled them for PowerPC and executed them on the Molen prototype. We also employed the related hardware accelerators as CCUs, annotated the kernel software code according to the Molen rules, and run the annotated code on the prototype configured with the related CCUs. Eventually, we measured the performance gains, just like we did for MPEG-2.

MPEG-4 repetitive padding: We resynthesized the padding unit, described in Chapter 3, for the Virtex II Pro technology and embedded it as a CCU in the Molen prototype. For the considered configurations of 4, 8 and 16 processing elements, synthesis tools reported maximal operating frequencies of 137 MHz, 107 MHz, and 92 MHz, respectively. These results suggested that all three configurations can be clocked by the *mem_clk* signal of the implemented prototype, i.e., we clocked them at 83 MHz (see Chapter 5). The padding CCU reads the texture and shape data for each boundary block, processes this block, and returns its padded texture data back into the main Molen memory. We noticed that the compiler code optimization effort influences the measured numbers of cycles, therefore we carried out experiments both without compiler optimization (option O0) and maximum optimization (O3). Experimental results are presented in table 6.10. We considered several BAB patterns to evaluate the performance gains, because the software implementation of the padding algorithm is data dependent unlike the CCU implemented padding

Table 6.10: PowerPC cycles for the repetitive padding algorithm per block.

Padding	Software -O0		Software -O3		CCU -O0		CCU -O3	
	8 x 8	16x16	8 x 8	16x16	8 x 8	16x16	8 x 8	16x16
PATT0	42581	132117	11637	36277				
PATT1	70329	259449	17865	65097				
PATT2	68233	232065	16889	54873	599	1143	368	912
PATT3	46088	232161	17445	52929				
PATT4	70820	256068	17093	60661				
Average	59610	222372	16186	53967	599	1143	368	912

units. The BAB patterns are notated by PATT*N* and an average cycle number for the considered BAB patterns is presented. Cycle numbers are reported for pure software PowerPC implementation and Molen Padding CCUs considering no compiler optimization (-O0), maxim optimization (-O3), and 8×8 and 16×16 processed block dimensions. We note that in the Molen execution scenario the measured cycle numbers include all data transfers between the CCU and the main memory, all XREGs transfers, the related address arithmetic, and the actual processing time. The address arithmetic is introduced to calculate the starting addresses of the texture and shape data in the main memory and is implemented in C code, therefore its execution in time depends on the compiler optimization capabilities. A deeper analysis of the CCU scenario cycle numbers suggests that the memory transfer cycles and data processing cycles are constant. That is:

$$T_{CCU} = T_{mem} + T_{padd} + T_{cd} \tag{6.5}$$

Where T_{CCU} denotes the total execution time of the padding kernel in the Molen CCU scenario in cycle numbers, T_{mem} - cycles for memory transfers, T_{padd} - cycles for the actual padding processing, and T_{cd} - a compiler dependent cycle number. Table 6.11 presents the cycle numbers for T_{mem}, T_{padd}, and T_{cd}. Based on the experimental results in Table 6.10, local kernel speedups per block are calculated and reported in Table 6.12.

The MPEG-4 ACQ function: We implemented the ACQ function design introduced in Chapter 3 as a CCU in the Virtex II Pro Molen prototype. The proposed structure was augmented with registers for the input BAB data and for the output result of the function. For the experiments, we considered only a 16PE structure, processing an entire 16×16 macroblock in two cycles. The ACQ CCU first loads all necessary data from the main memory and then processes them. The result of the ACQ computations is available in the corre-

Table 6.11: PPC cycles for T_{mem}, T_{padd}, T_{cd}, and T_{CCU}.

	8 x 8	16x16
T_{mem}	160	640
T_{padd}	72	136
T_{cd}-O0	367	367
T_{cd}-O3	136	136
T_{CCU}-O0	599	1143
T_{CCU}-O3	368	912

Table 6.12: Repetitive padding local speedups by the Molen prototype.

	si -O0		si -O3	
	8 x 8	16x16	8 x 8	16x16
PATT0	71	116	32	40
PATT1	117	227	49	71
PATT2	114	203	46	60
PATT3	77	203	47	58
PATT4	118	224	46	67
Average	100	195	44	59

Table 6.13: I/O parameters and data of the ACQ CCU.

Parameter	Description	Location	Size
original BAB address	input parameter1	XREG1	32 bits
decoded BAB address	input parameter2	XREG2	32 bits
α threshold	input parameter3	XREG3	9 bits
original BAB	input data	data memory	256 Bytes
decoded BAB	input data	data memory	256 Bytes
ACQ result	output parameter1	XREG4	1 bit

sponding XREG two cycles after the data are loaded into the local ACQ buffer. The input and output parameters of the ACQ CCU implementation are summarized in Table 6.13. As indicated, the total number of the parameters exchange registers is 4. Synthesis results for the implemented ACQ CCU version with 16 PE are reported in Table 6.14. We run the ACQ CCU on the Molen prototype as we considered two compiler optimization options again. Without optimization (-O0), the function was completed for 640 PowerPC cycles, considering -O3, the ACQ is computed in 415 PowerPC cycles. These computational times include all data and parameter transfers and are dominated by the BAB data

Table 6.14: ACQ CCU synthesis results for Virtex II Pro
(local data buffer not considered).

Device xc2vp20-5	used	available	%
Number of Slices	561	10304	5
Number of Slice Flip Flops	81	20608	<1
Number of 4 input LUTs	871	20608	4
Maximum Frequency [MHz]	90	N.A.	N.A.

Table 6.15: PowerPC cycles for the ACQ function per 16×16 BAB.

(CCU cycles, considered for s_i, are 640 with -O0, and 415 with -O3.)

α th	Pattern	ACQ	SW -O0		SW -O3		s_i -O0	s_i -O3
			Origin.	Fast	Origin.	Fast		
0	PAT00	1	202981	51751	28303	23391	81	56
	PAT16	0	164977	42087	22999	19071	66	46
	PAT32	0	50905	13183	7063	6087	21	15
	PAT64	0	50905	13167	7063	6087	21	15
	PAT128	0	50905	13151	7063	6087	21	15
	ALL0	0	50933	13119	7070	6087	21	15
	ALL255	0	13033	3495	1790	1791	5	4
16	PAT00	1	202981	51751	28304	23383	81	56
	PAT16	1	202981	51743	28304	23383	81	56
	PAT32	0	50905	13183	7064	6075	21	15
	PAT64	0	50905	13167	7064	6075	21	15
	PAT128	0	50905	13151	7064	6075	21	15
	ALL0	0	50933	13119	7071	6075	21	15
	ALL255	0	13033	3495	1791	1779	5	4
32	PAT00	1	202981	51751	28304	23383	81	56
	PAT16	1	202981	51743	28304	23383	81	56
	PAT32	1	202981	51687	28304	23383	81	56
	PAT64	0	50905	13167	7064	6075	21	15
	PAT128	0	50905	13151	7064	6075	21	15
	ALL0	0	50933	13119	7071	6075	21	15
	ALL255	0	13033	3495	1791	1779	5	4
64	PAT00	1	202981	51751	28304	23383	81	56
	PAT16	1	202981	51743	28304	23383	81	56
	PAT32	1	202981	51687	28304	23383	81	56
	PAT64	1	202981	51623	28304	23383	81	56
	PAT128	0	50905	13151	7064	6075	21	15
	ALL0	0	50933	13119	7071	6075	21	15
	ALL255	0	13033	3495	1791	1779	5	4
128	PAT00	1	202981	51751	28304	23383	81	56
	PAT16	1	202981	51743	28304	23383	81	56
	PAT32	1	202981	51687	28304	23383	81	56
	PAT64	1	202981	51623	28304	23383	81	56
	PAT128	1	202981	51559	28304	23383	81	56
	ALL0	0	50933	13119	7071	6075	21	15
	ALL255	0	13033	3495	1784	1779	5	4
Average			113933	29121	15869	13252	46	32

loads, as the actual processing takes only 8 PowerPC cycles. To determine the execution time of the ACQ software kernel, required to calculate the attained speedup, we extracted the relevant pieces of code from the MoMuSys application. We already mentioned that this is not an optimal MPEG-4 implementation, which was confirmed by our experiments. Therefore, we modified the original software code of the ACQ kernel to execute almost 4 times faster, as suggested by the experimentally measured execution cycle numbers. Table 6.15 reports the experimental results from running the software ACQ kernel on the Molen prototype. We considered a single original BAB pattern and patterns with different distortions from the original as decoded BABs. The pattern

number in column 2 of Table 6.15 is equal to the value of the α threshold, for which the ACQ function result switches from 0 to 1 (see column 3), i.e., *the maximum shape distortion for which the decoded quality is still acceptable, given the α threshold.* For example, PATT32 has an acceptable shape quality (distortion) for α threshold ≥ 32, but not for α threshold < 32. The ALL0 and ALL255 patterns are entirely transparent and entirely opaque BABs, respectively. Experiments were carried out for all defined values of the α threshold (see column 1, Table 6.15). We report cycle numbers both for the original ACQ kernel code (Origin.) and our (Fast) ACQ kernel implementation without compiler optimization in columns 4-5, and with maximum compiler optimization (-O3) in columns 6-7. Considering the measured cycles from the CCU execution, we calculated the achieved local kernel speedups s_i and report them in the last two columns of Table 6.15 both in no compiler optimization and in maximum compiler optimization scenario. The reported s_i are with respect to our faster implementation of the ACQ kernel. The bottom line of Table 6.15 contains the average values for the considered items.

Overall MPEG-4 speedup: To date, there are no established benchmarks or efficient software implementations of the Core and Main profiles of MPEG-4 dealing with arbitrary shaped objects. Therefore, we determine the overall speedup of a high profile MPEG-4 application speculatively, based on some theoretically established profiling results, reported in the literature. Due to the diversity of the possible kernels implemented in an MPEG-4 context, such profiling results can not be obtained for a general MPEG-4 application, but rather for specific implementation scenarios (contexts). We consider profiling results, reported in [7, 46, 62, 63, 90], to establish different scenarios, which implement only the normative parts of the Core MPEG-4 profile. For each of the considered kernels, we assume the local speedups for their CCU implementations, experimentally measured on the Virtex II Pro Molen prototype. With all the experimental data and assumptions, we employ the Amdahl's law again. To support the evaluation of the overall MPEG-4 speedup, we define the *average local speedup*:

*The **average local speedup** is defined as the speedup of all sections of the application software comprising the kernels considered for reconfigurable hardware execution with respect to their summarized reconfigurable implementation.*

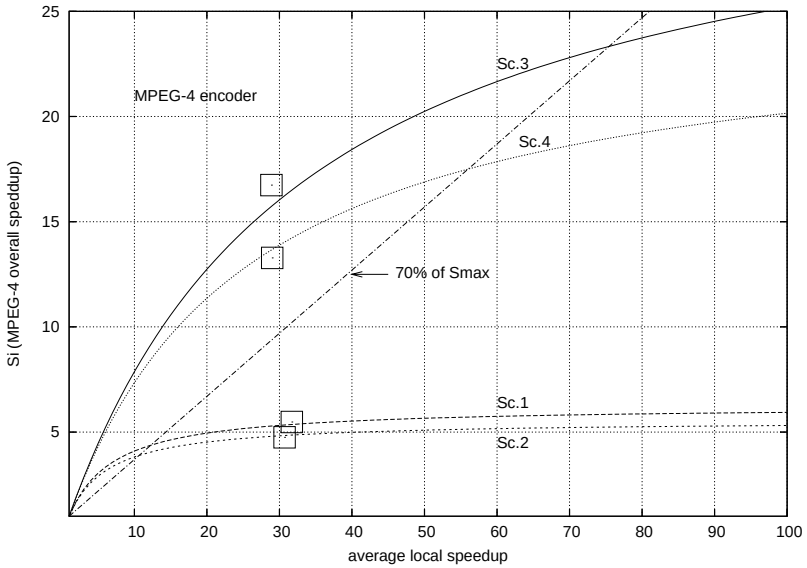
Table 6.16: Average local speedup in different MPEG-4 scenarios.

Scenario	a	Padding		ACQ		SAD		DCT		IDCT		$\bar{s}_{av}$
		a_i	s_i	a_i	s_i	a_i	s_i	a_i	s_i	a_i	s_i	
MPEG-4 Encoder												
Sc.1 [90]	0.84	0.038	100	0.104	46	0.660	28	0.006	300	0.010	26	31.7
Sc.2 [7]	0.82	0.039	100	0.104	46	0.660	28	0.005	300	0.008	26	30.7
Sc.3 [46]	0.97	0.001	100	0.064	46	0.900	28	0.004	300	0.005	26	28.9
Sc.4 [63]	0.96	0.007	100	0.056	46	0.880	28	0.008	300	0.007	26	29.0
MPEG-4 Decoder												
Sc.1 [90]	0.24	0,155	100	N.A.	46	N.A.	28	N.A.	300	0.089	26	49.1
Sc.2 [7]	0.24	0,155	100	N.A.	46	N.A.	28	N.A.	300	0.088	26	49.2
Sc.3 [46]	0.27	0,042	100	N.A.	46	N.A.	28	N.A.	300	0.226	26	29.4
Sc.5.1 [62]	0.16	0,160	100	N.A.	46	N.A.	28	N.A.	300	0.001	26	98.3
Sc.5.2 [62]	0.28	0,270	100	N.A.	46	N.A.	28	N.A.	300	0.010	26	90.8
Sc.5.3 [62]	0.23	0,140	100	N.A.	46	N.A.	28	N.A.	300	0.090	26	47.3

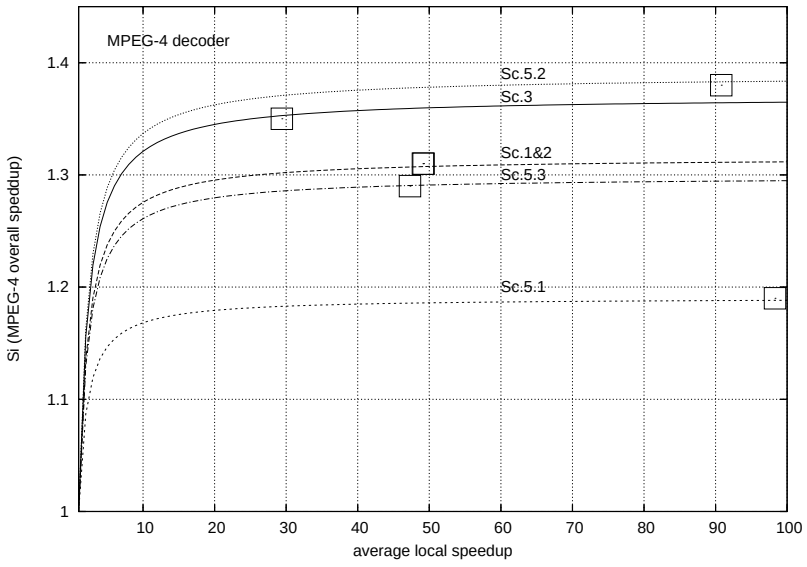
Denote the average local speedup by $\bar{s}_{av}$ and consider the notations from Equations (6.1) and (6.2). Formally, Definition 6.3.3 can be presented as follows:

$$\bar{s}_{av} = \frac{\sum T_{SEi}}{\sum T_{\rho i}} = \frac{a \cdot T}{\sum T_{\rho i}} = \frac{a}{\sum \frac{T_{\rho i}}{T}} = \frac{a}{\sum \frac{T_{SEi}}{s_i \cdot T}} = \frac{a}{\sum \frac{a_i}{s_i}} \tag{6.6}$$

With the help of $\bar{s}_{av}$, we can analyze the overall impact of all considered kernels on the MPEG-4 performance by substituting them with an imaginary single kernel. The average local speedup takes into account the individual contribution of each kernel with respect to its (of the kernel) local speedup and its part from the entire application execution. Table 6.16 presents the average local speedups calculated from experimental results for MPEG-4 high profile applications. The different scenarios are based on the profiling estimations from the publications referred in column one. Scenarios 5.1, 5.2, and 5.3 (extracted from [62]) correspond to bitstreams L6, L5, and Children, respectively. The theoretically estimated projected overall speedups of the same MPEG-4 high profile applications in the considered scenarios are presented in Table 6.17. Figure 6.6(a) graphically presents the influence of the average local speedup of the reconfigurable kernels on the overall speedup of a projected MPEG-4 encoder in the four considered scenarios. Identically, the influence on the overall MPEG-4 decoder speedup for the relevant six scenarios is depicted in Figure 6.6(b). Clearly, in the scenarios accelerating smaller parts of the application, the experimental speedup easily approaches the theoretically attainable maximum. Such are scenarios 1 and 2 for the encoder evaluation and all scenarios for the decoder evaluation, where overall speedup approaches 90% and above of the theoretical maximum. In the MPEG-4 encoder scenarios 3 and 4, however, the $\bar{s}_{av}$ value is not sufficient to attain even 70% of the theoretical maximum, illustrated by an explicit curve in Figure 6.6(a).



(a) MPEG-4 encoder;



(b) MPEG-4 decoder.

Figure 6.6: Projected MPEG-4 speedups in different scenarios.

Table 6.17: Estimated overall MPEG-4 speedups in different scenarios.

Scenario	a	$\bar{s}_{av}$	S	S_{MAX}
MPEG-4 Encoder				
Sc.1 [90]	0.84	31.7	5.48	6.41
Sc.2 [7]	0.82	30.7	4.75	5.43
Sc.3 [46]	0.97	28.9	16.74	38.46
Sc.4 [63]	0.96	29.0	13.28	23.64
MPEG-4 Decoder				
Sc.1 [90]	0.24	49.1	1.31	1.32
Sc.2 [7]	0.24	49.2	1.31	1.32
Sc.3 [46]	0.27	29.4	1.35	1.37
Sc.5.1 [62]	0.16	98.3	1.19	1.19
Sc.5.2 [62]	0.28	90.8	1.38	1.39
Sc.5.3 [62]	0.23	47.3	1.29	1.30

MPEG-4 evaluation summary: We implemented several MPEG-4 computationally demanding kernels on the Virtex II Pro prototype and measured directly the speedups versus their pure software implementation. For the repetitive padding and the ACQ kernels, specific and normative for the highest MPEG-4 profiles, we obtained average local speedups between 32X and 195X with respect to the considered compiler optimization and the size of the processed data. Due to the lack of an established MPEG-4 benchmark software and the various possible configuration contexts, we could only speculate on the overall attainable speedup. Considering profiling data, evaluated and reported in the related literature, we projected overall speedups between 20% and 16X depending on the particular MPEG-4 context implemented.

6.4 Conclusions

In this chapter, we evaluated the performance benefits of the Molen polymorphic media processor carrying out a series of experiments on the Virtex II Pro prototype. We considered three media applications: an MJPEG encoder, an MPEG-2 encoder-decoder, and a high-profile MPEG-4 encoder-decoder. Due to the limited resources, implemented in the utilized Molen prototype, and due to the different complexity of the considered media applications, we introduced three different approaches to analyze the performance benefits. We established the theoretical boundaries of the attainable speedups and compared our experimental data to the proposed theory. The MJPEG acceleration was evaluated

with a real experimental approach by running the entire application on the prototype processor and directly measuring the cycles consumed by both the pure software and the Molen scenario runs. A single CCU supporting four DCT related operations altogether was synthesized automatically for the MJPEG Molen execution and proved to give sufficient speedup. More precisely, around 82% of the theoretical maximum attainable speedup was experimentally measured. A different approach was considered for the evaluation of the MPEG-2 encoder and decoder. These applications appeared to be too large to fit into the implemented memory of the prototype, therefore they have been profiled in a bigger system. Three computationally demanding MPEG-2 kernels, consuming almost 70% of the entire software execution time, were implemented as CCUs. Their local speedups were measured using the real experimental approach on the Molen prototype. The essence of this approach was: utilizing the profiling data, the experimentally measured real kernel speedups, and the Amdahl's law, we projected the overall MPEG-2 speedup on the Molen processor. The projected speedups varied between 96% and 98% of the theoretically maximum attainable MPEG-2 speedups. We also considered future media applications, not available yet, more specifically- the arbitrary shaped objects processing in the high MPEG-4 profiles. We employed theoretical estimations based on experimentally measured real local speedups of several MPEG-4 computationally demanding kernels. Then, based on evaluations in the related literature, we speculated on the expected profiling data regarding the considered kernels and theoretically projected overall speedups between 1.2X and 16X for the implementation scenarios considered. Overall, we conclude that: *Media applications can be accelerated by the Molen reconfigurable processor dramatically.*

Chapter 7

General conclusions

We have addressed key performance drawbacks in media processing. As a solution, we have proposed a reconfigurable media augmentation of a general purpose processor, altogether referred to as the *Molen polymorphic media processor* and implemented the proposal into a fully operational processor prototype. To solve media computational problems, we have considered reconfigurable hardware units that perform media specific operations. The high data throughput requirements have been met by a proposed scalable memory organization, designed to deliver block organized visual data through a wide bandwidth. We have considered the Virtex II Pro™ technology as a prototyping platform. To evaluate the performance benefits of the prototype, we have carried out experiments on MJPEG, MPEG-2, and MPEG-4.

This chapter summarizes the contents of the dissertation, outlines its contributions and proposes future research directions. It is organized in three sections. Section 7.1 summarizes the main conclusions we obtained from the presented research efforts. In Section 7.2, we highlight the main contributions of this dissertation. Finally, in Section 7.3, we propose some open research directions motivated by short discussions.

7.1 Summary

In Chapter 2, the basis of the Molen processor was described as *the capability to control program execution and hardware reconfiguration from the software identically, utilizing emulation microcode, termed as $\rho\mu$-code*. Common shortcomings of recent reconfigurable proposals were considered by the Molen

polymorphic processor paradigm. We addressed opcode space explosion, ISA compatibility, technology dependence, design modularity, limited number of parameters, and parallel reconfigurable execution. The main advantages of the Molen approach can be summarized as follows:

- **Compact ISA extension.** For a given ISA, a single architectural extension comprising 4 to 8 additional instructions provides unlimited number of reconfigurable functionalities per single programming space. This realization is application independent and resolves the opcode space explosion problem as well as provides ISA compatibility and portability of reconfigurable programs.
- **Technology independent and modular design.** The design concept is not bound to any particular reconfigurable technology. It allows reconfigurable modules designed by a third party to be ported easily into the Molen organization.
- **Arbitrary number of parameters and parallel processing.** The Molen processor organization and the programming paradigm based on sequential consistency allow an arbitrary number of parameters as well as parallel executions of no data dependent operations.

In Chapter 3, we considered three computationally demanding kernels from the high profiles of MPEG-4. Two hardware approaches to realize the MPEG-4 repetitive padding algorithm in real-time were discussed. The first approach proposed a design of a simple dedicated systolic structure. The second approach described a scheme for general purpose ALU augmentation, which could accelerate the MPEG-4 padding algorithm by orders of magnitude. We proposed a pipelined implementation of the idea, thus preserving the original functionality and timing scheme of the target ALU. We proved that the proposed design is scalable by applying it on ALUs with different operand widths. At trivial hardware costs, we could easily achieve real-time performance figures at the most-demanding profile levels of MPEG-4 both by the systolic array implementation and by the augmented ALU. The results and discussions on the proposed hardwired padding units have been reported in [91–93]. Another performance demanding part of MPEG-4, the so-called accepted quality (ACQ) function, was also accelerated by a proposed systolic hardware implementation. Evaluations indicate capabilities of processing speeds far beyond the MPEG-4 real-time requirements. The proposed ACQ implementation was originally reported in [83]. In the same chapter, we also introduced a hardware accelerator of the Discrete Wavelet Transform (DWT) based on the so called

lifting scheme. Different performance enhancing design techniques were introduced in the unit, like pipelining, parallel module operation and data reuse. The performance evaluation of the unit suggested that for larger picture sizes and longer polynomial filters its processing efficiency grows. A brief description of the proposed DWT unit we published in [94]. All three considered functional accelerators were designed for implementations either as stand-alone units, or as custom computing units in the reconfigurable Molen coprocessor.

Chapter 4 introduces a scalable memory organization capable of addressing randomly aligned rectangular data patterns out of a virtual 2D data storage. High performance is achieved by reduced number of data transfers between memory hierarchy levels, efficient bandwidth utilization, and short hardware critical paths. In the proposed design, data are located in an array of byte addressable memory modules, which are loaded via an interface to a linearly addressable memory. Theoretical analysis proved the consistency and efficiency of the linear and the two-dimensional addressing schemes. A scalable implementation of the organization was evaluated by experimental reconfigurable synthesis. At reasonably small hardware costs, we achieved considerable speedups of up to 8x for an experimental case study design versus traditional linearly addressable memories. The design is envisioned to be more cost-effective compared to related works reported in literature. Though the proposed organization was intended for specific visual data processing applications, it can be also adopted by other general purpose applications, e.g., vector processing. We published a short description of the proposed memory organization in [95] and a complete, more elaborate work is presented in [96].

In Chapter 5, we proposed a working prototype of the Molen processor based on the Xilinx Virtex II Pro technology. With this prototype, we proved in practice that the Molen concept can be implemented by emulating reconfigurable operations by the ISA of the core GPP, without changing the design of the latter. Thus, we also proved the practical feasibility of the Molen paradigm in general. More specifically, we utilized the original PowerPC ISA to emulate the execution of the minimal functionally complete πISA on the Virtex II Pro platform FPGA. We proposed efficient designs of the potentially performance limiting parts of the $\rho\mu$-coded processor, namely: the arbiter, the $\rho\mu$-coded unit, the exchange registers, the memory organization, and the clock domains and distribution. We also addressed and investigated the generation, processing, memory alignment and loading of $\rho\mu$-codes and proposed alternative design considerations. Additionaly, we defined an open, non-restrictive, polymorphic interface for third party CCU designers based on a general microcode control, and supporting general access to the exchange registers and

the data memory. Synthesis results for the proposed ”backbone” Molen infrastructure, indicate just between 1% and 4% utilization per particular reconfigurable resource category of the selected Virtex II Pro xc2vp20 device. Thus, virtually the entire FPGA area remains available for CCU implementations. We also motivated a high-level software support for the Molen architecture giving practical tips how to program for the proposed Molen prototype.

Due to the closely coupled GPP and RP in the proposed prototype, high performance benefits were attained. Theoretical and experimental performance evaluations of the proposed prototype with some embedded CCUs were presented and analyzed in Chapter 6. We employed a performance evaluation methodology, comprising three approaches. These approaches were referred as to *real experimental*, *experimentally projected*, and *theoretical estimation*. The chapter established theoretical grounds to analyze the prototype performance for the considered applications MJPEG, MPEG-2, and MPEG-4. Automatically and manually generated CCU designs are utilized during the experiments. Based on the experimental results and employing theoretical analysis, we concluded that implementing a larger portion of the program execution time into performance efficient CCUs improves the overall performance speedup more dramatically than accelerating smaller portions of the execution time locally. Experimentally obtained speedup figures very closely approach the theoretically established maximum attainable speedups.

In [40, 97–100], we reported some of the considerations on the prototype design, as well as some performance evaluations and results.

We should note that in this dissertation we did not target the Molen processor beginning with a high level language (e.g., C) program. To this purpose, a compiler, not considered in the presentation, is required. Such a compiler is being developed currently and some results have been reported in [101, 102].

7.2 Contributions

The main contributions of this dissertation are highlighted below.

- Our major contributions in speeding up media applications are:
  - We proposed two hardware approaches to realize the MPEG-4 repetitive padding. To our best knowledge, our padding accelerators are the fastest and the most cost efficient designs supporting the MPEG-4 repetitive padding algorithm so far.

  - We designed a scalable structure performing the MPEG-4 ACQ function in hardware. This has been the first hardware implementation of the function ever reported in the literature.
  - We introduced an original design performing the Discrete Wavelet Transform based on the lifting scheme. The processing efficiency of the proposed design grows with the dimensions of the processed picture and the length of the particular polynomial filter implemented.
  - We proposed a scalable memory organization capable of addressing rectangular data patterns in a 2D data storage. High performance is achieved by reducing the number of data transfers between the memory hierarchy levels, efficient bandwidth utilization and short hardware critical paths. To date, regarding the considered rectangular access patterns, the design is envisioned to be the most cost-effective among related works. Though the proposed memory organization was intended for media specific algorithms, it can be also adopted by other general purpose operations, e.g., vector processing.

- We developed the $\rho\mu$-coded processing concepts to the maturity of a practical working implementation and introduced a working prototype, mapped on the Virtex II Pro technology of Xilinx. More specifically:

  - We proved that the Molen concept is feasible and can be implemented without having to redesign the original GPP. We developed a scheme for the synchronized operation of the core and the reconfigurable processor by emulating the Molen specific operations with the instruction set of the core (PowerPC) processor.
  - We proposed efficient designs of the potentially performance limiting parts of the $\rho\mu$-coded processor: the arbiter, the $\rho\mu$-coded unit, the exchange registers, the memory organization and the clock domains.
  - We proposed and implemented an open polymorphic interface for the potential CCU designers. The interface is determined both at software and at hardware level and is with minimal restrictions, thus enabling a wide range of third party media accelerating designs to be embedded as Molen CCUs.
  - The experiments with MJPEG, MPEG-2, and MPEG-4 on the Molen prototype suggest that speedups of 2X-3X can be expected.

We proved experimentally that in some scenarios of the considered media applications, the Molen processor can approach up to 98 % of the theoretically attainable maximum speedups.

7.3 Proposed research directions

We propose the following directions for continuation of the proposed research.

- In this dissertation, we considered the MPEG visual data compression standards. We paid special attention to the highest profiles of the MPEG-4 standard, but considered only natural video processing. Another computationally demanding part of the highest MPEG-4 profiles is the synthetic video processing which includes numerous algorithms from the computer graphics domain. An interesting research direction could be to investigate, e.g., the software algorithms for 2D and 3D mesh animations, facial and body animations. Specific kernels from these algorithms can be implemented as CCUs.
- We have not considered the MPEG-4 preprocessing stage, where the objects are initially determined. Computationally intensive algorithms involving edge detection and semantical object identification can be also targeted by the proposed Molen concept.
- The complete polymorphic ISA extension of eight instructions contains more potentials for application speedups. A more elaborate prototype can be developed to investigate experimentally the influence of the additional instructions on the overall application speedup.
- Considering the boundaries of the implementation efficiency of the reconfigurable units established in this dissertation, a tool for their automatic generation can be developed.
- In this dissertation, we focused on performance improvement of media applications without considering the power dissipation by the proposed designs. Low power constraints can be followed as a guideline for further research and design improvements. Low power Molen processors can extend the practical application domain of the Molen concept towards power critical implementations, e.g., mobile devices.

Appendix A

Amdahl's Law Illustrations

Figure A.1 illustrates Equation (6.4) giving the dependency of the maximum speedup theoretically attainable with respect on the portion of the application execution time considered for acceleration. Obviously, this dependency is not linear. If half of the execution time of a program ($a = 0.5$) is reduced to a

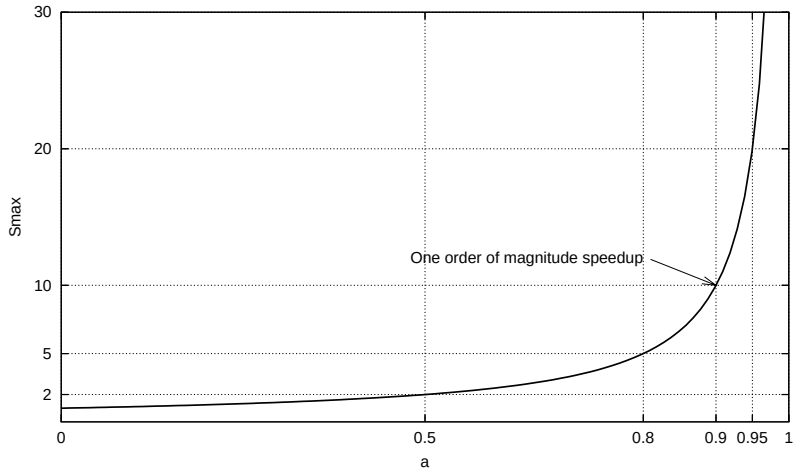


Figure A.1: Theoretically maximum attainable speedup, $S_{max} = \frac{1}{1-a}$.

minimum, the maximum attainable theoretical speedup will be 2. If, however, 80% of the application are accelerated, the theoretical limit of the speedup is already $5\times$. An order of magnitude acceleration is attainable only if more than 90% of the application are accelerated, as explicitly indicated in Figure A.1.

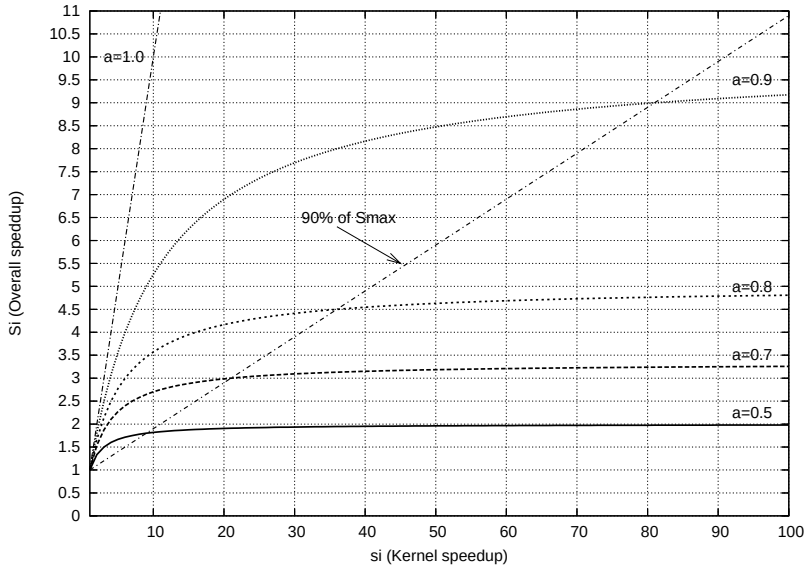


Figure A.2: Overall speedup dependance on the kernel speedup (different a).

Figure A.2 illustrates how the overall application speedup depends on the local speedup of a kernel. The dependency is depicted for values of a between 0.5 and 0.9 after Equation (6.1), assuming only one kernel, i.e., $a = a_i$ (a=1.0 is given only as a reference). Assume a practical value of the overall speedup to be 90% of the theoretically attainable maximum in acceleration, i.e., $S_i = 0.9 \times S_{max}$. Figure A.2 suggests that *to achieve 90% of S_{max}, a kernel that consumes between 50% and 90% of the entire application execution time, must be accelerated locally between 10 and 80 times, respectively.* In Figure A.2, this is illustrated by the intersecting points between the speedup curves for different a and the curve presenting 90% of S_{max}. We make this analysis more clear by the following example:

Example: Consider a kernel, which consumes 80% of the entire application execution time. According to Figure A.1, the maximum speedup that can be theoretically attained is 5 times. Figure A.2 suggests that 90% of this theoretical speedup can be achieved if the kernel is speeded up locally by a factor of about 36. Thus, the overall speedup of the application will be $0.9 \times 5 = 4.5$.

Similar analysis can be done for overall speedups less than 90% of S_{max}. In Figure A.2, the curves denoting less than 90% of S_{max} occupy some space to the left of the depicted curve for 90% of S_{max}, i.e., less local kernel speedups will be required.

Bibliography

[1] ISO/IEC JTC11/SC29/WG11, N3312, "MPEG-4 video verification model version 16.0."

[2] ISO/IEC JTC11/SC29/WG11 N4030, "MPEG-4 overview," March 2001.

[3] G. Blaauw and F. Brooks, *Computer Architecture: Concepts and Evaluation*. Addison-Wesley, 1997.

[4] S.G.Mallat, "A theory for multiresolution signal decomposition: The wavelet representation," *IEEE Trans. Patt. Anal. Mach. Intell.*, vol. 11, no. 7, pp. 674–693, 1989.

[5] Y. Q. Shi and H. Sun, *Image and Video Compression for Multimedia Engineering*. Boca Raton CRC Press, 2000.

[6] ISO/IEC JTC11/SC29/WG11 N2802, "ISO/IEC 14496-2. Generic Coding of Audio-visual Objects- Part2: Visual. Final Proposed Draft," July 1999.

[7] J. Kneip, S. Bauer, J. Vollmer, B. Schmale, P. Kuhn, and M. Reissmann, "The MPEG-4 video coding standard - a VLSI point of view," in *IEEE Workshop on Signal Processing Systems,(SIPS98)*, pp. 43–52, 8-10 Oct. 1998.

[8] S. Wong, S. Cotofana, and S. Vassiliadis, "Multimedia Enhanced General-Purpose Processors," in *International Conference on Multimedia and Expo*, pp. 1–4, July 2000.

[9] S. Wong, S. Cotofana, and S. Vassiliadis, "Coarse Reconfigurable Multimedia Unit Extension," in *9th Euromicro Workshop on Parallel and Distributed Processing PDP 2001*, pp. 235–242, February 2001.

[10] A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," *IEEE Micro*, vol. 16, pp. 42–50, August 1996.

[11] R. B. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro*, vol. 15, pp. 22–32, April 1995.

[12] R. B. Lee, "Subword Parallelism with MAX-2," *IEEE Micro*, vol. 16, pp. 51–59, August 1996.

[13] R.Razdan, *PRISC:Programmable Reduced Instruction Set Computer*. PhD thesis, Harvard University, May 1994.

[14] R. D. Wittig and P. Chow, "OneChip: An FPGA Processor With Reconfigurable Logic," in *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 126–135, April 1996.

[15] S.M.Trimberger, *Reprogramable Instruction Set Accelerator*. U.S. Patent No. 5,737,631, April 1998.

[16] J.R.Hauser and J.Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 12–21, April 1997.

[17] B. Kastrup, A. Bink, and J. Hoogerbrugge, "ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator," in *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 92–100, April 1999.

[18] P. M. Athanas and H. F. Silverman, "Processor Reconfiguration through Instruction-Set Metamorphosis," *IEEE Computer*, vol. 26, pp. 11–18, March 1993.

[19] M.Wazlowski, L.Agarwal, T.Lee, A.Smith, E.Lam, H.Silverman, and S.Ghosh, "PRISM-II Compiler and Architecture," in *Proc.IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 9–16, April 5-7, 1993.

[20] S. Hauck, T. Fry, M. Hosler, and J. Kao, "The Chimaera Reconfigurable Functional Unit," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 87–96, 1997.

[21] S. D. Haynes, P. Y. Cheung, W. Luk, and J. Stone, "SONIC - A Plug-In Architecture for Video Perocessing," in *Proceedings of the 9th International Conference on Field Programmable Logic and Applications (FPL 1999)*, pp. 23–30, LNCS 1673, 1999.

[22] T. Rissa, P. Y. K. Cheung, and W. Luk, "SoftSONIC: A Customisable Modular Platform for Video Applications," in *Proceedings of the 14th International Conference on Field Programmable Logic and Applications (FPL 2004)*, pp. 54–63, LNCS 3203, September 2004.

[23] T. Wiangton, P. Y. K. Cheung, and W. Luk, "A Unified Codesign Run-Time Environment for the UltraSONIC Reconfigurable Computer," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL 2003)*, pp. 396–405, LNCS 2778, September 2003.

[24] N. P. Sedcole, P. Y. K. Cheung, G. A. Constantinides, and W. Luk, "A Reconfigurable Platform for Real-Time Embedded Video Image Processing," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL 2003)*, pp. 606–615, LNCS 2778, September 2003.

[25] XILINX, *MicroBlaze RISC 32-Bit Soft Processor*. URL: http://www.xilinx.com/, 2002.

[26] ALTERA, *Nios 3.0 CPU Data Sheet*. URL: http://www.altera.com/, March, 2003.

[27] Tensilica, *Xtensa Product Brief*. URL: http://www.tensilica.com.

[28] M. Sima, S. Vassiliadis, S.Cotofana, J. van Eijndhoven, and K. Vissers, "Field-Programmable Custom Computing Machines - A Taxonomy," in *12th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 79–88, LNCS 2438, September 2002.

[29] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, 2002.

[30] S. Seng, W. Luk, and P. Y. K. Cheung, "Run-Time Adaptive Flexible Instruction Processors," in *Proceedings of the 12th International Conference on Field Programmable Logic and Applications (FPL 2002)*, pp. 545–555, LNCS 2438, 2002.

[31] M. Tremblay, J. O'Conner, V. Narayanan, and L. He, "VIS Speeds New Media Processing," *IEEE Micro*, vol. 16, pp. 10–20, August 1996.

[32] A. Turjan, T. Stefanov, B. Kienhuis, and E. Deprettere, "The Compaan Tool Chain: Converting Matlab into Process Networks," in *Designer's Forum of DATE 2002*, pp. 258–264, 2003.

[33] C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere, "Laura:Leiden Architecture Research and Exploration Tool," in *13th International Conference on Field Programmable Logic and Applications (FPL 2003)*, pp. 911–920, LNCS 2778, September 2003.

[34] A. L. Rosa, L. Lavagno, and C. Passerone, "Hardware/Software Design Space Exploration for a Reconfigurable Processor," in *Proc. of the DATE 2003*, pp. 570–575, 2003.

[35] M. Gokhale and J. Stone, "Napa C: Compiling for a Hybrid RISC/FPGA Architecture," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 126–135, April 1998.

[36] F. Campi, M. Toma, A. Lodi, A. Cappelli, R. Canegallo, and R. Guerrieri, "A VLIW Processor with Reconfigurable Instruction Set for Embedded Applications," in *In ISSCC Digest of Technical Papers*, pp. 250–251, Feb 2003.

[37] A. Ye, N. Shenoy, and P. Banerjee, "A C Compiler for a Processor with a Reconfigurable Functional Unit," in *ACM/SIGDA Symposium on FPGAs*, pp. 95–100, 2000.

[38] J. Becker and R. Hartenstein, "Configware and morphware going mainstream," *J. Syst. Archit.*, vol. 49, no. 4-6, pp. 127–142, 2003.

[39] S. Vassiliadis, S. Wong, and S. Cotofana, "The MOLEN $\rho\mu$-coded processor," in *11th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 275–285, August 2001.

[40] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The Molen Polymorphic Processor," *IEEE Transactions on Computers*, vol. 53, pp. 1363–1375, November 2004.

[41] S. Vassiliadis, S. Wong, and S. Cotofana, "Microcode processing: Positions and directions," *IEEE Micro*, vol. 23, pp. 21–30, July/August 2003.

[42] S. Vassiliadis, G. Gaydadjiev, K. Bertels, and E. Moscu Panainte, "The Molen Programming Paradigm," in *Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation*, pp. 1–7, July 2003.

[43] A. Padegs, B. Moore, R. Smith, and W. Buchholz, "The IBM System/370 vector architecture: Design considerations," *IEEE Transactions on Computers*, vol. 37, pp. 509–520, 1988.

[44] W. Buchholz, "The IBM System/370 vector architecture," *IBM Systems Journal*, vol. 25, no. 1, pp. 51–62, 1986.

[45] M. Moudgill and S. Vassiliadis, "Precise interrupts," *IEEE Micro*, vol. 16, pp. 58–67, January 1996.

[46] H.-C. Chang, L.-G. Chen, M.-Y. Hsu, and Y.-C. Chang, "Performance analysis and architecture evaluation of MPEG-4 video codec system," in *IEEE International Symposium on Circuits and Systems*, vol. II, pp. 449–452, 28-31 May 2000.

[47] H.-J. Stolberg, M. Berekovic, P. Pirsch, H. Runge, H. Moller, and J. Kneip, "The M-PIRE MPEG-4 codec DSP and its macroblock engine," in *IEEE International Symposium on Circuits and Systems*, vol. II, pp. 192–195, 28-31 May 2000.

[48] J. M. Rabaey, "Reconfigurable Processing: The Solution to Low-Power Programmable DSP," in *1997 ICASSP Conference*, pp. 275–278, April 1997.

[49] P. Athanas and H. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *IEEE Computer*, vol. 26, pp. 11–18, March 1993.

[50] S.C.Goldstein, H.Schmit, M.Moe, M.Budiu, S.Cadambi, R.R.Taylor, and R.Laufer, "PipeRench:a coprocessor for streaming multimedia acceleration," in *The 26th International Symposium on Computer Architecture*, pp. 28–39, May 1999.

[51] M. Putrino and S. Vassiliadis, "Resolution of branching with prediction," *International Journal of Electronics*, vol. 66, pp. 163–172, February 1989.

[52] ISO/IEC JTC11/SC29/WG11 N2485, "Implementation study group frequently asked questions," October 1998.

[53] XILINX, *DataSource CD-ROM*. XILINX, 2000.

[54] ALTERA, *Data Book*. Altera Corp., 1998.

[55] C. Chang, S. Vassiliadis, and J. Delgado-Frias, "An investigation of binary CLA and ripple CMOS adder designs," *Microprocessing and Microprogramming Journal*, vol. 40, pp. 1–21, January 1994.

[56] M. Putrino, S. Vassiliadis, and E. Schwarz, "Parallel binary byte adder / subtracter," *International Journal of Electronics*, vol. 65, pp. 139–153, February 1988.

[57] ISO/IEC JTC11/SC29/WG11, "New MPEG-4 profiles under consideration," July 2001.

[58] E. A. Edirisinghe, J. Jiang, and C. Grecos, "Shape adaptive padding for MPEG-4," *IEEE Transactions on Consumer Electronics*, vol. 46, pp. 514–520, August 2000.

[59] A. Kaup, "Object-based texture coding of moving video in MPEG-4," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 9, pp. 5–15, February 1999.

[60] J.-H. Moon, J.-H. Kweon, and H.-K. Kim, "Boundary block-merging (BBM) technique for efficient texture coding of arbitrarily shaped object," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 9, pp. 35–43, February 1999.

[61] C. Heer and K. Migge, "VLSI hardware accelerator for the MPEG-4 padding algorithm," in *IS&T:SPIE Conference on media processors*, vol. 3655, pp. 113–119, 1999.

[62] M. Berekovic, H.-J. Stolberg, M. B. Kulaczewski, P. Pirsh, H. Moler, H. Runge, J. Kneip, and B. Stabernack, "Instruction set extensions for mpeg-4 video," *Journal of VLSI Signal Processing*, vol. 23, pp. 27–49, October 1999.

[63] H.-C. Chang, Y.-C. Wang, M.-Y. Hsu, and L.-G. Chen, "Efficient algorithms and architectures for MPEG-4 object-based video coding," in *IEEE Workshop on Signal Processing Systems*, pp. 13–22, 11-13 Oct 2000.

[64] S. Vassiliadis, G. Kuzmanov, and S. Wong, "MPEG-4 and the New Multimedia Architectural Challenges," in *in Proceedings of the 15th International conference on Systems for Automation of Engineering and Research (SAER 2001)*, pp. 24–32, January 2001.

[65] W. Sweldens, "The lifting scheme: A custom-design construction of biorthogonal wavelets," *Appl. Comput. Harmon. Anal.*, vol. 3, no. 2, pp. 186–200, 1996.

[66] C.M.Brislawn, "Classification of nonexpansive symmetric extension transforms for multirate filter banks," *Applied and Comp. Harmonic Analysis*, vol. 3, pp. 337–357, 1996.

[67] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *J. Fourier Anal. Appl.*, vol. 4, no. 3, pp. 245–267, 1998.

[68] G. Fernández, S. Periaswamy, and W. Sweldens, "LIFTPACK: A software package for wavelet transforms using lifting," in *Wavelet Applications in Signal and Image Processing IV*, pp. 396–408, Proc. SPIE 2825, 1996.

[69] R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo, "Losless image compression using integer to integer wavelet transforms," in *International Conference on Image Processing (ICIP), Vol. I*, pp. 596–599, IEEE Press, 1997.

[70] R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo, "Wavelet transforms that map integers to integers," *Appl. Comput. Harmon. Anal.*, vol. 5, no. 3, pp. 332–369, 1998.

[71] D.C.Burger and T.M.Austin, "The simplescalar tool set, version 2.0," Tech. Rep. CS-TR-1997-1342, University of Wisconsin-Madison, 1997.

[72] P. Budnik and D. J. Kuck, "The organization and use of parallel memories," *IEEE Transactions on Computers*, vol. 20, pp. 1566–1569, December 1971.

[73] P. M. Kogge, *The Architecture of Pipelined Computers*. McGraw-Hill, 1981.

[74] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on Computers*, vol. C-24, pp. 1145–1155, December 1975.

[75] D. C. van Voorhis and T. H. Morrin, "Memory systems for image processing," *IEEE Transactions on Computers*, vol. C-27, pp. 113–125, February 1978.

[76] K. Kim and V. K. Prasanna, "Latin squares for parallel array access," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 4, pp. 361–370, 1993.

[77] D. lei Lee, "Scrambled Storage for Parallel Memory Systems," in *Proc.IEEE International Symposium on Computer Architecture*, pp. 232–239, May 1988.

[78] J. W. Park, "An efficient buffer memory system for subarray access," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, pp. 316–335, March 2001.

[79] R. F. Sproull, I. Sutherland, A. Thomson, S. Gupta, and C. Minter, "The 8 by 8 display," *ACM Transactions on Graphics (TOG)*, vol. 2, no. 1, pp. 32–56, 1983.

[80] J. Kneip, K. Ronner, and P. Pirsch, "A data path array with shared memory as core of a high performance DSP," in *Proceedings of the International Conference on Application Specific Array Processors*, pp. 271–282, August 1994.

[81] J. P. Wittenburg, M. Ohmacht, J. Kneip, W. Hinrichs, and P. Pirsh, "HiPAR-DSP: a parallel VLIW RISC processor for real time image processing applications," in *3rd International Conference on Algorithms and Architectures for Parallel Processing, 1997. ICAPP 97.*, pp. 155–162, December 1997.

[82] H. Kloos, J. Wittenburg, W. Hinrichs, H. Lieske, L. Friebe, C. Klar, and P. Pirsch, "HiPAR-DSP 16, a scalable highly parallel DSP core for system on a chip: video and image processing applications," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 3, pp. 3112–3115, IEEE, May 2002.

[83] G. Kuzmanov, S. Vassiliadis, and J. T. J. van Eijndhoven, "A 2D Adressing Mode for Multimedia Applications," in *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation (SAMOS 2001)*, pp. 291–307, LNCS 2268, July 2001.

[84] XILINX, *Virtex-IIPro Platform FPGA Handbook*. Xilinx, October 2002.

[85] J.S.S.M.Wong, *Microcoded Reconfigurable Embedded Processors*. PhD thesis, Delft University of Technology, December 2002.

[86] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the AFIPS 1967 Spring Joint Computer Conference*, pp. 483–485, 1967.

[87] B. Kienhuis, E. Rypkema, and E. Deprettere, "Compaan: deriving process networks from Matlab for embedded signal processing architectures," in *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES)*, pp. 13–17, May 2000.

[88] S. Vassiliadis, E. Hakkennes, J. Wong, and G. Pechaneck, "The Sum Absolute Difference Motion Estimation Accelerator," in *24th EUROMICRO conference (EUROMICRO 98)*, vol. 2, pp. 559–566, August 25-27 1998.

[89] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Proceedings of the IFIP Congress '74*, pp. 471–475, August 5-10 1974.

[90] P. Kuhn and W. Stechele, "Complexity analysis of the emerging MPEG-4 standard as a basis for VLSI implementation," in *SPIE Visual Comunications and Image Processing (VCIP)*, vol. 3309, pp. 498–509, Jan. 1998.

[91] G. Kuzmanov, S. Vassiliadis, and J. T. J. van Eijndhoven, "Hardwired MPEG-4 Repetitive Padding," *IEEE Transactions on Multimedia*, accepted for publication, to appear in August 2005.

[92] G. Kuzmanov and S. Vassiliadis, "ALU Augmentation for MPEG-4 repetitive padding," in *Proceedings of the 2002 Euromicro Conference on Massively-Parallel Computing Systems (MPCS'02)*, pp. 45–51, April 2002.

[93] G. Kuzmanov and S. Vassiliadis, "Reconfigurable repetitive padding unit," in *Proceedings of the 12 th ACM Great Lakes Symposium on VLSI (GLSVLSI'02)*, pp. 98–103, ACM Press, April 2002.

[94] G. Kuzmanov, B. Zafarifar, P. Shrestha, and S. Vassiliadis, "Reconfigurable DWT unit based on lifting," in *Proceedings of ProRISC 2002*, pp. 325–333, November 2002.

[95] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "Visual Data Rectangular Memory," in *Proceedings of the 10th International Euro-Par Conference, (Euro-Par 2004)*, pp. 760–767, LNCS 3149, September 2004.

[96] G. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis, "Multimedia Rectangularly Addressable Memory," *IEEE Transactions on Multimedia*, accepted for publication, to appear in 2005/2006.

[97] G. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis, "Loading $\rho\mu$-code: Design Considerations," in *Proceedings of the Third International Workshop on Computer Systems: Architectures, Modeling, and Simulation (SAMOS 2003)*, pp. 8–11, LNCS 3133, July 2003.

[98] G. Kuzmanov and S. Vassiliadis, "Arbitrating Instructions in an $\rho\mu$-coded CCM," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL 2003)*, pp. 81–90, LNCS 2778, September 2003.

[99] G. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis, "The MOLEN Processor Prototype," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, pp. 296–299, April 2004.

[100] G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis, "The Virtex II Pro MOLEN Processor," in *Proceedings of the Fourth International Workshop on Computer Systems: Architectures, Modeling, and Simulation (SAMOS 2004)*, pp. 192–202, LNCS 3133, July 2004.

[101] E. M. Panainte, K. Bertels, and S. Vassiliadis, "Compiling for the Molen Programming Paradigm," in *Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*, pp. 900–910, September 2003.

[102] E. Moscu-Panainte, K. Bertels, and S. Vassiliadis, "The PowerPC Backend Molen Compiler," in *14th International Conference on Field-Programmable Logic and Applications (FPL 2004)*, pp. 434–443, LNCS 3203, September 2004.

List of Publications

International Journals

1. G. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis. **Multimedia Rectangularly Addressable Memory**. *IEEE Transactions on Multimedia*, accepted for publication, to appear in 2005/2006.

2. G. Kuzmanov, S. Vassiliadis, and J. T. J. van Eijndhoven. **Hardwired MPEG-4 Repetitive Padding**. *IEEE Transactions on Multimedia*, accepted for publication, to appear in August 2005.

3. S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Moscu Panainte. **The Molen Polymorphic Processor**. *IEEE Transactions on Computers*, 53(11): 1363–1375, November 2004.

Conference Proceedings

1. G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis. **Visual Data Rectangular Memory**. In *Proc. of the Tenth International Euro-Par Conference (Euro-Par 2004)*, LNCS 3149, pages 760–767, September 2004.

2. G. Kuzmanov, G. Gaydadjiev, and S. Vassiliadis. **The Virtex II Pro MOLEN Processor**. In *Proc. of the Fourth International Workshop on Computer Systems: Architectures, Modeling, and Simulation (SAMOS 2004)*, LNCS 3133, pages 192–202, July 2004.

3. G. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis. **The MOLEN Processor Prototype**. In *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, pages 296–299, April 2004.

4. G. Kuzmanov and S. Vassiliadis. **Arbitrating Instructions in an $\rho\mu$-coded CCM**. In *Proc. of the 13th International Conference on Field Programmable Logic and Applications (FPL 2003)*, LNCS 2778, pages 81–90, September 2003.

5. G. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis. **Loading $\rho\mu$-code: Design Considerations**. In *Proc. of the Third International Workshop on Computer Systems: Architectures, Modeling, and Simulation (SAMOS 2003)*, LNCS 3133, pages 11–19, July 2003.

6. G. Kuzmanov, B. Zafarifar, P. Shrestha, and S. Vassiliadis. **Reconfigurable DWT unit based on lifting**. In *Proc. of ProRISC 2002*, pages 325–333, November 2002.

7. G. Kuzmanov and S. Vassiliadis. **Reconfigurable Repetitive Padding Unit**. In *Proc. of the 12 th ACM Great Lakes Symposium on VLSI (GLSVLSI'02)*, pages 98–103, ACM Press, April 2002.

8. G. Kuzmanov and S. Vassiliadis. **ALU Augmentation for MPEG-4 Repetitive Padding**. In *Proc. of the 2002 Euromicro Conference on Massively-Parallel Computing Systems (MPCS'02)*, pages 45–51, April 2002.

9. S. Vassiliadis, G. Kuzmanov, and S. Wong. **MPEG-4 and the New Multimedia Architectural Challenges**. In *Proc. of the 15th International Conference on Systems for Automation of Engineering and Research (SAER-2001)*, pages 24–32, September 2001.

10. G. Kuzmanov, S. Vassiliadis, and J. T. J. van Eijndhoven. **A 2D Adressing Mode for Multimedia Applications**. In *Proc. of Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation (SAMOS 2001)*, LNCS 2268, pages 291–307, July 2001.

Earlier publications, not directly related to this dissertation:

1. P. Manoilov, G. Kouzmanov, T. Stefanov, and A. Popov. **Development of a Low Area Custom Microprocessor Core**. *Automatics & Informatics Journal*, No 5, pages 33–39, 1999. (in Bulgarian, English abstract)

2. P. Manoilov, G. Kouzmanov, T. Stefanov, and A. Popov. **Two Approaches in One for a Quick and Efficient Design of Low Area Custom Microprocessor Cores**. In *Proc. of the 7th International Conference "Electronics'98"*, book 2, pages 57-64, Sozopol, Bulgaria, 1998.

Samenvatting

In dit proefschrift besteden wij aandacht aan hoge prestatie media verwerking. Wij stellen een herconfigureerbare media aanvulling op een standaard processor voor en implementeren deze in een volledig operationeel processor prototype, waaraan gerefereerd wordt als de *Molen polymorphic media processor*. De werking van het prototype is gebaseerd op het co-processor architectuur paradigma. Specifieker gesproken regelt een kern bestaande uit een standard processor de uitvoering en herconfigurering van de herconfigureerbare co-processor, waarbij de laatste ingesteld is voor specifieke media algoritmen. Wij stellen oplossing voor die essentiele media verwerkingsproblemen wat betreft *media specifieke verwerkingssnelheid* en *beperkte geheugen bandbreedte*. Om media reken problemen op te lossen beschouwen wij hardware eenheden die specifieke media operaties uitvoeren. Aandacht wordt besteed aan *MPEG-4* functionaliteit die tot op heden niet in brede zin onderzocht is. Het probleem van de beperkte geheugen bandbreedte is opgelost door de introductie van een schaalbare geheugen organisatie, welke een voldoende hoeveelheid data levert aan de eenheden die per blok georganiseerde visuele data verwerken. De experimenten suggereren een *8x snellere data verplaatsing*. Het voorgestelde Molen prototype is geimplementeerd met de Xilinx Virtex II Pro ™ technologie. Zonder de ingebouwde PowerPC kern opnieuw te ontwerpen emuleren wij de herconfigureerbare operaties door gebruik te maken van de originele PowerPC instructies. De gehele Molen ”basis” infrastructuur gebruikt *minder dan 1% van de herconfigureerbare grondstoffen* van de prototype chip xc2vp20. Als gevolg hiervan is zogoed als het gehele herconfigureerbare oppervlak beschikbaar voor de media verwerkingseenheden. Om het geimplementeerde Virtex II Pro™ Molen prototype te evalueren voeren wij experimenten met MJPEG, MPEG-2, en MPEG-4. Deze experimenten geven duidelijk aan dat de beschouwde aanpak gebruikt kan worden om media toepassingen te versnellen. Specifieker gesproken de behaalde resultaten suggereren dat een *totale applicatie versnelling van 2x-3x* verwacht mag worden, welke *tot 98% van de theoretisch maximaal haalbare versnelling* kan komen. Herconfigureerbare technologieen, anders dan de Virtex II pro™, worden ook beschouwd en laten zien dat het voorstel onafhankelijk is van de technologie.

Кратък обзор

Предмет на настоящата дисертация е високо производителната обработка на визуална информация, която съпътства редица медийни приложения. Конкретното предложение, наречено "полиморфичен медиен процесор Молен" (the Molen polymorphic media processor), се състои от процесор с общо предназначение, разширен с реконфигурируем хардуер и е реализирано върху функционално завършен прототип. Молен основава действието си на копроцесорен архитектурен модел, в който процесор с общо предназначение контролира изпълнението и конфигурацията на реконфигурируем копроцесор, настройвайки последния към специфични медийни алгоритми. Предложени са решения на ключови проблеми в медийната обработка, свързани с характерната висока изчислителна скорост и необходимостта да бъдат обменяни голямо количество данни с паметта за единица време. Проблемите от изчислително естество са разрешени с разработката на реконфигурируеми хардуерни устройства, изпълняващи специфични медийни операции. Особено внимание е отделено на някои функции в медийния стандарт MPEG-4, чиито потенциал за хардуерна реализация не бе напълно изследван преди настоящата дисертация. Проблемът с интензивния обмен на данни е разрешен посредством мащабируема организация на паметта, способна да захранва бързи устройства, които обработват блоково организирана визуална информация. Експериментален анализ показва до 8-кратно увеличение на скоростта за обмен на данни. Предложеният прототип на процесора Молен е реализиран върху технологията Virtex II Pro™ на Xilinx. Основно предимство е възможността, без да се препроектира вграденият процесор PowerPC, да се емулират реконфигурируеми операции посредством оригиналната му система от инструкции. Пълната скелетна инфраструктура на представения прототип на Молен заема по-малко от 1% от ресурсите на използвания реконфигурируем чип xc2vp20. По този начин практически цялата реконфигурируема площ остава на разположение за реализация на различни медийни хардуерни устройства. С цел оценка на реализирания прототип са представени експерименти върху програмни приложения на медийните стандарти MJPEG, MPEG-2 и MPEG-4. Експериментите ясно показват,

че предложените подходи могат да се приложат успешно при ускоряване на различни алгоритми за обработка на визуална информация. Резултатите доказват практически достижимо общо ускорение на разглежданите медийни приложения от порядъка на 2-3 пъти, като доближават 98% от теоретично изчисленото максимално бързодействие. Предложението е универсално и технологично независимо, за което свидетелстват и допълнително включените в настоящата дисертация изследвания върху реконфигурируеми платформи, различни от Virtex II Pro™.

Curriculum Vitae

Georgi KUZMANOV was born on the 12th of August, 1974 in Sofia, Bulgaria. In 1988 he was admitted in the Vocational High School of Microprocessor Technology - Pravetz, Bulgaria, where he obtained the qualification of a ”Technician in Microprocessor Technology” in 1993. The same year, Georgi Kuzmanov enlisted the Computer Systems Faculty of the Technical University of Sofia (TU Sofia), Bulgaria. In 1998, he received his M.Sc. degree and an engineering title in computer systems from TU Sofia, successfully defending his thesis titled ”Design, Analysis, and Area Minimization of the Operating Unit of an Application Specific Microprocessor Core”. The results of his graduation work have been implemented in a family of microprocessor cores produced by ”Info MicroSystems” Ltd., Sofia and their successor ”FabLess” Ltd., Sofia.

Between 1998 and 2000, Georgi Kuzmanov was with ”Info MicroSystems” Ltd., Sofia, where he had been involved in several reconfigurable computing and ASIC projects as a research and design engineer.

In June 2000, Georgi Kuzmanov joined the Computer Engineering (CE) lab of Delft University of Technology (TU Delft), The Netherlands, as a researcher, where he had been working towards his Ph.D. degree with scientific advisor prof. dr. Stamatis Vassiliadis. Recently, he has been employed as a full-time researcher in the same scientific group. This dissertation contains the outcome of his research activity in the CE Lab, TU Delft, during the period 2000-2004.

Since 1995 Georgi Kuzmanov is an IEEE member. His current research interests include: reconfigurable computing, video and image processing, multimedia embedded systems, computer arithmetic, computer architecture, and computer organization.