The TM3270 Media-processor

Jan-Willem van de Waerdt

The work described in this thesis was carried out at Philips Semiconductors, San Jose, USA.

On the front cover: Realization of the TriMedia TM3270 media-processor in a 90 nm CMOS process technology.
On the back cover: Another important tape-out in the year 2005.

CIP-gegevens Koninklijke Bibliotheek, Den Haag
van de Waerdt, Jan-Willem
The TM3270 Media-processor
Proefschrift Technische Universiteit Delft, - Met lit. opg.
ISBN 90-9021060-1
Trefw.: Media-processor, processor design

©Philips Electronics N.V. 2006
All rights are reserved.
Reproduction in whole or in part is prohibited
without the written consent of the copyright owner.

The TM3270 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 dinsdag 10 oktober 2006 om 15:00 uur
door

Jan-Willem VAN DE WAERDT

informatica ingenieur
geboren te Maarsbergen

Dit proefschrift is goedgekeurd door de promotor:

Prof.dr. S. Vassiliadis

Samenstelling promotiecommissie:

Rector Magnificus	Technische Universiteit Delft, voorzitter
Prof.dr. S. Vassiliadis	Technische Universiteit Delft, promotor
Prof.dr. J.E. Smith	University of Wisconsin-Madison
Prof.dr. M.J. Flynn	Stanford University
Prof.dr. P. Stenstrom	Chalmers University of Technology
Prof.dr. T.M. Conte	North Carolina State University
Prof.dr.ir. P. Van Mieghem	Technische Universiteit Delft
Prof.dr.ir. P.M. Dewilde	Technische Universiteit Delft
Prof.dr. C.I.M. Beenakker	Technische Universiteit Delft, reservelid

ISBN 90-9021060-1

The TM3270 Media-processor

Jan-Willem van de Waerdt

Abstract

In this thesis, we present the TM3270 VLIW media-processor, the latest of TriMedia processors, and describe the innovations with respect to its predecessor: the TM3260. We describe enhancements to the load/store unit design, such as a new data prefetching technique, and architectural enhancements, such as additions to the TriMedia Instruction Set Architecture (ISA). Examples of ISA enhancements include collapsed load operations, two-slot operations and H.264 specific CABAC decoding operations. All of the TM3270 innovations contribute to a common goal: a balanced processor design in terms of silicon area and power consumption, which enables audio and standard resolution video processing for both the connected and portable markets. To measure the speedup of the individual innovations of the TM3270 design, we evaluate processor performance on a set of complete video applications: motion estimation, MPEG2 encoding and temporal upconversion. Each of these applications have been optimized to take advantage of the TM3270 enhancements, and the associated speedups have been measured to evaluate the impact of e.g. load/store unit improvements and new operations. We show that load/store unit improvements, such as data prefetching, may improve the dynamic performance complexity (processor cycle count) by more than a factor two, for larger off-chip memory latencies. The speedup of individual ISA enhancements are measured in terms of both static (VLIW instruction count) and dynamic (processor cycle count) performance complexity, and both at the level of individual kernels and complete applications. Combined, the TM3270 enhancements result in speedups of more than a factor two, for the evaluated video applications.

Acknowledgements

This thesis describes the TriMedia TM3270 media-processor, as designed at Philips Semiconductors in San Jose, USA. Although the TM3270 design only spans the last two years (2004-2005), the knowledge build up to enable such an activity spans the last eight years of my working career. Processor design is not a one-person activity, but a collaborative effort of a team of people, and I would like to take this opportunity to thank them.

During my working career with Philips (first at Philips Research in Eindhoven, the Netherlands, and later at Philips Semiconductors in San Jose, USA), I had the opportunity to work with a series of individuals that contributed to my development as a processor designer. I would like to explicitly mention the following people: Pieter Struik, for the initial years at Philips Research and his efforts to familiarize me with real world programming languages and UNIX tools. Paul Stravers, for the introduction to processor architecture and implementation. Especially his explanation of the difference between a latch and a flip-flop is still greatly appreciated. It was also his advice to move to Silicon Valley that contributed to my further development. Gerrit Slavenburg, father of the TriMedia architecture, for forcing me to acquire knowledge in all domains of VLIW media-processor design: processor, compiler, application and real-time operating system design. Furthermore, his daily enthusiasm about our work and his drive for excellence has greatly improved the quality of the TM3270 design as described in this thesis. Stamatis Vassiliadis for his academic guidance in writing the TM3270 related conference papers and this thesis. Furthermore, I enjoyed the discussions we had on the history of processor design in general and his experiences at IBM building real world processors. I hope that this thesis reflects that processor design is more than academic architectural evaluations of imaginative machines.

As mentioned, processor design is a team effort. In the case of the TM3270 design, the processor design team collaborated with application teams and those that are responsible for the toolchain. I would like to thank those in the Philips Semiconductors application and toolchain teams for their help in defining the TM3270 processor architecture. It is your work that enables the success of this processor

in the market place. Although I can take credit for the TM3270 as chief architect and as a designer, I am thankful for the help of my colleagues in the processor design team: Dinesh Amirtharaj, Carlos Basto, Sanjeev Das, Jean-Paul van Itegem, Kulbhushan Kalra, Sebastian Mirolo, Pedro Rodrigues, Chris Yen and Bill Zhong. Without you guys, the TM3270 and its predecessors would not have seen the light of day, and would have died at a conceptual PowerPoint level.

I also thank Hans van Antwerpen, Jos van Eijndhoven and Jan Hoogerbrugge for their suggestions and the discussions we had through the years.

Jan-Willem van de Waerdt San Jose, USA, January 2006

Contents

1 Introduction **1**
1.1 Background and related work 2
1.1.1 Overview of media processing platforms 2
1.1.2 Strengths and weaknesses 5
1.1.3 Positioning media-processors and the TM3270 8
1.2 Main contributions 10
1.3 Overview of the thesis 11
1.3.1 Structure of the thesis 11
1.3.2 Performance evaluation environment 12

2 Architecture **15**
2.1 TM3260 overview 17
2.2 TM3270 overview 20
2.3 ISA enhancements 21
2.3.1 Non-aligned memory access 22
2.3.2 Multiplication with rounding 24
2.3.3 Two-slot operations 25
2.3.4 Collapsed load operations 27
2.3.5 CABAC operations 27
2.3.6 Potpourri 29
2.4 Instruction cache LRU update 30
2.5 Data prefetching 33
2.6 Conclusions 35

3 Implementation **37**
3.1 Processor pipeline 37
3.2 Instruction fetch unit 38
3.2.1 VLIW instruction encoding 38
3.2.2 Instruction fetch unit pipeline 42

3.3 Load/store unit . . . 47
3.3.1 Load/store unit pipeline . . . 47
3.3.2 Memory organization . . . 49
3.3.3 Memory arbitration . . . 52
3.3.4 Data prefetching . . . 54
3.4 Conclusions . . . 56

4 Realization **57**
4.1 CMOS realization . . . 57
4.2 Power consumption . . . 60
4.3 Performance . . . 64
4.3.1 MediaStone . . . 64
4.3.2 CABAC operations . . . 66
4.4 Conclusions . . . 68

5 Motion estimation **71**
5.1 Description of the algorithm . . . 72
5.1.1 The estimator . . . 72
5.1.2 Block-matching . . . 75
5.2 Block-matching implementations . . . 76
5.2.1 Traditional block-matching . . . 77
5.2.2 Down-sampled block-matching . . . 81
5.2.3 Static performance complexity . . . 82
5.3 Dynamic performance complexity . . . 84
5.3.1 Comparing the implementations . . . 85
5.3.2 Memory latency . . . 87
5.3.3 Data prefetching . . . 88
5.4 Conclusions . . . 89

6 MPEG2 encoder **93**
6.1 Description of the algorithm . . . 94
6.2 Motion estimator . . . 95
6.2.1 Macroblock matching . . . 96
6.2.2 The estimator . . . 97
6.3 Texture pipeline . . . 100
6.3.1 Difference calculation . . . 101
6.3.2 Discrete cosine transform . . . 103
6.3.3 Quantization . . . 104
6.3.4 Run length encoding . . . 106
6.3.5 Inverse quantization . . . 106

6.3.6 Inverse discrete cosine transform 107
6.3.7 Image reconstruction kernel 108
6.3.8 Putting it all together 109
6.4 Bitstream generation 111
6.5 Dynamic performance complexity 113
6.6 Conclusions 114

7 Temporal upconversion 117
7.1 Description of the algorithm 118
7.2 Implementation 122
7.2.1 Six implementations 122
7.2.2 Static performance complexity 124
7.3 Dynamic performance complexity 125
7.3.1 Comparing the implementations 127
7.3.2 Memory latency 128
7.3.3 Data prefetching 130
7.3.4 Write miss policy 132
7.4 Conclusions 133

8 Conclusions 135
8.1 Summary of conclusions 135
8.2 Main contributions 137
8.3 Further research 138

Bibliography 141

A New operations 149
A.1 Single slot operations 150
A.2 Two-slot operations 155
A.3 CABAC operations 162

Samenvatting 167

Biography 169

Chapter 1

Introduction

Processor design has made considerable progress in the last half century. Increased circuit density allows for both higher performance integrated circuits and cheaper computers built from fewer components (as indicated by by G. Moore in [40, 41]). Furthermore, the use of CMOS process technology allows for low power implementations of these components [25].

In the cost-driven embedded consumer market, audio and video processing were initially addressed with dedicated hardware. Dedicated hardware could deliver the required performance at a lower price point than programmable processors. However, the increased complexity of audio and video standards made programmability attractive, and the increased performance of application domain specific processors made programmability a possibility. E.g., whereas video standards such as MPEG2 were initially performed by dedicated hardware, today's video standards such as H.264/AVC are performed by application (domain) specific processors [66, 20, 19]. As a result, today's consumer devices have more programmable processing capabilities than the mainframes of the 1960s. Low power processor implementations enable the application in the portable, battery-operated domain, e.g. mobile phones.

This thesis describes the design of the TM3270 media-processor, the latest processor of Philips Semiconductors' TriMedia architecture family. The TM3270 is an application domain specific processor, targeting both video and audio processing. It is intended as a programmable media-processing platform for the embedded consumer market.

The remainder of this chapter is organized as follows. Section 1.1 gives an overview of related work and provides a taxonomy of media processing platforms. Section 1.2 lists the main contribution of the thesis to the field of programmable media-processors. Section 1.3 completes this chapter with an overview of the thesis

and a description of the performance evaluation environment as used in the later chapters.

1.1 Background and related work

A wide range of media processing platforms exists on which to implement video and audio processing. In this section we give an overview of these platforms, discuss their relative strengths and weaknesses and position media-processors in general and the TM3270 in particular.

1.1.1 Overview of media processing platforms

Figure 1.1 gives an overview of media processing platforms. General-Purpose Processors (GPPs), originally designed to accomodate generic program execution, have been extended with SIMD-style instructions to the instruction set architecture (ISA), to exploit intra-word parallelism (GPP+SIMD). E.g., Intel's x86 architecture family has been extended with MMX instructions [2] and IBM/Motorola's PowerPC architecture family has been extended with AltiVec instructions [12]. The SIMD-style extensions are typically generic media processing domain instructions, rather than application specific instructions targeting specific media kernels. These processors provide GPP functionality, e.g. a virtual memory management unit and user/protected modes for operating system support. However, media processing data movement support, such as non-aligned memory access and the streaming nature of media data types, is typically poorly addressed.

Multiple, mostly academic, approaches have been proposed to address the inefficient media processing data movement capabilities of the GPP+SIMD approach. These approaches extend a GPP with streaming vector capabilities (GPP+vector). The extension is tightly connected to the GPP: a single (vector) instruction sequence controls both the GPP and the vector unit. Typically, the vector unit has its own (vector) register-file, datapath and access path to data memory. The use of vector instructions, to exploit inter-word parallelism, and the support for strided memory access limits instruction fetch pressure, as many data movement instructions have become unnecessary. Examples of this approach are Motorola's reconfigurable streaming vector processor (RSVP) [7], the complex streamed instruction (CSI) set architecture from TU Delft [6] and the MediaBreeze architecture from Texas university [56]. The matrix oriented multimedia (MOM) approach from UPC in Barcelona [8] merges SIMD-style with vector-style extensions to create matrix-style instructions.

The efficiency of the GPP+vector approaches relies on a large amount of data level parallelism, regularity in memory accesses that can be expressed with stride

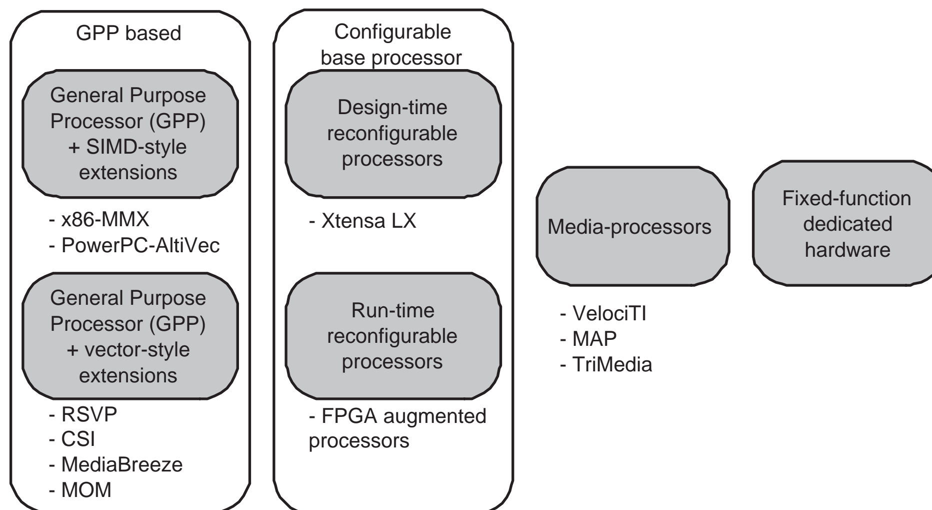

Figure 1.1: *Media processing platforms for video and audio processing.*

values and a stream-based processing of multimedia data. Whereas this may have been typical for older video codec standards, this assumption is less true for newer standards. As an example, consider the granularity at which video codecs use motion vector data. For MPEG2, a single motion vector is present for every 16x16 block of image pixels. For MPEG4, a motion vector may be present for every 8x8 block, and for H.264, a motion vector may be present for every 4x4 block. In general we can observe a decrease in block size and an increase in control overhead. Furthermore, the dependency between blocks is increasing, which limits the parallel processing of multiple blocks. E.g. for H.264, processing a 4x4 block may require that the blocks to its left and above it have already been processed. It could be stated that video codecs are getting more control intensive and offer less data level parallelism. As a result, approaches that solely rely on stream-based processing on large vectors of independent data elements become less efficient.

Rather than extending an established GPP architecture with fixed SIMD- or vector-style capabilities, processors can be extended with reconfigurable capabilities. The instruction set extensions of the GPP+SIMD and GPP+vector approaches are typically a common denominator of what is useful in the media processing domain, rather than application specific. Through reconfigurability, the extensions can be made a better fit in terms of cost and performance for a specific application. We distinguish two types of reconfigurability: *design-time* reconfigurability and *run-time* reconfigurability.

Design-time reconfigurable processors allow the user (in this case the recon-

figurable processor designer) to add application (domain) specific extensions to a base processor. An example of this approach is Tensilica's Xtensa LX configurable processor. A standard definition video decoder based on Tensilica's technology is described in [20]. To enable the required functionality, over 200 instructions were added to the ISA of the base processor. These new instructions are specific to the task of video decoding and as a result the processor's functionality is limited (additional operations need to be added to enable video encoding functionality).

Run-time reconfigurable processors allow the user (in this case the application designer) to add application (domain) specific extensions to a base processor. As opposed to design-time reconfigurability, these additions can be made after the processor has been created. In [51], a Philips Semiconductors' TriMedia processor is extended with a FPGA fabric, on which the user can implement specific instructions. In [70], the organization of a processor-FPGA hybrid design is described, including a description of the often neglected programming paradigm and compiler technology to address such a design. Other approaches limit the flexibility to a coarse grained reconfigurability, which may improve the cost efficiency of the approach. Run-time reconfigurable processors are like chameleons in the sense that they adapt their behavior to the application at hand. However, the cost of reconfigurability in terms of silicon area is likely to be higher than that of design-time reconfigurable processors.

In the 1990s, the need for an efficient programmable platform for video and audio processing led to the design of media-processors. These processors typically have a very long instruction word (VLIW) architecture to exploit instruction level parallelism [15], combined with SIMD-style instructions to exploit intra-word parallelism. Aggressive compiler technology combined with guarded execution of individual operations allows VLIW processors to extract more instruction level parallelism than traditional superscalar approaches [24, 21]. In [15], J.A. Fisher argues that through compiler techniques, such as trace scheduling, parallelism can be extracted from 100% of the code base for VLIW processors, whereas vector processing is typically applicable to only a fraction of the code base and requires hand optimization. Media-processors typically have a unified register-file for integer, floating point and SIMD-style operands, whereas GPPs typically have separate register-files. Furthermore, media-processor register-files tend to be larger than GPP register-files, such that a large data working set can be kept in registers, preventing the generation of load and store operations as a result of spilling due to register pressure. Media-processors typically support non-aligned memory access and the streaming nature of media data types through either data prefetching techniques or direct memory access (DMA) techniques [29]. Examples of media-processors are Texas Instruments' VelociTI architecture family [47], Equator's MAP architecture family [1] and Philips Semiconductors' TriMedia architecture

family [43].

The last approach to be discussed in this overview is a fixed function dedicated hardware platform. This approach provides an application specific solution, without any flexibility in terms of programmability[1], potentially at a lower price point than programmable approaches in terms of silicon area and power consumption. Dedicated hardware may be attractive to implement a well-defined video or audio processing task that has no need for flexibility.

One might argue that our partitioning of media processing platforms into five distinct approaches is somewhat artificial and indeed the best solution for a specific application (domain) may be a combination of approaches.

1.1.2 Strengths and weaknesses

Whereas the previous section listed media processing platforms on which to implement video and audio processing, this section compares these platforms along the following axes:

- *Application domain.* This axis represents the width of the application domain that can be addressed with the platform.
- *Cost.* Cost is an important factor in the cost-driven embedded consumer market, and is a multi-facetted axis. We distinguish the development cost of the solution, the silicon area of the solution and the power consumption of the solution. Especially in the portable battery-operated market, power consumption is an important factor.
- *Infrastructure.* This axis takes on different forms for the different platforms. For the GPP-based and media-processor approaches it includes aspects such as the availability of toolchains (compiler, debugger), operating systems, off-the-shelf video codecs, etc. For reconfigurable processors it includes the processor development environment as offered by the reconfigurable processor company. For a dedicated hardware platform it includes the computer aided development environment.
- *Performance.* This axis represents the performance level that can be achieved with the approach. Performance level is measured in context of the target application. As an example, for video decoding the performance level can be expressed in terms of image resolution (e.g. CIF, standard definition, high definition).

[1]The dedicated hardware may have control/status registers to direct/observe its behavior from an external processor.

- *Time-to-market.* This axis expresses the speed with which an application can be introduced to the market place.

A comparison along these axes is by no means complete, but does give us the opportunity to highlight the relative strengths and weaknesses of the platforms. A choice for a specific platform will depend on the importance of the different axes for the application (domain) at hand; it is unlikely that a single approach is the best fit for all applications. The answer to the question *"What is the best approach for a certain application (domain)?"* is in the end defined by the success of the approach in the market place. Table 1.1 gives a summary of our perspective on the relative strengths and weaknesses, on which we elaborate in the following. Similar evaluations of media processing platforms can be found in [9, 48].

Axis	*GPP+SIMD*	*GPP+vector*	*Design-time reconf.*	*Run-time reconf.*	*Media-processor*	*Dedicated hardw.*
Application domain	++	++	-	+	+	−−
Cost - development	++	?	-	-	+	−−
Cost - silicon area	−−	-	+	-	-/+	++
Cost - power consumption	−−	−−	+	-/+	-/+	++
Infrastructure	++	?	-	-	+	−−
Performance	−−/+	−−/+	+	+	-/+	++
Time-to-market	++	+	-	+	++	−−

Table 1.1: *Relative strengths and weaknesses of media processing platforms. '/' Indicates a range and '?' indicates a lack of data, as the solution has not yet been applied in the market.*

Application domain. As GGPs were originally designed for general-purpose processing, it should come as no surprise that the GPP based approaches cover the widest application domain. Dedicated hardware typically offers an application specific solution. Similarly, design-time reconfigurable solutions are typically application (domain) specific (the instruction set extensions are limited to a certain application (domain)). Run-time reconfigurable solutions allow for customization of the instruction set, to adapt to new applications. Media-processors typically target the full range of video and audio processing, but are fixed after design, just like design-time reconfigurable solutions. However, their high instruction issue rate may offer enough raw computational performance to address new applications,

whereas design-time reconfigurable solutions typically have limited computational performance in application domains that are outside their original design scope.

Cost. In terms of development costs, the GPP+SIMD approach is attractive as a result of its wide application domain. Because of the market success of GPPs, it is highly likely that the required platform, including application software, exists for the target application (domain). This argument holds true, but to a lesser degree, for the media-processor approach. In terms of silicon area, dedicated hardware is to be preferred over the other approaches; unnecessary area overhead related to processor design can be eliminated for a fixed-function implementation. The GPP+SIMD approach tends to be larger than the other programmable approaches at a similar performance level, as the ISA is less specialized to a specific application domain and their media processing data movement support is limited. In terms of power consumption, the smaller approaches (in terms of silicon area) that perform the target application at a low operating frequency are preferable.

Infrastructure. Standardization is partly responsible for the market success of GPPs and media-processors. As a result, a wide range of compilers, operating systems and off-the-shelf codecs are available for these platforms, either from the processor provider or from third-party software providers. For reconfigurable processors, the user depends on the infrastructure of the reconfigurable processor provider. As these providers are not charitable institutions, the quality of their infrastructure most likely depends on the success of their solution in the market place; a reconfigurable processor company with a high-quality (costly) infrastructure but with a limited customer base is not a sustainable business in the long run. The dedicated hardware approach typically requires a "do it yourself" way of working.

Performance. Performance should be adequate to address the application (domain) at hand. Currently, this excludes some approaches for certain applications. E.g., it is unlikely that any of the programmable approaches can address the performance requirements of a high definition H.264 video encoder[2]. The GPP based approaches cover a relatively wide performance range: low cost solutions with limited performance and high cost solutions with more performance.

Time-to-market. The ideal solution in terms of silicon area and power consumption may be useless when it is late to the market. New markets may be best addressed with an acceptable solution that is readily available; to be later replaced by an area and power optimized solution. Time-to-market is heavily related to the width of the application domain and the quality of the infrastructure

[2]Advances in processor design continuously improve performance levels, however, whenever a certain application becomes within reach, it is likely that a new application with higher performance requirements is introduced.

of the approach.

The relative strengths and weaknesses of the different approaches suggest that the best solution for a specific application (domain) may be a combination of approaches. As an example, consider a high definition H.264 video encoder application. Performance complexity may prohibit a programmable approach, but the standard's functional complexity may prohibit the development of a bug-free dedicated hardware implementation in a reasonable time frame. In this particular example, a combination of a programmable approach with dedicated hardware co-processors may be a viable solution. The programmable component addresses the standard's functional complexity and the co-processors address the performance complexity of the standard's media kernels.

1.1.3 Positioning media-processors and the TM3270

The TriMedia TM3270 processor is a media-processor targeting video and audio processing. Its positioning is a result of the strengths and weaknesses of the media-processor approach, as repeated in Table 1.2. The width of its application domain allows for the implementation of video, audio and general-purpose processing tasks. Its power consumption is acceptable to allow for application in the portable battery-operated market. This combination of a wide application domain and the ability to address both connected and portable markets positions the TM3270 as a standard media-processor that can rely on both Philips Semiconductors internal and external software suppliers.

Axis		*Rationale*
Application domain	+	Covers full range of video and audio processing.
Cost - development	+	Hardware platform readily available, software potentially available from provider or third-party software suppliers.
Cost - silicon area	-/+	Specialization for media processing (in terms of generic media operations and data movement).
Cost - power cons.	-/+	Specialization for media processing makes it more efficient than a GPP+SIMD approach, but less efficient than more application specific approaches.
Infrastructure	+	Standardization resulted in support in terms of compilers, operating systems, codecs, etc.
Performance	-/+	Acceptable for audio and standard definition video processing. High definition video processing may be out of reach.
Time-to-market	++	Hardware platform and potentially software available.

Table 1.2: *Relative strengths and weaknesses of the TM3270 media-processor.*

Figure 1.2 illustrates the advantage of a wide application domain. In 3G mobile

phones, the TM3270 may be used implement video telephony, performing both the video and audio processing. In the *decoding chain*, the TM3270 is used to demultiplex an incoming bitstream and to decode video and audio. The video decoding path starts with a H.264 decode (e.g. QCIF or CIF resolution at 15 frames/sec.). Next, a motion estimation algorithm is performed to identify object movement in the video stream. This motion information is used by a motion-compensated temporal up-converter, which adapts the frame rate of the incoming video (15 frames/sec.) to that of the phone display (e.g. 60 frames/sec.). In a last step, a spatial up-converter and image enhancement algorithm are performed to adapt the resolution of the incoming stream (QCIF or CIF) to that of the display (e.g. 640x400). The audio decoding path consists of an audio decode, which may be extended with a post-processing algorithm to enhance the sound quality. At the same time, the TM3270 is used to perform similar functionality in the *encoding chain*.

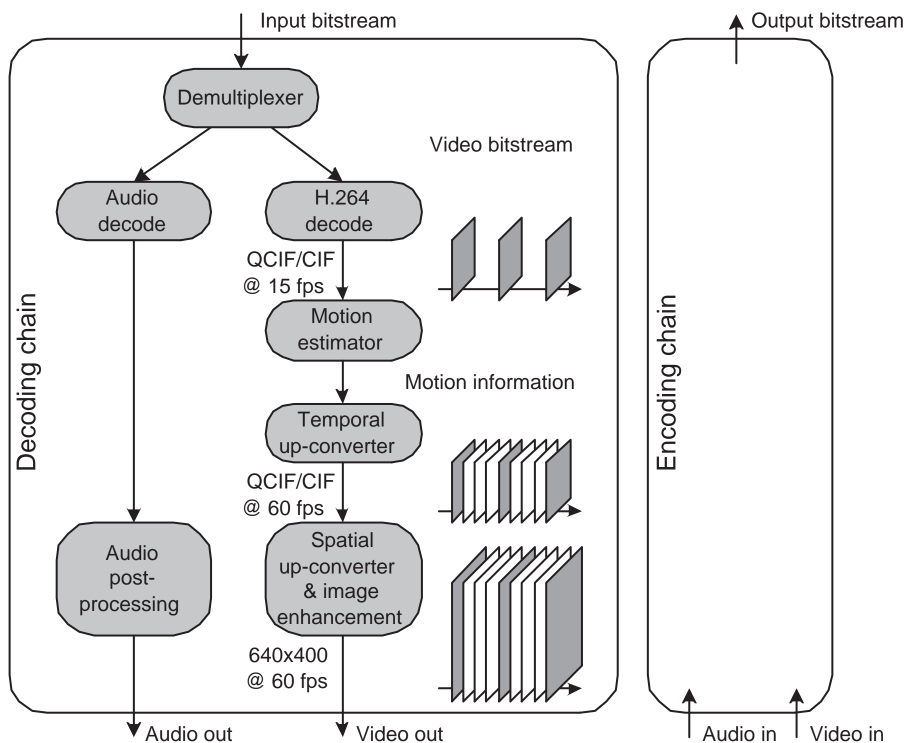

Figure 1.2: *A possible use of the TM3270 in a 3G mobile phone.*

The previous example gives *an impression* of the possibilities of the TM3270

media-processor. Other mobile phone design constraints such as the use of an established GPP (such as the ARM processor), power consumption and compartilization of functionality will most likely result in a partitioning of tasks over multiple processors. When considered in isolation, each of the tasks performed by the TM3270 is most likely more efficiently performed by dedicated hardware or application specific processors. However, the ability to time-share the TM3270 processor for multiple tasks make it an interesting platform in terms of silicon area and offers flexibility to address future standards [50].

1.2 Main contributions

As will be described in the remainder of this section, the TM3270 media-processor design has a series of innovations that distinguish it from other media-processors. These innovations are driven by the need to address both the connected and portable markets, the need for a balanced design in terms of silicon area and performance level and the requirements of the latest video processing algorithms.

The TM3270 design provides *enough performance to address the requirements of standard and some high definition video processing algorithms* in the connected market, such as high-end TV sets. At the same time, its *low power consumption* enables successful application in portable battery-operated markets. The processor's pipeline partitioning and the design of individual units, such as the instruction fetch unit and the load/store unit, are a result of a *trade-off between performance, power and silicon area.*

The instruction fetch unit implements a sequential instruction cache design to limit power consumption and supports *a cache line replacement policy that prevents cache trashing as a result of code sequences with limited temporal locality.*

The load/store unit design provides high performance through a *semi multi-ported cache*, providing high data bandwidth to the data cache, at a limited area penalty when compared to a single-ported cache. The cache sustains a high store bandwidth by allowing two operations per VLIW instruction and a high load bandwidth by sustaining a single load operation per VLIW instruction with a bandwidth of twice the datapath size. All load and store operations support *non-aligned memory access, without incurring any processor stall cycles.* To our knowledge, the particular implementation of the data cache is unprecedented. Furthermore, *a new data prefetching technique* is introduced. From an architectural perspective the technique provides limited overhead to the programmer and from an implementation perspective it adds limited overhead to the design in terms of silicon area.

The TM3270 extends the TriMedia ISA with a series of new operations. *Col-*

lapsed load operations combine the functionality of a traditional load operation with that of a 2-taps filter function. These operations are particular useful to perform horizontal fractional pixel calculations in video processing algorithms. The TM3270 is the first processor in the market to support *two-slot operations*, which were introduced in [68]. These operations are executed by functional units that are situated in two neighboring VLIW issue slots, and as a result have twice the register-file bandwidth: two-slot operations may consume up to *four* 32-bit sources, and produce up to *two* 32-bit results. *CABAC decoding operations* address the specific requirements of the H.264 standard's Context-based Adaptive Binary Arithmetic Coding (CABAC) decoding process. These new operations allow the TM3270 to decode a standard definition H.264 video bitstream in real-time.

The individual innovations all contribute to a common goal: a balanced processor design in terms of silicon area and power consumption, which enables audio and standard resolution video processing for both the connected and portable markets.

1.3 Overview of the thesis

The thesis can be roughly divided in three parts. The first part covers Chapters 2, 3 and 4, and describes the TM3270 media-processor design. The second part covers Chapters 5, 6 and 7, each of these chapters presents the performance evaluation of a different video application. The third part is Appendix A, which defines some of the new TM3270 operations. Section 1.3.1 briefly discusses the content of the individual chapters. Section 1.3.2 describes the performance evaluation environment that is used in Chapters 5, 6 and 7.

1.3.1 Structure of the thesis

In Chapter 2 we describe the architecture of the TM3270 TriMedia media-processor. We start with the main design targets of the TM3270 and give an overview of its predecessor: the TM3260. Next, we describe those architectural functions that the TM3270 processor adds with respect to the TM3260. These additions include ISA extensions, a new cache line replacement algorithm for the instruction cache and a new data prefetching technique for the load/store unit.

Chapter 3 describes the implementation of the TM3270. Again, the focus is on those aspects of processor design that distinguish the TM3270 from its predecessor. We describe the processor pipeline and give an overview of the units that make up the processor implementation. The instruction fetch unit and load/store unit are discussed in greater detail.

Chapter 4 describes the realization of the TM3270 in a low power CMOS process technology, with a 90 nm feature size. In particular, we describe the physical

realization (with floorplan and area data) and power consumption. Furthermore, we present performance data that compares the TM3270 to the TM3260 on a series of video processing algorithms and kernels. We also present performance data of a standard definition H.264 video decoder, and quantify the speedup of the new CABAC decoding operations (as described in Chapter 2).

Chapter 5 evaluates the TM3270 performance on a motion estimator. Motion estimation has multiple applications; e.g. it is part of video encoders such as the MPEG2 encoder (as described in Chapter 6) and it is a prerequisite of motion-compensated temporal upconverter algorithms (as described in Chapter 7). We describe different implementations of a motion estimation algorithm that take advantage of TM3270 enhancements to the TriMedia architecture, such as new operations and non-aligned memory access. We evaluate the static performance complexity of these implementations to determine the speedup of the individual enhancements. Furthermore, we measure the dynamic performance complexity of these implementations to determine the effect of the new data prefetching technique and the sensitivity of processor performance to off-chip memory latency.

Chapter 6 evaluates the TM3270 performance on a MPEG2 video encoder. We describe how new TM3270 operations are used to speedup the individual kernels of the MPEG2 encoder texture pipeline. Furthermore, we discuss the dynamic performance complexity of the complete MPEG2 encoder, including an analysis of the sensitivity of processor performance to off-chip memory latency.

Chapter 7 evaluates the TM3270 performance on a motion-compensated temporal upconverter. We describe different implementations of the algorithm that take advantage of TM3270 enhancements to the TriMedia architecture, such as new operations and non-aligned memory access. We evaluate the static performance complexity of these implementations to determine the speedup of the individual enhancements. Furthermore, we measure the dynamic performance complexity of these implementations to determine the effect of the new data prefetching technique, data cache write miss policy and the sensitivity of processor performance to off-chip memory latency.

Finally, Chapter 8 concludes the thesis, summarizing our main contributions and findings, and proposing areas for further research.

Appendix A describes in detail the new TM3270 operations.

1.3.2 Performance evaluation environment

To evaluate the dynamic performance complexity (including processor stall cycles as a result of cache misses) of the video applications in Chapters 5, 6 and 7, a performance evaluation environment is used that represents realistic System-on-Chip processor behavior. Most of today's SoCs in the embedded consumer

market have a unified memory architecture; i.e. the off-chip SDRAM memory is shared between the TM3270 and other on-chip devices to reduce cost. The environment consists of the TM3270 media-processor operating at 450 MHz, a 32-bit DDR SDRAM controller operating at 200 MHz and a delay block in the on-chip interconnect structure (Figure 1.3).

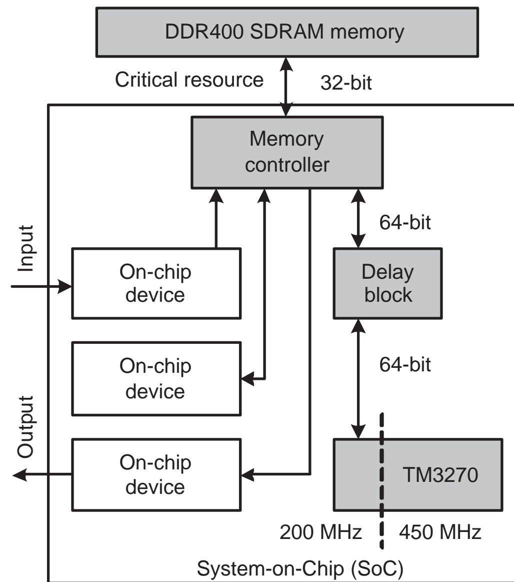

Figure 1.3: *Performance evaluation environment.*

The TM3270 Verilog HDL model is used for simulation to guarantee a 100% accurate representation of processor and cache behavior. The processor has an asynchronous clock domain transfer, which allows for independent processor and memory clock frequencies (the performance evaluations use a 450 MHz processor clock and a 200 MHz memory clock). The on-chip memory interconnect has a datapath width of 64-bit and the off-chip DDR SDRAM interconnect has a datapath width of 32-bit[3]. Typically, the off-chip SDRAM bandwidth is a critical resource (if it is not, cost reduction is possible by limiting the amount of SDRAM data pins or by using lower frequency (lower cost) SDRAM memories). As other on-chip devices consume more SDRAM bandwidth, the SDRAM latency as observed by the processor increases and so does the processor cache miss penalty. To mimic this behavior, a delay block is used to artificially delay memory traffic between the TM3270 and the off-chip SDRAM. By changing the delay, we can measure

[3]The off-chip SDRAM interconnect operates at a double data rate, effectively doubling the 32-bit data transmission frequency to 400 MHz.

the dependency of processor performance on off-chip SDRAM latency. The delay block is located in the 200 MHz memory clock domain: one delay cycle represents 2.25 processor delay cycles.

Chapter 2

Architecture

The TM3270 is a member of the Philips Semiconductors' TriMedia architecture family of media-processors [43, 46]. The architecture finds its origin in the LIFE research project, which was executed at Philips Research in Palo Alto [32, 31]. To avoid misunderstandings in this thesis, we assume the terminology and definitions of [3] as related to computer architecture, implementation and realization:

> *Architecture* concerns the specification of the function that is provided to the programmer, such as addressing, addition, interruption, and input/output (I/O). *Implementation* concerns the method that is used to achieve this function, such as parallel datapath and microprogrammed control. *Realization* concerns the means used to materialize this method, such as electrical, magnetic, mechanical and optical devices and the powering and packaging for them. *(G.A. Blaauw and F.P. Brooks jr., "Computer Architecture, Concepts and Evolution").*

In this chapter, we describe the TM3270 architecture. Obviously, an architecture can have multiple implementations, and an implementation can have multiple realizations. Besides, processor architectures change over time, e.g. their most prominent function, the *Instruction Set Architecture (ISA)*, evolves to include new instructions that were deemed necessary by the architect to better address the processor application domain. Typically, architectural changes are incremental to ensure backward compatibility; i.e. programs that run on an older architecture will also run on a newer architecture of the same processor family. In the case of the TriMedia architecture family, compatibility is defined at program source-level, rather than binary-level. The transformation of program source code into binary code is assumed to be performed by a compiler/scheduler, rather than by the pro-

grammer directly. This provides us with additional freedom when implementing the processor architecture. In the spirit of Blaauw's definition of architecture, this excludes the set of compiler writers from the set of programmers; programmers work with the architecture, whereas the VLIW compiler writer requires knowledge of the architecture's implementation. The evolution of the TriMedia architecture family is illustrated in Figure 2.1. Four separate architecture levels and their processor members are identified; with each level a super-set of a lower level.

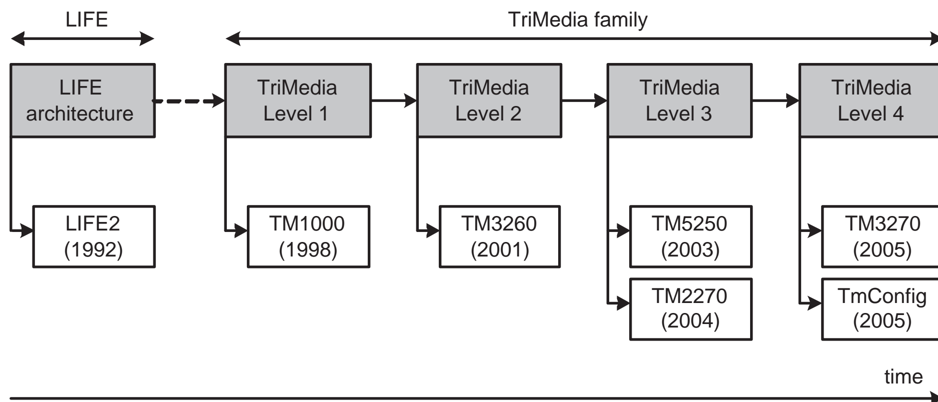

Figure 2.1: *Evolution of the TriMedia architecture family.*

In general, the TriMedia architecture family targets the multimedia application domain. Originally, this domain included the 3D graphics processing subdomain [14]. However, the increased computational demands of this domain has led to the design of specific graphics processors [39]. Therefore, TriMedia processors have focused on the video and audio processing subdomains. In particular, the TM3270 media-processor was designed with the following main targets (all of which impact processor architecture, implementation and realization):

- *Application domain.* In line with the TriMedia architecture family, the TM3270 targets video and audio processing. With video processing being the most computationally demanding, most of the design choices are made to address video processing requirements. A specific video requirement is the ability to perform main profile H.264 decoding at main level [44] at a sustained bitrate of 2.5 Mbits/s with a maximum dynamic performance complexity of 300 MHz. This performance requirement sets a lower bound on the processor's acceptable performance level.

- *Area.* Consumer markets, such as the portable battery-operated, set-top box and television markets, require a low-cost realization in terms of silicon area.

- *Power.* The portable battery-operated market requires low power consumption to allow for longer "playing time". Indirectly, power consumption is related to cost, e.g. high power consumption may require more expensive IC packages or active cooling techniques.

- *Synthesizable.* To allow for cheap and fast migration of a processor implementation from one CMOS process technology to another, a synthesizable processor design is preferred.

This chapter describes the architecture of the TM3270. We found that a clean separation between architectural function and implementation method is hard to maintain. For embedded processors, implementation aspects related to performance efficiency and cache design are sometimes found to be exposed to the programmer at the architectural level. As a result, the description of some architectural functions will be related to the specific implementation method. We focus on those architectural functions that the TM3270 processor adds with respect to its predecessor: the TM3260. Section 2.1 gives an overview of the TM3260 architecture. For a complete description of the TM3260 architecture, the reader is referred to [46]. An overview of the TM3270 architecture, implementation and realization was earlier published as [66]. Section 2.2 gives an overview of the TM3270 architecture. Section 2.3 describes TM3270 ISA additions to the TriMedia architecture. Section 2.4 describes the new cache line replacement algorithm for the instruction cache. Section 2.5 describes the new data prefetching approach. Finally, Section 2.6 presents a summary and some conclusions.

2.1 TM3260 overview

The TM3260 processor is the first TriMedia processor that has a fully synthesizable design. It is *binary-level* backward compatible with its predecessor, the TM1000, and adds 13 new operations to the ISA. Like all other TriMedia processors it is a five-issue very long instruction word (VLIW) processor. It supports a 32-bit address space (4 Gbyte of addressable memory), and has a 32-bit datapath. The processor has 128 32-bit general-purpose registers, $r0$, ..., $r127$, organized in a unified register-file structure. Register $r0$ always contains the integer value 0, register $r1$ always contains the integer value 1. The TM3260 issues one VLIW instruction every cycle. Each instruction may contain up to five operations and each of the operations may be guarded; i.e. their execution can be made conditional based on the value of the least significant bit of the operation's guard register. This allows the compiler/scheduler to perform aggressive speculation/predication to exploit parallelism in the source code [24], to achieve high processor performance.

All of the general-purpose registers can be used as guard register. The following gives an example of a VLIW instruction with five operations (operations in issue slots 1, 3, and 4 are guarded):

```
IF r7   IADD      r4 r5 -> r8,         // issue slot 1
        UIMM      0x12345678 -> r13,   // issue slot 2
IF r10  FMUL      r21 r22 -> r23,      // issue slot 3
IF r30  STD32D(0) r31 r32,             // issue slot 4
        LD32D(0)  r41  -> r43;         // issue slot 5
```

Each operation has a fixed latency in terms of VLIW instructions, which is known by the compiler/scheduler at compile time. For example, the IADD operation has a latency of 1 instruction, thus the result of the IADD operation may be used as a source operand to an operation in the next VLIW instruction. In general, the result of an operation with latency i issued in VLIW instruction j may be used as a source operand to an operation in VLIW instruction j+i. Conditional and unconditional jump operations have 3 delay slots; i.e. when a jump is taken in VLIW instruction j, the operations in the next three sequential VLIW instructions j+1, j+2 and j+3 are executed.

Operations are executed by functional units and certain restrictions exist in how operations can be packed into a VLIW instruction. For example, load operations are executed by the load/store unit, which is only available in issue slots 4 and 5. An overview of the available functional units, their latency, and example operations is given in Table 2.1. Furthermore, no more than five results (of previously issued operations) can be written to the register-file in the same cycle. Typically, the packing of operations into VLIW instructions is not done by the programmer, but by the scheduler, which takes care of the mentioned operation restrictions.

Unlike traditional processor architectures, the TriMedia architecture only allows for special event handling (such as interrupts and exceptions) during interruptible jump operations, e.g. IJMPI. To support this event handling model, most operations either do not generate exception conditions, e.g. IADD, or set silent exception flags, e.g. FMUL or FADD. Silent exception flags and pending interrupt flags are only considered during interruptible jump operations. The limitation of special event handling to specific points in the scheduled code has several advantages. Firstly, the compiler/scheduler limits the use of general-purpose registers at these points, such that less architectural state needs to be saved before special event handling can commence. Secondly, the compiler/scheduler may perform more aggressive speculative scheduling, by ignoring the silent exception flags related to wrongly speculated operations [36].

In addition to the general-purpose registers, there are special purpose registers: *PC* (Program Counter), *PCSW* (Program Control and Status Word), *DPC* (Des-

Name	*Latency*	*Issue slots*					*Example operations*
CONST	1	1	2	3	4	5	IIMM, ...
ALU	1	1	2	3	4	5	IADD, ISUB, ...
SHIFTER	1	1	2	3	4	5	ASL, ASR, ROL, ...
JUMP	3		2	3	4		JMPI, JMPT, IJMPI, IJMPT, ...
DSPALU	2	1		3		5	DSPIDUALADD, ...
IMUL	3		2	3			UMUL, UMULM, DUALIMUL, ...
FALU	3	1			4		FADD, FSUB, ...
FMUL	3		2	3			FMUL, ...
FCMP	1		2	3			FGTR, FGEQ, FEQL, ...
FTOUGH	17		2				FDIV, FSQRT, ...
LS	3				4	5	ST32D, LD32D, ...
LS_SPECIAL	-					5	DINVALID, PREF, DCB, ...

Table 2.1: *TM3260 functional units. All functional units, except for the FTOUGH unit, are fully pipelined. The floating-point units FALU, FMUL, FCMP and FTOUGH are single precision IEEE-754 compliant.*

tination Program Counter), *SPC* (Source Program Counter) and *CCCOUNT* (Clock Cycle Counter). The *PC* register gives the program counter of the VLIW instruction that is currently issued by the processor. The *PCSW* register is a selection of control fields (e.g. endianness and floating point rounding) and status fields (e.g. silent exception flags). The *DPC* and *SPC* registers are related to special event handling. The *DPC* register is updated during every taken interruptible jump, with the target address of the jump operation. The *SPC* register is updated during every taken interruptible jump that is not interrupted by a special event handler. The handler uses *SPC* to determine the start of a VLIW instruction sequence in which the special event occurred and uses *DPC* as return address to resume the program, after the special event has been taken care of by the handler. Special event handling is supported by a dedicated exception vector address *EXCVEC* and 64 separate interrupt vector addresses *INTVECn* (n = 0, 1, ..., 63). The *CCCOUNT* register is a 64-bit counter, which can be set to increment on either every issued VLIW instruction or every processor cycle. Furthermore, the TM3260 processor includes a series of peripherals, most notably the four 32-bit timers. An overview of the TM3260 architectural state is given in Figure 2.2.

The TM3260 has a 64 Kbyte instruction cache (8-way set-associativity, 64 byte line size and a hierarchical least-recently used replacement algorithm) and a 16 Kbyte data cache (8-way set-associativity, 64 byte line size and a hierarchical least-recently used replacement algorithm). The data cache is dual-ported; a single VLIW instruction may contain two load operations, two store operations, or one load and one store operation. The TM3260 does not support non-aligned memory access. The TM3260 does not support hardware memory coherency. It is the

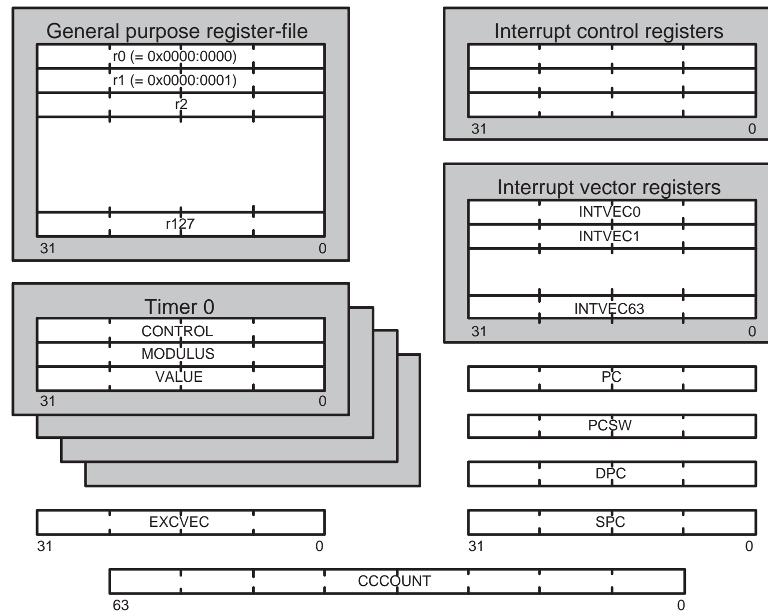

Figure 2.2: *Architectural state overview of the TM3260.*

responsibility of the programmer to use dedicated cache coherency operations, e.g. DINVALID (invalidate a cache line), DCB (victimize a dirty cache line), etc., to ensure SoC level coherency.

2.2 TM3270 overview

The TM3270 processor is the most recent TriMedia processor. It has a fully synthesizable design and is *source-level* backward compatible with its predecessors. It adds several architectural features to the TriMedia architecture. This section gives an overview of the TM3270 functional units and their latencies. The following sections describe other differences and additions in greater detail. To accomodate the need for speed, the TM3270 is deeper pipelined than the TM3260. This is reflected by the longer latencies of some of the functional units and the increased jump latency (Table 2.2). E.g., multiplication and load operations have a four cycle latency, whereas the TM3260 performed these operations with a three cycle latency. Although deeper pipelining has a negative impact on the cycles / VLIW instructions ratio (CPI), it allows for a higher frequency design.

The TM3270 has a 64 Kbyte instruction cache (8-way set-associativity, 128 byte line size and a LRU replacement algorithm). To improve the instruction cache hit

Name	*Latency*	*Issue slots*					*Example operations*
CONST	1	1	2	3	4	5	IIMM, ...
ALU	1	1	2	3	4	5	IADD, ISUB, ...
SHIFTER	1	1	2	3	4	5	ASL, ASR, ROL, ...
JUMP	5		2		4		JMPI, JMPT, IJMPI, IJMPT, ...
DSPALU	2	1		3	4		DSPIDUALADD, ...
IMUL	4		2	3			UMUL, UMULM, DUALIMUL, ...
FALU	4	1			4		FADD, FSUB, ...
FMUL	4		2	3			FMUL, ...
FCMP	2		2	3			FGTR, FGEQ, FEQL, ...
FTOUGH	17		2				FDIV, FSQRT, ...
LS_ST	-				4	5	ST32D, ...
LS_LD	3					5	LD32D, ...
LS_SPECIAL	-					5	DINVALID, PREF, DCB, ...
LS_FRAC	6					5	LD_FRAC8, ...
SUPER_ALU	1	1 + 2		3 + 4			SUPER_PACKMSBYTES, ...
SUPER_DSPALU	2	1 + 2		3 + 4			SUPER_DUALIMEDIAN, ...
SUPER_IMUL	4		2 + 3				SUPER_IFIR16, ...
SUPER_CABAC	4		2 + 3				SUPER_CABAC_STR, ...
SUPER_LS_LD	4				4 + 5		SUPER_LD32D, ...

Table 2.2: *TM3270 functional units.*

rate, the TM3270 has a full LRU replacement scheme, rather than the TM3260 hierarchical LRU scheme. The TM3270 has a 128 Kbyte data cache (4-way set-associativity, 128 byte line size and a LRU replacement algorithm). The data cache is *pseudo* dual-ported, a single VLIW instruction may contain one load operations, two store operations, or one load and one store operation. The TM3270 supports non-aligned memory access. The data cache size is increased from 16 Kbyte for the TM3260 to 128 Kbyte for the TM3270. The increased capacity is able to capture the data working set of most video algorithms operating at a standard definition (SD) resolution (NTSC: 720*480, PAL: 720*576). To limit the area impact associated with the increased cache size, the TM3270 data cache is pseudo dual-ported, rather than the TM3260 fully dual-ported data cache, which allows for a more area efficient implementation.

2.3 ISA enhancements

The TM3270 targets video and audio processing. Given the higher computational requirements of video processing, most new operations find application in this domain. Although we investigated new operations addressing the video application domain, we do not want to end up with an ISA that is application specific; i.e. we do not intend to build an application specific processor. While identifying

potentially interesting operations, we applied certain selection rules:

- Fits the processor architecture. It is probably subjective to assign a precise meaning to the word "fits". However, this rule reflects our intention to keep the architecture as clean as possible. The ISA should have a certain consistency. We identified the following restrictions to ensure consistency:
  - No operations with architectural state. This excludes e.g. the use of multipliers with accumulator values that are not transferred through operation operands.
  - Operations are limited to up to two issue slots.
  - Operations should support guarding.
  - For SIMD operations, the operands are partitioned into sub-operand fields. The sub-operand sizes and semantics should preferably be the same.
- Reuse of available processor resources. New operations typically add functionality to the existing datapath. It is the intent to restrict the additional silicon area to a minimum, to allow for a low-cost implementation of the architecture.
- Applicability in multiple domains. This reduces the risk that one ends up with operations that provide a solution within the scope of a specific kernel, but have no applicability outside this scope.
- Significant performance enhancement. New operations should improve performance. Performance improvement should be measured at the application level, rather than the kernel level; i.e. the contribution of a kernel to the application should be taken into account.

It is the interplay of the selection rules that decides whether a new operation is useful or not; i.e. an operation that adds a significant amount of silicon area may be justified due to wide applicability and significant performance enhancement of the operation. An expert in the areas of processor architecture and video processing should preferably judge the ISA enhancement as obvious. Having set the stage for the selection of operations, some of the new operations are described in the following sections.

2.3.1 Non-aligned memory access

The TM3270 supports non-aligned memory access for load and store operations that target the processor's memory aperture. The TM3260 does not support this

feature: least significant bits of a memory access address A[31:0] (bits 1 and 0 for 32-bit accesses, and bit 0 for 16-bit accesses) are discarded and the memory access is performed as if these bits were '0'. In case of address miss-alignment (A[1:0] $\neq$ "00" for 32-bit accesses, and A[0] $\neq$ '0' for 16-bit accesses) a silent exception status bit is set. Traditional control processor architectures, such as the MIPS architecture [28, 54], typically generate an address miss-alignment exception under this condition.

The MIPS architecture supports non-aligned memory access in software with dedicated operation, such as the LWL and LWR operations to retrieve 32-bit data elements. A non-aligned 32-bit data element is retrieved by a pair of these operations, LWL retrieves the byte elements left of the 32-bit address boundary, and LWR retrieves the byte elements right of the 32-bit address and merges these bytes with the LWL retrieved bytes. Texas Instruments' VelociTI media-processor architecture [47] supports non-aligned memory access in hardware. Normal load and store operations are used to access miss-aligned data elements. This approach eliminates the need for dedicated operations and when compared to the MIPS approach eliminates an operation to access a miss-aligned data element. However, a miss-aligned memory access may incur a stall cycle, which has a negative impact on processor performance. Similar to the VelociTI architecture, the TM3270 supports non-aligned memory access in hardware with normal load and store operations. The TM3270 *does not incur any stall cycles* for a miss-aligned memory access.

SIMD processing partitions operation operands into multiple sub-operands that are operated upon in a similar manner. Non-aligned memory access efficiently extends traditional SIMD computational processing, e.g. QUADADD or DUALMUL, to the memory access domain. Without non-aligned memory access, the potential gain of SIMD computational processing may be lost when the sub-operands that are operated upon cannot be efficiently accessed in memory. Consider a four-way 8-bit SIMD addition QUADADD of two operands, one located in processor register *r2* and the other located in memory at address A. Without non-aligned memory access support, the code sequence looks like:

```
alignment = A & 3;
A         &= 0xffff:fffc;                     // force alignment
temp1     = Mem[A];                           // aligned memory access
temp2     = Mem[A+4];                         // aligned memory access
temp      =   (temp1 << (alignment * 8))      // merge byte elements
            | (temp2 >> (32 - (alignment * 8)));
result    = QUADADD (r2, temp);
```

Whereas, with non-aligned memory access support, the code sequence looks like:

```
result = QUADADD (r2, Mem[A]);                // possible miss-aligned memory access
```

Non-aligned memory access has the obvious advantage of using fewer operations, which improves processor performance. As a result of using less operations, non-aligned memory access reduces the code size and when code size is at the boundary of instruction cache capacity, a small reduction may result in a significant performance gain due to the elimination of instruction fetch unit stall cycles as a result of cache misses. Another side effect of using fewer operations is a potential reduction in register-file pressure. For code with a large amount of parallelism, high register-file pressure may result in spilling of register operands. Spill code adds additional operations to move operands between the register-file and memory, potentially degrading processor performance [73].

2.3.2 Multiplication with rounding

The TriMedia architecture multiplication operations have no architectural state (such as the accumulator register employed by the MIPS architecture). Stateless multiplication simplifies the exploitation of instruction level parallelism in multi-issue processors, especially when multiple multiplications can be issued simultaneously [52]. Furthermore, limitation of architectural state simplifies context save and restore in the case of interrupt or exception handling.

Multiplications promote the data type of the source operands, e.g. when multiplying two 16-bit sources, a 32-bit result is produced. In case of two-way 16-bit SIMD multiplication, the two 32-bit results cannot be represented within a single 32-bit destination register. Multiple approaches exist to address this problem. *Saturation* clips the results of an arithmetic operation to a range that can be represented within the operation destination, e.g. the result of a 16x16 bit signed multiplication may be clipped to the two-complement 16-bit signed integer range of [0x8000, 0x7fff]. *Truncation* throws away some of the lower bits of the operation, e.g. the lower 16 bits of the result of a 16x16 bit signed multiplication may be thrown away, at a loss of precision (truncation is similar to post-normalization as employed by floating point operations). Rather than simply throwing away least significant bits from a certain "cut-off" bit position, *rounding* may be performed. E.g., a 16x16 bit signed multiplication with a 32-bit result of 0x1234:8765 may be truncated with rounding to a 16-bit result of 0x1235 (assuming a rounding to the nearest representable integer), rather than truncated to a 16-bit result of 0x1234. Especially for code sequences with multiplications in which the truncation error accumulates, rounding may provide the additional precision to guarantee algorithm compliancy (e.g. MPEG2 8x8 DCT/IDCT kernels).

The TM3270 adds several multiplication operations to the TriMedia ISA that support saturation, truncation and rounding. To illustrate the functionality, we

describe the DUALISCALEUI_RNINT operation. This two-way 16-bit SIMD multiplication calculates two 16-bit signed results, each a product of a 16-bit unsigned and 16-bit signed value:

```
DUALISCALEUI_RNINT src1 src2 -> dst

temp        = src1[31:16]*src2[31:16];
rounding    = (temp < 0) ? 0x1fff : 0x2000;        // round to nearest integer
temp        = (temp + rounding) >> 14;
dst[31:16] = IMIN (IMAX (0x8000, temp), 0x7fff);

temp        = src1[15:0]*src2[15:0];
rounding    = (temp < 0) ? 0x1fff : 0x2000;        // round to nearest integer
temp        = (temp + rounding) >> 14;
dst[15:0]   = IMIN (IMAX (0x8000, temp), 0x7fff);
```

The first source operand $src1$ holds two unsigned 16-bit values, for which we assume a 2.14 fractional representation (2 integer bit positions and 14 fractional bit positions). The second source operand $src2$ holds two signed 16-bit values, for which we assume a s1.14 fractional representation (1 sign bit, 1 integer bit position and 14 fractional bit positions). The multiplication has a s3.28 fractional representation (1 sign bit, 3 integer bit positions and 28 fractional bit positions). After multiplication, a rounding factor is added to the in-between result to achieve "rounding to the nearest integer, away from zero". Truncation throws away the 14 lower significant bits of the rounded in-between result. Finally, the truncation result is saturated to the 16-bit signed integer range of [0x8000, 0x7fff]. Together, truncation and saturation normalize the rounded in-between result to the same s1.14 fractional representation as the second source operand $src2$. The 2.14 and s1.14 fractional representations of the sources and the truncation by 14 bit positions of the rounded in-between result allow for gain factors greater than 1 (with a maximum unsigned gain factor of 0b11.11111111111111 (binary representation)).

2.3.3 Two-slot operations

Two-slot or super operations were first proposed in [68], but only find first employment in the TM3270. Two-slot operations are executed by functional units that are situated in two neighboring issue slots. As a result, these functional units have twice the register-file bandwidth: operations may consume up to four 32-bit sources and produce up to two 32-bit results. To illustrate the ability of two-slot operations, we describe the SUPER_DUALIMEDIAN and SUPER_LD32R operations.

The SUPER_DUALIMEDIAN operation has three sources and a single destination. This two-way 16-bit SIMD operation calculates a 3-taps median filter on two-way 16-bit signed inputs:

```
SUPER_DUALIMEDIAN src1 src2 src3 -> dst

dst[31:16] = IMIN (IMAX (IMIN (src1[31:16], src2[31:16]), src3[31:16]),
                   IMAX (src1[31:16], src2[31:16]));
dst[15:0]  = IMIN (IMAX (IMIN (src1[15:0],  src2[15:0]),  src3[15:0]),
                   IMAX (src1[15:0],  src2[15:0]));
```

Without the new operation, two DUALIMIN and two DUALIMAX operations are required to implement the same functionality (both operations are available in the TriMedia ISA). This implementation occupies four issue slots, whereas the two-slot operation occupies only two issue slots, a reduction of a factor two. Furthermore, the SUPER_DUALIMEDIAN, DUALIMIN and DUALIMAX operations all have a latency of two cycles. The DUALIMIN and DUALIMAX implementation has a compound latency of six cycles, whereas the two-slot operation has a latency of two cycles, a reduction of a factor three. As the 3-taps median filter is a basic building block of many video algorithms, e.g. the temporal up-conversion algorithm (Chapter 7), performance is significantly improved when the SUPER_DUALIMEDIAN operation is used.

The SUPER_LD32R operation has two sources and two destinations. The operation retrieves two consecutive 32-bit values from memory:

```
SUPER_LD32R src3 src4 -> dst1 dst2

A           = src3+src4;        // calculate the memory address
dst1[31:24] = Mem[A];           // big endian mode assumed
dst1[23:16] = Mem[A+1];
dst1[15:8]  = Mem[A+2];
dst1[7:0]   = Mem[A+3];
dst2[31:24] = Mem[A+4];
dst2[23:16] = Mem[A+5];
dst2[15:8]  = Mem[A+6];
dst2[7:0]   = Mem[A+7];
```

The source operands are taken from the second operation in the operation pair, which explains why sources $src3$ and $src4$, rather than sources $src1$ and $src2$ are used, to calculate the memory address[1]. The new operation doubles the load bandwidth, when compared to a traditional LD32R operation. The restriction to consecutive memory address locations limits the applicability of the new operation, when compared to two separate LD32R operations. However, the SUPER_LD32R is efficiently supported by the TM3270 data cache organization, whereas the support for two separate LD32R operations would be more expensive or would produce more unpredictable execution behavior [45]. Texas Instruments' VelociTI architecture includes a similar load operation: LDDW, it also loads two 32-bit values from consecutive memory addresses [49]. However, its destination

[1] The rationale for this decision is given in Section 3.3.1

registers are restricted to a neighboring register pair, which limits the freedom of the scheduler's register allocator.

2.3.4 Collapsed load operations

Collapsed load operations with interpolation combine the functionality of a load operation, with that of a 2-taps filter function. Collapsed load operations [59] are a new type of operations that involves memory collapsing rather than the ALU collapsing presented in [69]. To illustrate the ability of collapsed load operations, we describe the LD_FRAC8 operation:

```
LD_FRAC8 src1 src2 -> dst

A            = src1;
weight       = src2[3:0];
dst1[31:24] = ((16-weight)*Mem[A]   + weight*Mem[A+1] + 8) / 16;
dst1[23:16] = ((16-weight)*Mem[A+1] + weight*Mem[A+2] + 8) / 16;
dst1[15:8]  = ((16-weight)*Mem[A+2] + weight*Mem[A+3] + 8) / 16;
dst1[7:0]   = ((16-weight)*Mem[A+3] + weight*Mem[A+4] + 8) / 16;
```

The operation retrieves five bytes from consecutive memory addresses, and performs a 2-taps filter function on neighboring bytes to produce a four-way 8-bit SIMD result. Note that a more traditional 32-bit architecture requires two loads to retrieve the five bytes and potentially multiple operations to perform the filter function. The LD_FRAC8 operation allows for efficient implementation of horizontal fractional 8-bit pixel calculation. This function is a basic building block of motion estimation (Chapter 5), which constitutes a significant computational part of video encoders, such as MPEG2 (Chapter 6) and H.264/AVC [44].

2.3.5 CABAC operations

Context-based Adaptive Binary Arithmetic Coding (CABAC) [38, 23] constitutes a significant part of a H.264/AVC video decoder [44, 72]. The intrinsic sequential behavior of the CABAC decoding process prohibits an efficient implementation on a multi-issue processor. Performance evaluations of a decoder indicated that as much as 50% of the overall decoding time may be spent in the CABAC process, when no specific operation support for CABAC decoding is provided (Section 4.3.2). The TM3270 adds specific CABAC operations to the TriMedia ISA to reduce these computational requirements [62, 66]. The following code sequence gives the definition of *biari_decode_symbol* function, which decodes a single binary value *bit* from a CABAC coded bitstream:

```
LpsRangeTable[64][4]    // range table for least probable symbol (LPS)
MpsNextStateTable[64]   // MPS state transition table
LpsNextStateTable[64]   // LPS state transition table

biari_decode_symbol (   // decodes a single binary value "bit" from the CABAC coded stream
   inout value,               // coding value, 10-bit value
   inout range,               // coding range, 9-bit value
   inout state,               // modeling context state, 6-bit
   inout mps,                 // modeling context MPS, 1-bit
   in    stream_data,         // bitstream data
   inout stream_bit_position, // bit position in "stream_data"
   out   bit)                 // decoded binary value
{
   stream_data_aligned = stream_data << stream_bit_position;
   range_lps           = LpsRangeTable[state][(range >> 6) & 3)];
   temp_range          = range - range_lps

   if (value < temp_range) {                              // MPS: most probable symbol
      value = value;             range = temp_range;
      bit   = mps;
      mps   = mps;               state = MpsNextStateTable[state];
   } else {                                               // LPS: least probable symbol
      value = value - temp_range; range = range_lps;
      bit   = !mps;
      mps   = mps ^ (state != 0); state = LpsNextStateTable[state];
   }

   while (range < 256) { // renormalization, at most 8 bits can be consumed
      value              =   (value << 1)
                           | ((stream_data_aligned >> 31) & 1);
      range              <<= 1;
      stream_data_aligned <<= 1;
      stream_bit_position += 1;
   }
}
```

The *biari_decode_symbol* function constitutes a significant part of the computational complexity of the CABAC decoding process. The conditional constructs, table lookups, and limited parallelism result in a relative long VLIW schedule length. The use of TM3270 operations (such as the CLSAME operation to efficiently implement the renormalization) shortens the VLIW schedule length, but does not bring it down to an acceptable level.

Despite its complexity, the *biari_decode_symbol* function does not maintain state across function calls; i.e. it functionality is fully determined by its input arguments. This opens up the possibility to implement the functionality with new, CABAC decoding specific operations. Ideally, we would like to implement the *biari_decode_symbol* functionality with a *single* new operation. However, the amount of input and output function arguments exceeds the capability of a single two-slot operation. Closer investigation of the function shows that an implementation with *two* new two-slot operations is possible, by grouping of semantically related arguments as 16-bit sub-operands of a 32-bit operand. The *value* (10-bit

value) and *range* (9-bit value) arguments are both related to a context, and are grouped in a two-way 16-bit representation. The *state* (6-bit value) and *mps* (1-bit value) arguments define the state of a probability model for a context, and are grouped in a two-way 16-bit representation. The other arguments are represented by dedicated operands. We introduce two new operations: SUPER_CABAC_CTX and SUPER_CABAC_STR. The SUPER_CABAC_CTX operation calculates the new values of the context modeling: $dst1$ contains $(value, range)$ and $dst2$ (*state*, *mps*). Note that for this calculation, all function input arguments are required: $src1$ contains $(value, range)$, $src2$ contains *stream_bit_position*, $src3$ contains *stream_data* and $src4$ contains $(state, mps)$. The SUPER_CABAC_STR operation calculates the new values related to the bitstream processing: $dst1$ contains *stream_bit_position* and $dst2$ contains *bit*. Note that for this calculation, only a subset of the function input arguments are required (*stream_data* is not required): $src1$ contains $(value, range)$, $src2$ contains *stream_bit_position*, $src3$ is not used and $src4$ contains $(state, mps)$. With the definition of the *biari_decode_symbol* function and the grouping of function arguments in register operands, the interfaces of the two-slot SUPER_CABAC_CTX and SUPER_CABAC_STR operations are as follows:

```
SUPER_CABAC_CTX src1 src2 src3 src4 -> dst1 dst2

src1 = (value, range)
src2 = stream_bit_position
src3 = stream_data
src4 = (state, mps)
dst1 = (value, range)      \\ function defined by "biari_decode_symbol"
dst2 = (state, mps)

SUPER_CABAC_STR src1 src2 src4 -> dst1 dst2

src1 = (value, range)
src2 = stream_bit_position
src4 = (state, mps)
dst1 = stream_bit_position \\ function defined by "biari_decode_symbol"
dst2 = bit
```

Clearly, the new CABAC operations violate our original intend that operations should have "applicability in multiple domains". However, the benefit in terms of "performance enhancement" is so significant that we decided upon these specific operations.

2.3.6 Potpourri

In this section we describe a selection of new operations that are not captured by the new operation-types as described in the previous sections. In [30] some missing operations in terms of ISA orthogonality were identified. E.g., the TriMedia

architecture level 2 includes the two-way 16-bit DUALASR operation, which performs two 16-bit arithmetic right shifts. However, its counterpart, the DUALASL, was not accounted for. Several of these examples exist, and the identified missing operations were added to TriMedia architecture level 4.

When decoding a media bitstream, efficient calculation of the amount of leading '0' or '1' bits of a 32-bit operand is a useful functionality. To this end, the CLSAME operation was added to the TriMedia architecture:

```
CLSAME src1 src2 -> dst

temp = src1 ^ src2;
clz  = 0;

while (   (clz < 32)
       && (temp & (1 << (31-clz)) == 0))
   clz++;

dst = clz;
```

This new operation performs an "exclusive or" on its source operands, and calculates the amount of leading '0' bits of the in-between result. When source *src*1 contains the value 0, the amount of leading '0' bits of source *src*2 is calculated, when source *src*1 contains the value 0xffff:ffff, the amount of leading '1' bits of source *src*2 is calculated.

The ALLOC_SET data cache operation was added to set a data cache line (128 bytes) to a pre-defined 32-bit data value:

```
ALLOC_SET src1 src2

address = src2 & 0xffff:ff80;          // start of 128 byte line

for (i = 0; i < 32; i++) {
   Mem[address++] = src1[31:24];       // big endian mode assumed
   Mem[address++] = src1[23:16];
   Mem[address++] = src1[15:8];
   Mem[address++] = src1[7:0];
}
```

This new operation is useful to pre-set a sparsely encoded data structure with a pre-defined value as contained within *src*2. Furthermore, it is used to efficiently implement the *memset* standard C-library function.

2.4 Instruction cache LRU update

Both the TM3270 and its predecessor, the TM3260, support instruction and data cache locking on a cache line granularity. Instruction cache locking is used to

guarantee that certain pieces of code are kept in the instruction cache; i.e. the associated cache lines are not victimized by the LRU replacement algorithm. Locking is especially useful for infrequently executed code sequences that require predictable/timely execution behavior. The locked region of the overal cache capacity cannot be used for other code. This is acceptable as long as the locked region is small. However, as the locked region increases the limited cache capacity for other code may degrade overall processor performance. Therefore, it is not advisable to lock large, infrequently executed code sequences.

The TM3270 is used as an embedded processor and responsible for interrupt driven control of other SoC devices. Interrupt handlers are examples of code sequences that may be executed infrequently. The interrupt frequency of each interrupt source may be low, however, the amount of interrupt sources may be significant. Although "good" program practices advise to use small handler code, real-world handler code may measure multiple Kbytes in size. Locking multiple large interrupt handlers in the instruction cache will most likely degrade overall processor performance. Therefore, we intend to limit the impact of handler code on other code with respect to cache utilization, rather than optimize the predictable/timely execution behavior of handler code.

Typical handler code is characterized by a control processing like execution behaviour with if-constructs, but without too many loop-constructs. As a result, code is typically executed once, rather than multiple times. This does not match well with the temporal locality as offered by the instruction cache, as is illustrated by the following example. The TM3270 has a 8-way set associative instruction cache, a LRU replacement algorithm and 128 byte cache lines (each way contains 64 cache lines or 0x2000 bytes). For the sake of simplicity, we assume the LRU is in its default state (way 0 is least-recently used and way 7 is most-recently used). Furthermore, we assume that the cache is filled with media-processing code: Figure 2.3, a. We assume that the media processing code in the cache covers the memory region from address 0x0:0000 to 0x1:0000. Assume a first interrupt handler executes a fully sequential code sequence, starting at address 0x1:0000 and with a size of 0x1800. After the handler is executed the cache utilization is given by Figure 2.3, b. Next, a second interrupt handler executes a fully sequential code sequence, starting at address 0x1:1800 and with a size of 0x4000. After the handler is executed the cache utilization is given by Figure 2.3, c. Next, a third interrupt handler executes a fully sequential code sequence, starting at address 0x1:5800 and with a size of 0x2800. After the handler is executed the cache utilization is given by Figure 2.3, d. At this point, half of the cache capacity (ways 0, 1, 2 and 3, or 0x8000 bytes) is utilized by interrupt handler code that is unlikely to be used in the near future (low interrupt frequency). Furthermore, half of the original media-processing code has been removed from the cache. When the

processor continues with the media-processing code at address 0x0:2000 it finds it removed from the cache and will reload the code starting at way 4, overwriting other media processing code still present in the cache. After executing a fully sequential sequence, starting at address 0x0:2000 and with a size of 0x6000, the cache utilization is given by Figure 2.3, e.

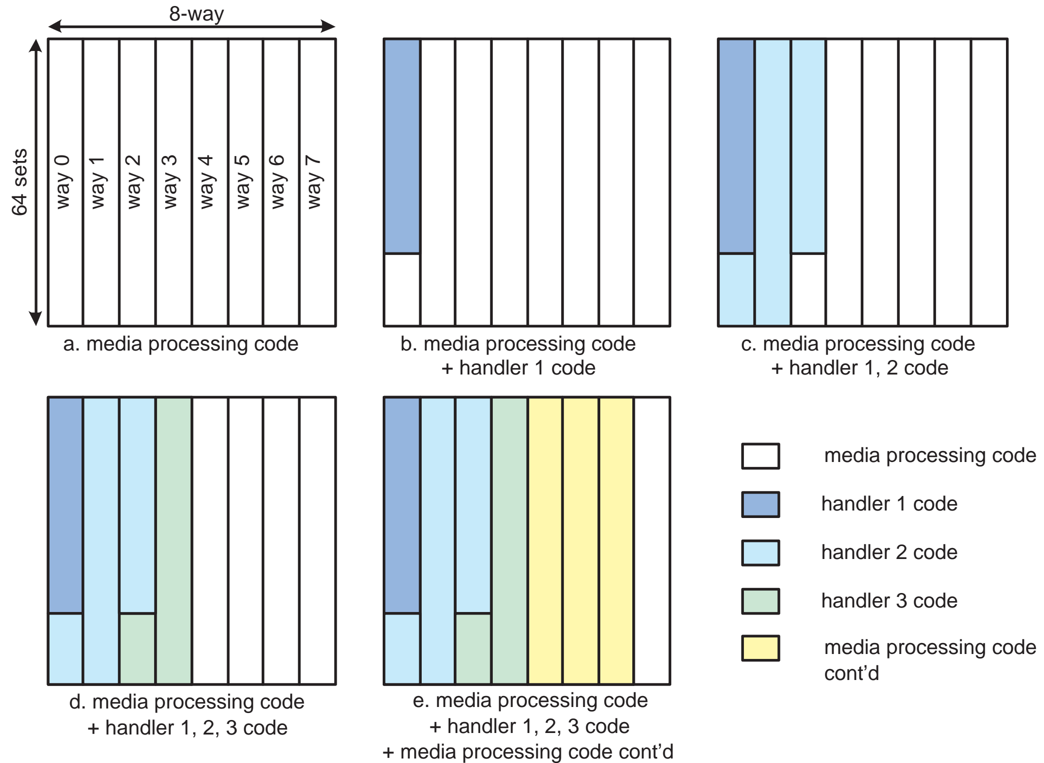

Figure 2.3: *Instruction cache utilization, with LRU status update turned on.*

The previous example illustrates that code with non-existent temporal locality may utilize a large cache capacity due to the continuous update of the LRU status of the cache sets. To address this over-utilization of cache capacity by code with low temporal locality, we made the LRU status update programmable [71]. When the LRU update is turned on, the cache functions as described above, when the LRU update is turned off, cache line updates are restricted to the LRU line of each of the cache sets. When we turn off the LRU update for handler code, a rerun of the example scenario results in instruction cache utilization as given by Figure 2.4. Handler code associated cache line updates are restricted to way 0: the LRU way for each of the cache sets. Initially, handler code removes media processing

code from way 0, however, handler code does not proceed to way 1 but removes previously retrieved handler code in way 0 instead. Since handler code is executed infrequently and without too many loop-constructs, removing previous retrieved and executed handler code from the cache should have no or limited performance impact. Note that when the processor continues with the media-processing code at address 0x0:2000 it finds the code still present in the cache.

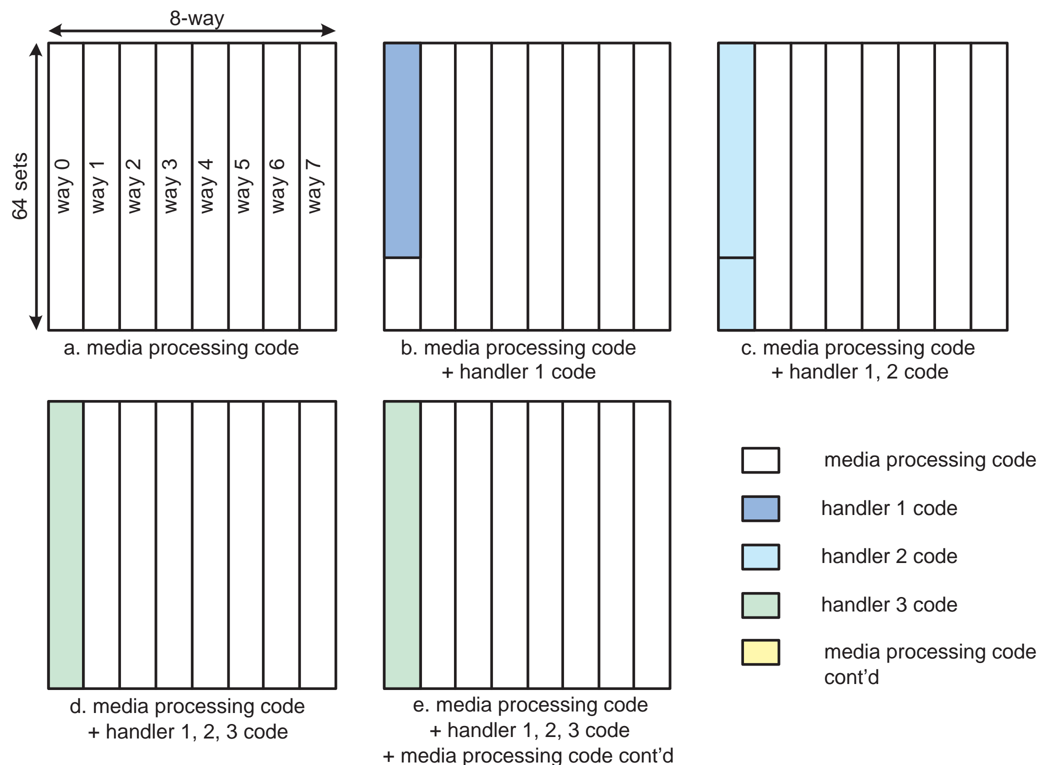

Figure 2.4: *Instruction cache utilization, with LRU status update turned off.*

2.5 Data prefetching

Data prefetching anticipates the use of data by the processor: it retrieves data from the off-chip memory into the processor data cache *before* the actual use of the data by the processor. Successful prefetching improves processor performance by eliminating compulsory misses and the associated stall cycles. Multiple prefetching approaches exist, ranging from fully software based approaches [4] to

fully hardware based approaches [27, 61, 22].

Both the TM3270 and its predecessor, the TM3260, support software based prefetching through dedicated PREFETCH operations. These operations may be added by the programmer to the source code, or by the compiler/scheduler to the binary code. Although the use of PREFETCH operations may improve processor performance, the process of adding the operations is cumbersome. Furthermore, the PREFETCH operations may consume valuable issue slots, which may have a negative performance impact. To address these drawbacks, the TM3270 adds a new prefetch approach to the TriMedia architecture that does not require the use of explicit PREFETCH operations in the instruction stream.

The new prefetch approach is a combined software/hardware based approach. It is based on memory regions, and allows for a prefetching pattern that reflects the access pattern of a data structure mapped onto a certain address space [57]. The TM3270 supports four separate prefetch memory regions. The identification of these memory regions and the required prefetch pattern is under *software control*, and defined by the following parameters (n = 0, 1, 2, 3): PFn_START_ADDR, PFn_END_ADDR and PFn_STRIDE. The reason for the decision to rely on software rather than hardware to identify the memory regions is two fold. Firstly, the programmer is likely to have the knowledge of a data structure access pattern, whereas a hardware mechanism needs to dynamically detect the access pattern. Secondly, no memory structures, such as stride prediction tables [16], associated with hardware prefetch pattern detection mechanisms are required, which allows for a low cost implementation in terms of silicon area. Note that prefetching of unused data may degrade processor and SoC performance, due to an increase of memory bandwidth. The up-front programmer's knowledge about a data structure access pattern is more likely to prevent unnecessary prefetching, than a hardware detection mechanism.

The first two prefetch parameters, PFn_START_ADDR and PFn_END_ADDR, are used to identify a memory region, the third prefetch parameter, PFn_STRIDE, is used to specify the prefetch pattern for the associated region. When the processor *hardware* detects a load from an address A within a prefetch region x, a prefetch request for address A+PFx_STRIDE is sent to the prefetch unit. When the prefetch address is not yet present in the data cache it is retrieved from the off-chip memory and *put into the cache*. The large data cache capacity of 128 Kbyte and the 4 way set associativity make it unlikely that useful data is victimized. Furthermore, no dedicated prefetch storage structures such as stream buffers or stream caches are required [27, 13], which allows for a low cost implementation in terms of silicon area.

Traditional next-sequential cache line or one-block-ahead [53] prefetching is realized by setting the prefetch pattern to the cache line size of 128 bytes. At reset,

prefetch memory region 3 is set to perform next-sequential cache line prefetching for the complete 32-bit address space of 4 Gbyte (PF3_START_ADDR = 0, PF3_END_ADDR = 0xffff:ffff and PF3_STRIDE = 128).

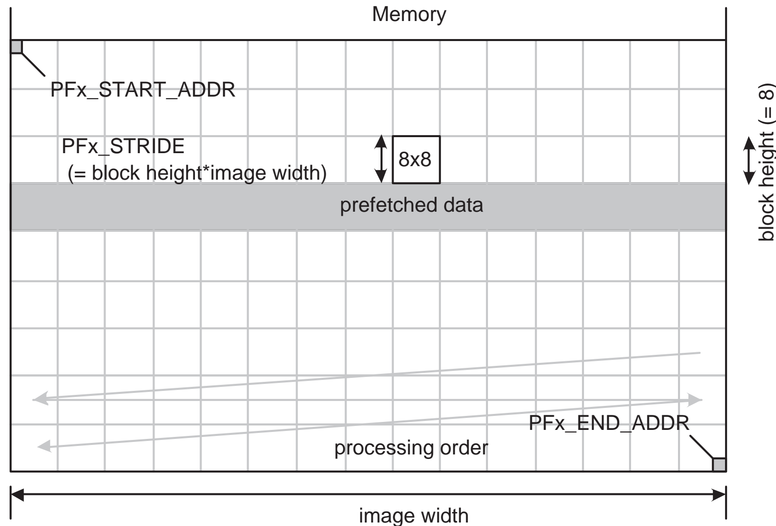

Figure 2.5: *Memory region based prefetching, an architecture perspective. Prefetch memory region settings to prefetch a row of blocks in advance.*

The effectiveness of memory region based prefetching becomes apparent when we consider e.g. the block-based processing of an image (Figure 2.5). Assume an image of byte element in memory. The image is processed at 8x8 block-size granularity. Starting with the upper-left block, the blocks are processed in a left-to-right, and top-down order. The memory region (PFx_START_ADDR and PFx_END_ADDR) is set to include the image, and the associated prefetch pattern (PFx_STRIDE) is set to the image width times the block height of 8. While processing a certain row of blocks, the lower row of blocks is prefetched into the data cache. If the time to process a row of blocks exceeds the time to prefetch the lower row of block, the processor will not incur any stall cycles due to compulsory misses.

2.6 Conclusions

In this chapter we gave an overview of the TM3270 architecture and described in more detail those architectural functions that distinguish the TM3270 from its

predecessor: the TM3260. In terms of extensions to the TriMedia ISA, we described two-slot operations, collapsed load operations, multiplication operations with rounding and clipping support and CABAC decoding operations. Besides these ISA extensions, we described an instruction cache replacement policy that prevents cache trashing for code with low temporal locality. Furthermore, we described memory region based prefetching: a combined software/hardware technique that prefetches data into the data cache with limited overhead to the programmer.

Chapter 3

Implementation

In the previous chapter we described the TM3270 architecture. This chapter describes the TM3270 implementation: the methods used to achieve the architectural functions. As in the previous chapter, we focus on those aspects of the TM3270 design that distinguish it from its predecessor: the TM3260. Section 3.1 describes the processor pipeline and gives an overview of the units that make up the processor implementation. Section 3.2 describes the instruction fetch unit and Section 3.3 describes the load/store unit. Section 3.4 presents a summary and some conclusions.

3.1 Processor pipeline

This section gives an overview of the TM3270 pipeline partitioning (Figure 3.1). The pipeline depth depends on operation latency: single-cycle latency operations have a pipeline depth of 7 stages (stages I1, I2, I3, P, D, X1 and W), multiple-cycle latency operations have additional execute stages Xi ($i \geq 2$). Stages I1 through I3 implement the sequential instruction cache: the cache tags (stage I1) and the instruction storage (stage I3). Every cycle, a 32-byte chunk of instruction information can be retrieved from the instruction storage. Chunks of instruction information are stored in a 4-entry instruction buffer in stage P. The instruction buffer decouples the front-end of the pipeline (stages I1, I2, I3 and P) from the back-end of the pipeline (stages D, X1, X2, ..., W). In stage P, a VLIW instruction is retrieved from the instruction buffer through proper alignment of the instruction information based on the program counter (PC). Next, the individual operations are extracted from the VLIW instruction and the guard and source registers are identified. Stage D decodes the individual operations and determines the operations' operands: immediate operand fields in the operation words are extracted, the

register-file is accessed with the guard and source register identifiers and register operand bypassing is performed. The TriMedia architecture uses a unified register-file structure. To sustain the issue rate of five operations per VLIW instruction, 5 guard register read ports and 10 source register read ports are available. Note that the five guard register read ports are only 1-bit wide, as guarding only depends on the least significant bit of an operation's guard register. Stages X1 through X6 are the execute stages; the amount of execute stages is determined by the operation latency. The load/store unit is implemented in stages X1 through X6 and is connected to issue slots 4 and 5. Two-slot operations are executed by functional units that are situated in two neighboring issue slots. Note that these functional units have twice the register-file bandwidth.

Conditional and unconditional jump operations are performed in stage X1. The outcome of jump operations may affect the address that is used to access the instruction cache tags in stage I1. The cache tags (in stage I1) and the jump execution (in stage X1) are separated by 5 pipeline stages, which matches the amount of architectural visible jump delay slots. Matching architectural and pipeline delay, allows for stall cycle free control flow changes without jump/branch-prediction support [33]. The elimination of jump-prediction support allows for a low-cost implementation and avoids unexpected stall cycles due to miss-predictions. However, the task of the compiler/scheduler is complicated: the 25 issue slot of the 5 jump delay slot VLIW instructions need to be filled with useful operations. The TriMedia compiler/scheduler [21] exploits guarded execution to perform aggressive speculation [24] to fill the jump delay issue slots with useful operations.

Stage W gathers the operation results from the functional units and allows for up to 5 writes to the register-file.

3.2 Instruction fetch unit

This section describes the instruction fetch unit (IFU), which includes the 64 Kbyte instruction cache (8-way set-associative, 128 byte line). The IFU design decisions depend on the VLIW instruction encoding, as described in Section 3.2.1. Section 3.2.2 describes the pipeline paritioning of the sequential instruction cache.

3.2.1 VLIW instruction encoding

A VLIW instruction may contain up to five operations, which are template-based encoded in a compressed format to limit code size. Every VLIW instruction starts with a 10-bit template field, which specifies the compression of the operations in the *next* VLIW instruction. The 10-bit template field has five 2-bit compression

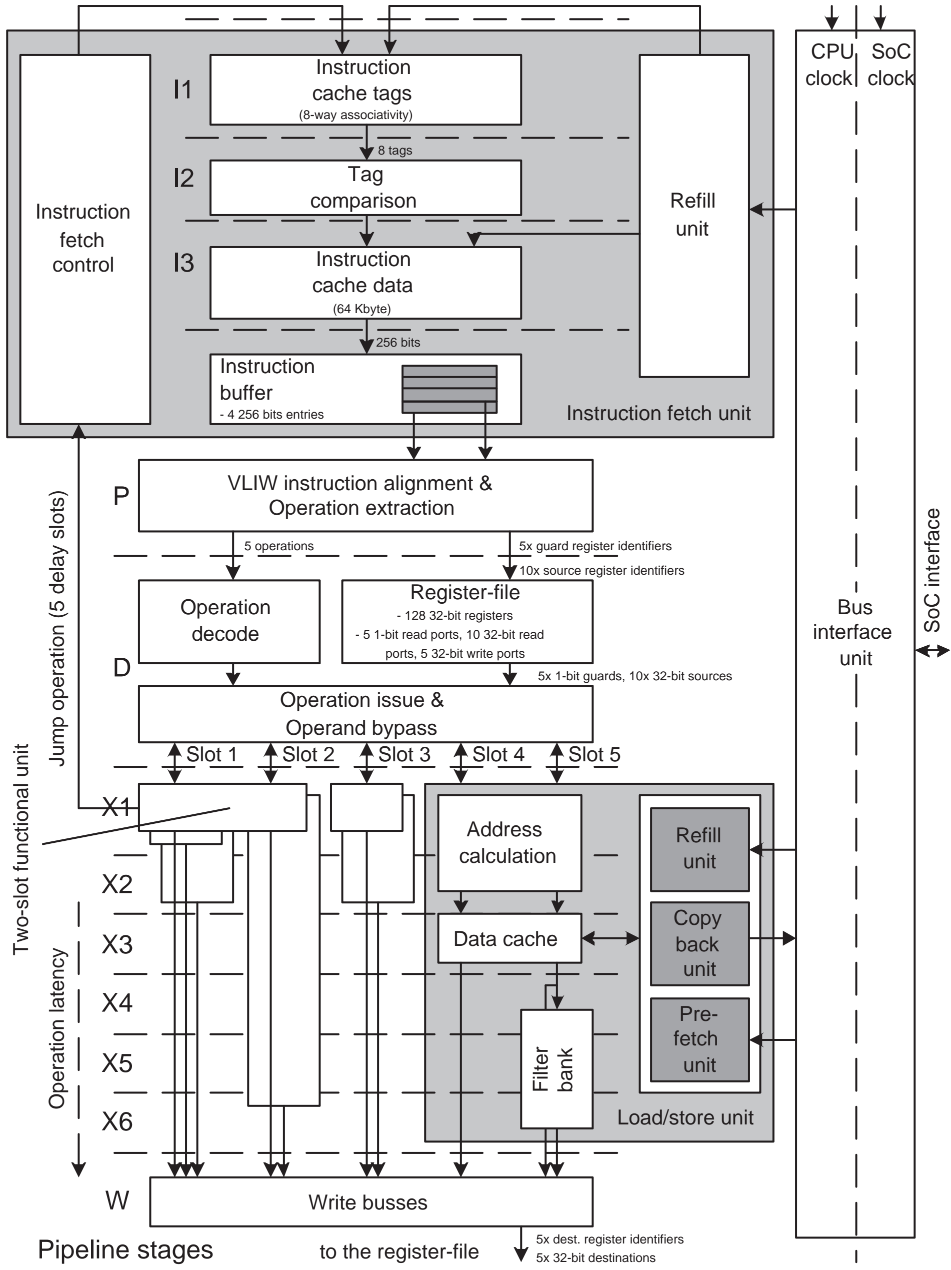

Figure 3.1: *TM3270 pipeline partitioning.*

sub-fields, which are related to the processor's issue slots 1 through 5. An issue slot's 2-bit compression sub-field specifies the size of the operation encoding.

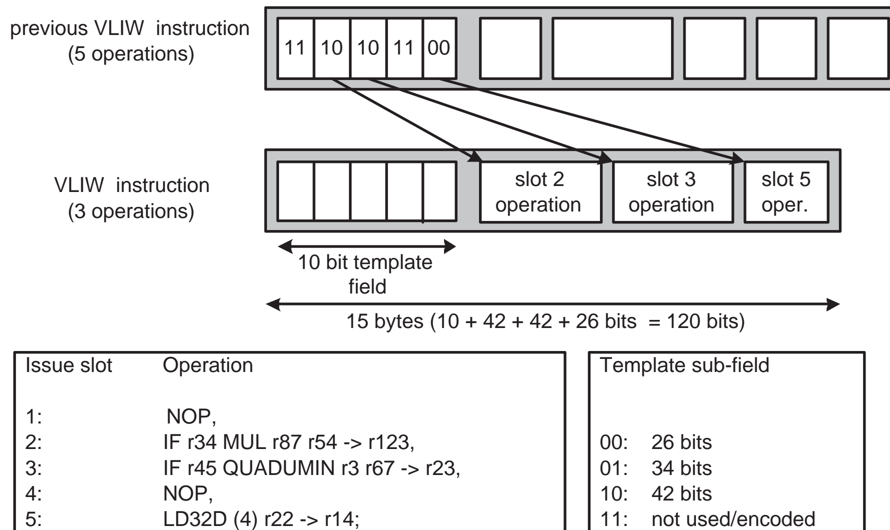

Figure 3.2: *VLIW instruction encoding.*

Figure 3.2 gives an example of a VLIW instruction containing three operations in slots 2, 3, and 5. Issue slots 1 and 4 are not used, as specified by the "11" encoding of the related compression sub-fields. Since issue slot 1 is not used, the first encoded operation is for issue slot 2. A VLIW instruction without any operations is efficiently encoded in 2 bytes, with a "11:11:11:11:11" template field. A VLIW instruction with all operations of the maximum size of 42 bits is encoded in 28 bytes, with a "10:10:10:10:10" template field and 5 * 42 bits for the operation encoding. This compression scheme allows for an efficient encoding of code with low instruction level parallelism: A NOP operation is encoded by a 2-bit "11" compression sub-field.

A VLIW instruction's template field is encoded as part of the *previous* VLIW instruction. As a result, an instruction's compression template is available one cycle before the instruction's compressed encoding, which relaxes the timing requirements of the VLIW instruction decoding process. However, this design decision introduces complications for conditional and register-indirect control flow changes. Consider the following VLIW instruction sequence with a conditional control flow change:

```
i:      NOP,  IF r2 JMPT target_address,   NOP,    NOP,    NOP;
i+1:    NOP,  NOP,                         NOP,    NOP,    NOP;   // delay slot 1
i+2:    NOP,  NOP,                         NOP,    NOP,    NOP;   // delay slot 2
i+3:    NOP,  NOP,                         NOP,    NOP,    NOP;   // delay slot 3
i+4:    NOP,  NOP,                         NOP,    NOP,    NOP;   // delay slot 4
i+5:    NOP,  NOP,                         NOP,    NOP,    NOP;   // delay slot 5

// next VLIW instruction: "i+6" or "j"?

i+6:    NOP,  NOP,                         NOP,    NOP,    NOP;

target_address j:
        NOP,  NOP,                         NOP,    NOP,    NOP;
```

The conditional jump operation in VLIW instruction i may be either taken or not-taken, dependent on the value of register $r2$ (the operation guard). As a result, VLIW instruction $i + 5$ precedes either VLIW instruction $i + 6$ or VLIW instruction j. However, the encoding of VLIW instruction $i + 5$ can only include the template field of one of its successors. Register-indirect control flow changes result in further complications[1]:

```
r13 may contain j, k, l, etc.

i:      NOP,  IF r12 JMPT r13,             NOP,    NOP,    NOP;
i+1:    NOP,  NOP,                         NOP,    NOP,    NOP;   // delay slot 1
i+2:    NOP,  NOP,                         NOP,    NOP,    NOP;   // delay slot 2
i+3:    NOP,  NOP,                         NOP,    NOP,    NOP;   // delay slot 3
i+4:    NOP,  NOP,                         NOP,    NOP,    NOP;   // delay slot 4
i+5:    NOP,  NOP,                         NOP,    NOP,    NOP;   // delay slot 5

// next VLIW instruction: "i+6", "j", or "k", or "l"?

i+6:    NOP,  NOP,                         NOP,    NOP,    NOP;

target_address j:
        NOP,  NOP,                         NOP,    NOP,    NOP;

target_address k:
        NOP,  NOP,                         NOP,    NOP,    NOP;

target_address l:
        NOP,  NOP,                         NOP,    NOP,    NOP;
```

The indirect jump operation in VLIW instruction i jumps to a target address kept in register $r13$, which is known at execution time, but is unknown at compile time. As a result, the compiler has no way of knowing which target VLIW instruction template field to include as part of the VLIW instruction $i+5$ encoding.

To overcome the complications as encountered by conditional and register-indirect control flow changes, the decision was made to encode jump target VLIW

[1] Register-indirect control flow changes are typically used to implement return from subroutine constructs or switch statements.

instructions uncompressed, they have a "10:10:10:10:10" template field which is implied by it being a jump target (rather than an explicitly encoded format field in the previous VLIW instruction). Since jump targets do not require the use of the format field of the previous VLIW instruction, this field can be used to encode the format field of the not-taken execution path; in the previous code sequences the format field of VLIW instruction $i + 5$ is used to encode the format of VLIW instruction $i + 6$.

3.2.2 Instruction fetch unit pipeline

The main reason for a sequential, rather than parallel, instruction cache design is power consumption. Figure 3.3 illustrates the sequential and parallel cache design alternatives. The relatively large 8-way set-associativity and instruction cache bandwidth requirements would make a parallel cache design expensive: to sustain the required instruction bandwidth of 28 bytes per cycle (the maximum VLIW instruction length), 8 * 28 bytes would need to be retrieved. In our sequential cache design, we retrieve only a single 32-byte chunk of instruction information every cycle[2]. As a result, the SRAM power consumption of the instruction memory structure is significantly reduced. However, the sequential cache design adds a jump delay cycle.

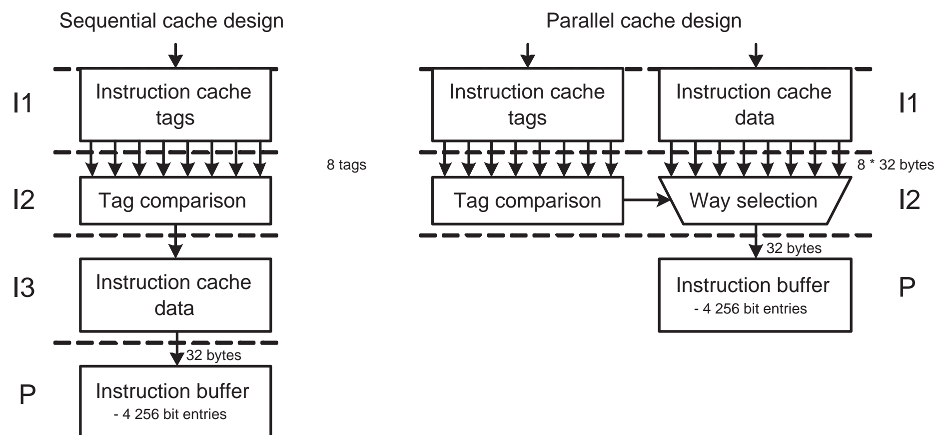

Figure 3.3: *Sequential (left) and parallel (right) instruction cache design.*

Due to the variable length VLIW instruction encoding as discussed in the previous section, a VLIW instruction's size is only known after the format field of

[2]Fetching 32 bytes, rather than the required 28 bytes, simplifies the organization of the instruction information in the instruction cache SRAM memories.

its predecessor is extracted. As a result, a VLIW instruction's program counter (PC) is only available in stage P, and not available in instruction fetch stages I0, I1, I2 and I3. The instruction fetch control unit determines the stage I0 address to control the fetching of 32-byte chunks of instruction information from the cache. Note that due to the pipeline delay from the I0-stage instruction fetch to the P-stage instruction extraction (4 stages), the calculation of the I0 fetch address needs to run at least 4 instructions ahead, to guarantee non-stalling execution behavior[3].

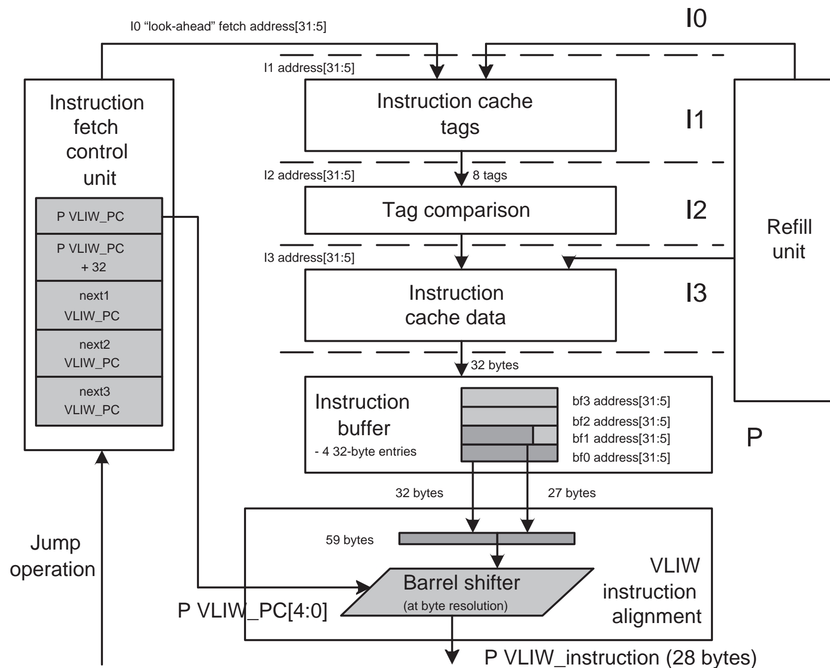

Figure 3.4: *TM3270 instruction fetch unit (IFU) pipeline.*

The instruction fetch control unit contains a prioritized list of instruction addresses; one of these addresses is used as the I0 "look-ahead" fetch address (Figure 3.4). The highest priority entry always contains the known P-stage VLIW_PC, the second highest priority entry contains VLIW_PC + 32. These top two entries are *both* related to the P-stage VLIW instruction. The other entries are related to successive VLIW instructions, and their addresses are dependent on the execution flow. In the case of sequential instruction flow, the addresses are calculated as 32-byte increments of VLIW_PC. Note that the PC's of successive instructions are not yet known; therefore, the calculated addresses are look-ahead addresses. In

[3]Non-stalling execution behavior is only guaranteed when no instruction cache misses occur.

the case of non-sequential instruction flow (taken jump operations), the addresses of jump target instructions are known and used as look-ahead addresses. To determine the I0 "look-ahead" fetch address, all of the IFU instruction fetch addresses (I1, I2, I3 addresses, and the four instruction buffer addresses) are taken into account. The I0 fetch address is determined as follows: the highest priority address that is not being fetched (present in stages I1, I2 or I3) or has been fetched (present in one of the four instruction buffer entries) is used as the next I0 "look-ahead" fetch address. Instruction cache misses are detected in stage I2, and the refill unit resolves these misses. The instruction buffer accepts a fetched 32-byte chunk in the P-stage. The buffer accepts the 32-byte chunks in the priority ordering as dictated by the entries in the instruction fetch control unit.

Assume an instruction flow with a P-stage VLIW_PC of 0x0000:0092 and a size of 0x14 bytes. The required instruction information crosses a 32-byte chunk boundary. Two chunks (0x0000:0080 and 0x0000:00a0) are required to extract the complete instruction, and therefore both chunks need to be fetched and available in the instruction buffer. Although an instruction's size is known when it is retrieved from the instruction buffer in the P-stage, the size is not yet known when its instruction information is accessed in the instruction cache in the I0 stage. With a chunk size of 32 bytes and a maximum instruction size of 28 bytes, it is likely that instruction information frequently crosses a 32-byte chunk boundary. To ensure that the instruction buffer always contains enough instruction information to extract a VLIW instruction in the P-stage, both of the top two entries of the instruction fetch control unit are related to the P-stage VLIW instruction.

In the case of sequential instruction flow and a VLIW_PC of 0x0000:0092 (with a size of 0x14 bytes), the instruction fetch control unit addresses are as follows: 0x0000:0092, 0x0000:00b2, 0x0000:00d2, 0x0000:00f2, 0x0000:0112 (in decreasing priority). The first two entries are related to the P-stage VLIW instruction; the last three entries are related to the three successive instructions. When the P-stage instruction is retrieved from the instruction buffer, the P-stage instruction size (0x14) is used to calculate the new P-stage VLIW_PC: 0x0000:00a6, and to calculate the new instruction fetch control unit addresses: 0x0000:00a6, 0x0000:00c6, 0x0000:00e6, 0x0000:0106, 0x0000:0126. Figure 3.5 illustrates the change of instruction fetch control unit entries for the sequential instruction flow.

In the case of non-sequential instruction flow (taken jump operations), the instruction fetch control unit receives jump target addresses from stage X1 (Figure 3.1). The TM3270 has 5 jump delay slots; i.e. the 5 VLIW instructions following the VLIW instruction containing the taken jump operations are executed before the instruction flow changes to the VLIW instruction at the jump target address. When the VLIW instruction with the taken jump operation is in stage X1, the first jump delay instruction is in stage D and the second jump delay instruction is in

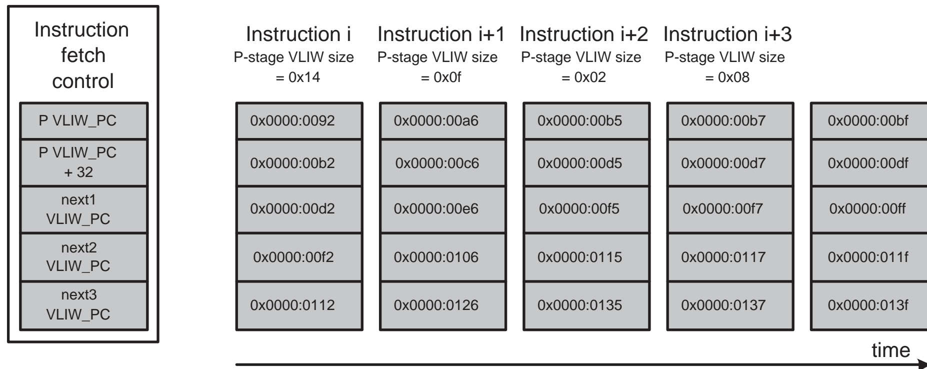

Figure 3.5: *Change of instruction fetch control unit entries for sequential instruction flow.*

stage P (the first two entries of the instruction control unit). The last three entries of the instruction control unit are related to the third, fourth and fifth jump delay instructions. When the P-stage instruction is retrieved from the instruction buffer, the new instruction fetch control unit addresses are calculated and the last entry is used to hold the address of the jump target instruction, as provided by the jump operation. Assume a VLIW_PC of 0x0000:0092 (with a size of 0x14 bytes), the instruction fetch control unit addresses are as follows: 0x0000:0092, 0x0000:00b2, 0x0000:00d2, 0x0000:00f2, 0x0000:0112. Furthermore, assume a taken jump operation in stage X1, with a target address of 0x0001:0000. When the P-stage instruction is retrieved from the instruction buffer, the P-stage instruction size (0x14) is used to calculate the new instruction fetch control unit addresses: 0x0000:00a6, 0x0000:00c6, 0x0000:00e6, 0x0000:0106, 0x0001:0000. The last entry holds the address of the jump target instruction. Figure 3.6 illustrates the change of instruction fetch control unit entries for the non-sequential instruction flow.

Note that it takes 4 instructions for the jump target instruction j, to reach the top entry of the instruction fetch control unit, which is the same number as the 4 cycles the instruction fetch pipeline takes to retrieve the instruction's 0x0001:0000 chunk of instruction information from the cache (stages I0, I1, I2 and I3). As a result, non-stalling execution behavior is guaranteed when the jump target instruction is completely contained within the 0x0001:0000 chunk. In our example, this condition is met: the 28-byte target instruction is encoded by instruction information bytes 0x0001:0000 through 0x0001:001b, inclusive[4]. However, for jump target addresses with a chunk address offset greater than 0x4, the instruction informa-

[4] Jump target instructions are uncompressed and are therefore always encoded by 28 bytes.

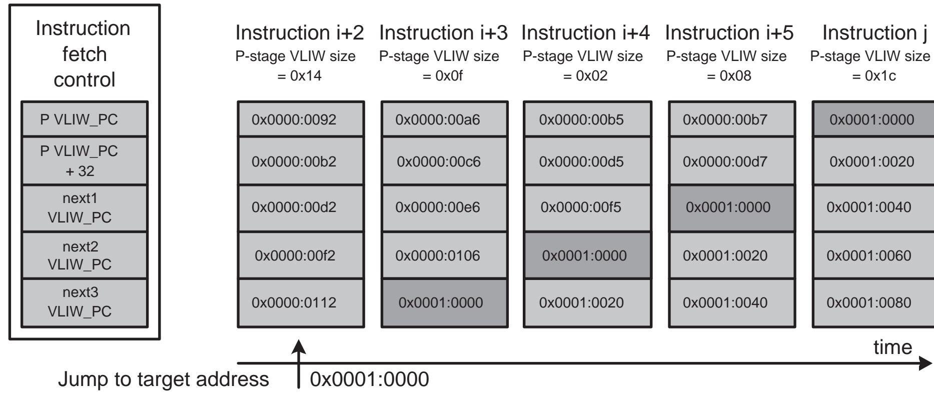

Figure 3.6: *Change of instruction fetch control unit entries for non-sequential instruction flow.*

tion crosses a 32-byte chunk boundary. The fetch for the next-sequential chunk (0x0001:0020) results in a stall cycle. The linker can prevent these stall cycles through proper alignment of jump target instructions; jump target addresses have chunk address offsets in the range of 0x0 through 0x4, to ensure that a 28-byte jump target is contained within a 32-byte chunk[5].

For a sequential flow of instructions, the fetching ahead of 32-byte chunks of instruction information at 32-byte increments of the P-stage VLIW_PC will eventually result in a full instruction buffer, as the maximum VLIW instruction size is only 28 bytes. This results in a non-stalling execution behavior, however, it may degrade performance in the presence of taken jump operations. Figure 3.6 illustrates the problem: when instruction $i+2$ is in the P-stage, potential I0 fetch addresses include 0x0000:00d2, 0x0000:00f2 and 0x0000:0112. When instruction $i+$ 5 (the fifth and last jump delay instruction) is in the P-stage, its PC (0x0000:00b7) and size (0x08) are known and it becomes obvious there was no need to fetch the chunks for addresses 0x0000:00d2, 0x0000:00f2 and 0x0000:0112. However, these fetches may have started and the associated instruction information accepted by the instruction buffer. The instruction information may have been useful for instructions $i + 6$, $i + 7$, etc., but not for the jump target instruction j. As a result of fetching ahead, the top two instruction buffer entries, on which the VLIW instruction aligner operates, contain useless instruction information. To prevent stall cycles associated with one-by-one removal of unused buffer entries, the buffer's FIFO behavior is extended with ”clear-all” functionality, which clears all buffer entries. This function is used to clear all buffer entries when the fifth

[5]Note that this linker optimization may increase code size.

and last jump delay instruction has been extracted from the buffer. As a result, the newly arriving 32-byte chunk of the jump target address is accepted as the top instruction buffer entry.

3.3 Load/store unit

This section describes the TM3270 load/store unit, which includes the 128 Kbyte data cache (4-way set-associative, 128 byte line size). The cache has a LRU replacement policy, a copy-back write policy and an allocate-on-write-miss policy. The allocate-on-write-miss policy is an improvement over the fetch-on-write-miss policy as used by the TM3260 data cache. Allocating a cache line in case of a write miss, rather than fetching the cache line, reduces the write miss penalty and the bandwidth to main memory. The following subsections describe the flow of load and store operations through the load/store unit pipeline (Section 3.3.1), the data cache memory organization (Section 3.3.2), memory arbitration (Section 3.3.3) and the implementation of memory region based data prefetching (Section 3.3.4).

3.3.1 Load/store unit pipeline

The TM3270 load/store unit is pseudo dual-ported and connected to issue slots 4 and 5: store operations are issued in slots 4 and 5, traditional load operations are issued in slot 5 (Figure 3.7)[67, 60]. The two-slot load operations, e.g. SUPER_LD32R, are issued in slots 4+5 and double the bandwidth of ordinary load operations. Although these operations use two issue slots, the cache access path of traditional load operations in slot 5 is re-used; slot 4 is only used to provide an additional destination operand. Traditional and two-slot load operations have a 4 cycle latency and produce results in stage X4. The collapsed load operations, e.g. LD_FRAC8, are issued in slot 5. Compared to traditional load operations, they have their pipeline extended with filter functionality in stages X4, X5 and X6. They have a 6 cycle latency and produce a result in stage X6.

Stage X1 calculates the address of memory operations based on the operation's source operands and addressing mode. The addresses of both the first and last referenced byte are calculated; i.e. for a 32-bit memory operation the first byte address *addr_lo* and the last byte address *addr_hi* are three bytes apart ($addr_hi = addr_lo + 3$). Stage X2 performs access arbitration for the tag and data memory structures. Although the functionality provided by this stage is limited, the delay is significant. This is because a large amount of relatively wide address and data busses need to be multiplexed and routed to the different memory structures. Furthermore, SRAM memory structure implementations typically have a

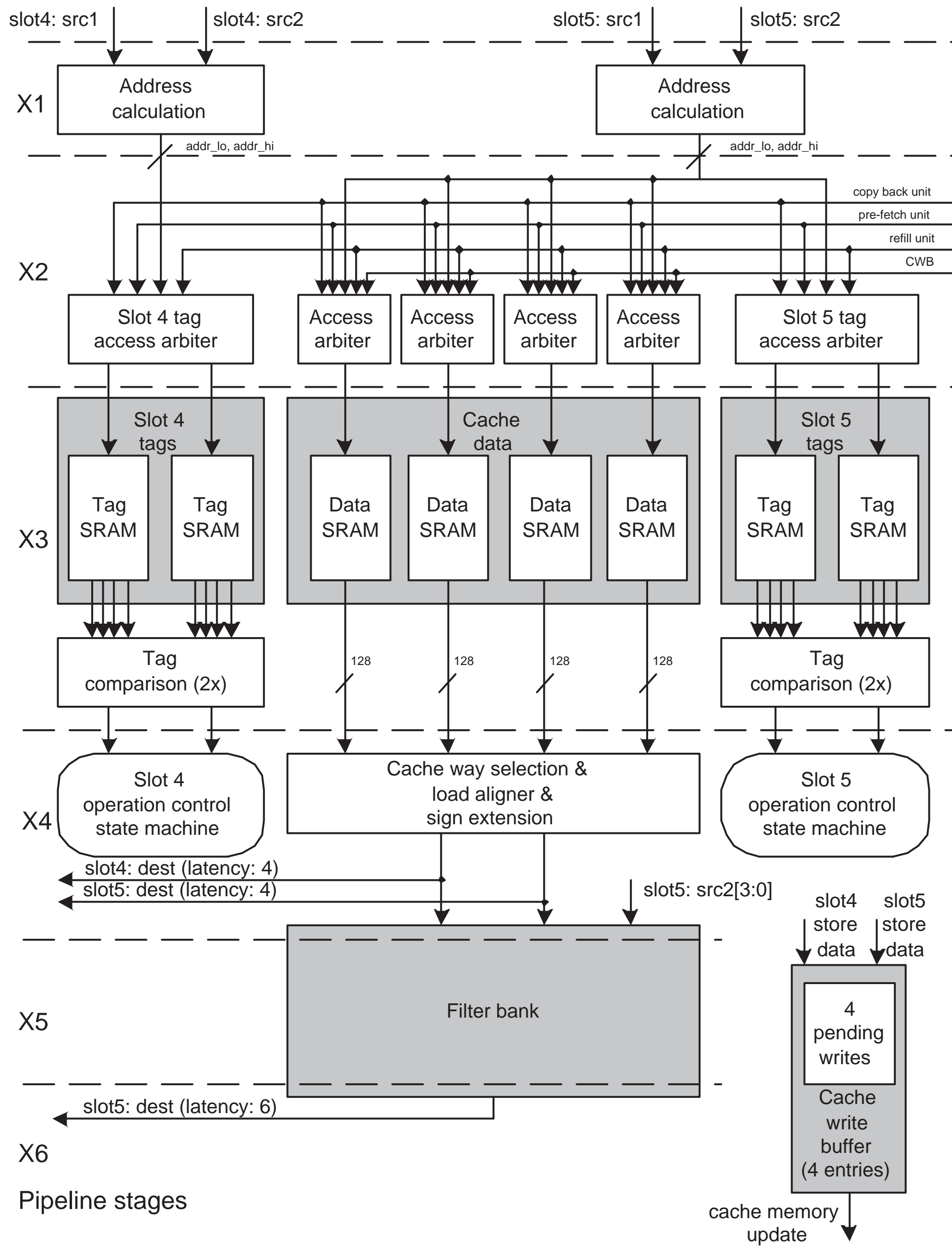

Figure 3.7: *TM3270 load/store unit pipeline.*

large setup time, which extends the presence of these memories from stage X3 into stage X2. Stage X3 contains the tag and data memory structures (this stage also contains the LRU and byte validity structures, which are not depicted). The data memory structures have a clock frequency that is close to the processor clock frequency. The tag memory structures are somewhat faster, which allows for the inclusion of tag comparison logic in stage X3. Stage X4 contains two state machines that control the flow of operations through the load/store unit pipeline(s). Furthermore, it includes a block that re-organizes retrieved cache data based on cache way information, operation type (e.g. signed or unsigned load operation) and processor endianness. Stages X5 and X6 contain a filter bank to provide the filter functionality as required for collapsed load operations. At the lower right side of Figure 3.7, the cache write buffer (CWB) holds pending updates to the data cache.

The flow of load and store operation through the pipeline is as follows. Stage X1 calculates the addresses of the first and last referenced bytes of load and store operations. For load operations, stage X2 requests access to the cache tag and cache data structures, which are read in stage X3. For store operations, stage X2 requests access to the cache tag structure, which is read in stage X3. Note that store operations do not require access to the cache data structure: they do not produce a destination register operand. Both load and store operations require access to the cache tag structure to generate the cache hit signal. The generation of a cache hit signal for loads is somewhat complicated by the allocate-on-write-miss policy, since the validity of the requested bytes needs to be checked. The stage X4 state machines use the retrieved cache information, such as the cache hit signal, to further control the flow of load and store operations. In case of a load hit, way selection is performed on the retrieved cache data. In case of a load miss, a cache line is retrieved from main memory by the refill unit. Note that two types of load misses may occur: A load miss may indicate that the cache line is not present in the cache or that the cache line is present in the cache, but the requested bytes are not all valid. In the first case, the retrieved cache line is put directly in the cache, in the second case, the retrieved cache line is merged with the already valid bytes in the cache. In case of a store hit, the store data is put into the CWB. In case of a store miss, the refill unit allocates a cache line.

The CWB acts as temporary storage for pending stores to the data cache and allows us to delay the moment at which the store data is put in the cache.

3.3.2 Memory organization

Figure 3.8 shows the organization of the data cache memory structures in SRAMs. The cache tag, cache data and cache byte validity memory structures are im-

plemented with off-the-shelf single ported SRAMs with bit-write functionality, to allow for a selective update of memory bits as identified by a bit mask. Available SRAMs had a maximum data width restriction of 128 bits.

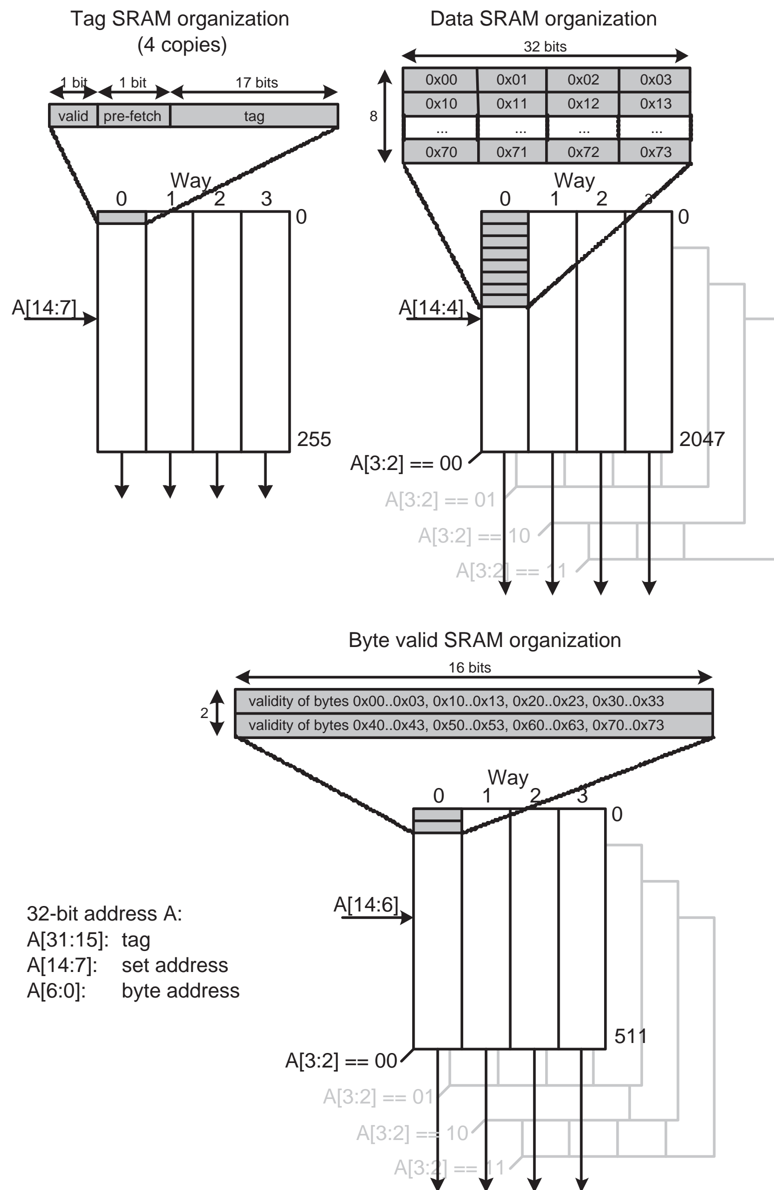

Figure 3.8: *Data cache SRAM organization.*

Issue slots 4 and 5 have dedicated tag memory structures and each structure contains two copies of the cache tags, resulting in a total of four copies of the cache tags. Each tag structure has two copies of the cache tags to support non-aligned memory access, without incurring processor stall cycles. The SRAMs are indexed with the set address $A[14:7]$ of a 32-bit address A, and each SRAM entry contains the tag information of the 4 ways within the selected set. The tag information consists of a cache line tag, a cache line valid bit and a cache line prefetch bit.

The data memory structure is partitioned into 4 separate SRAMs of 128 bits wide. The SRAMs are indexed with address $A[14:4]$ of a 32-bit address A, and each entry SRAM entry contains cache line data for each of the four ways. The SRAM partitioning is based on bits 3 and 2 of the address A; all byte elements that share these address bits reside in the same SRAM. As a result, a 128 byte cache line uses 8 entries of each SRAM, and cache lines are SRAM interleaved at a granularity of 4 byte elements.

The byte validity memory structure has a similar organization as the data memory structure. It is partitioned into four separate SRAMs of 64 bits wide. The SRAMs are indexed with address $A[14:6]$. The 128 byte validity bits of a cache line are located in 2 entries of each SRAM. Note that a cache line byte and the related byte validity bit are located in corresponding SRAMs, since both structure are partitioned based on bits 3 and 2 of the address A. As a result, the access arbitration for the two structures can be shared.

The data cache organization is illustrated by means of an example. Consider a 32-bit memory operation for address A=0x0000:00fc (*addr_lo* = 0x0000:00fc and *addr_hi* = 0x0000:00ff). The set address (bits 14 downto 7) is 0x01 for both *addr_lo* and *addr_hi*. The set addresses are used to index the two copies of the tag SRAMs in a tag memory structure. The tag is 0x0:0000 for both *addr_lo* and *addr_hi*. Address bits 3 and 2 determine which of the four data and byte validity SRAMs are accessed. Address bits 3 and 2 are 0b11 for both *addr_lo* and *addr_hi*, so data and byte validity SRAM "11" are accessed. The data SRAM is indexed with $A[14:4]$ = 0x00f and the byte validity SRAM is indexed with $A[14:6]$ = 0x003.

The 32-bit memory operation of the previous example was aligned, and consequently the address indexes for the different SRAMs are the same for both *addr_lo* and *addr_hi*. For non-aligned accesses this no longer holds. A non-aligned 32-bit memory operation for address A=0x0000:00fd (*addr_lo* = 0x0000:00fd and *addr_hi* = 0x0000:0100) requires access to multiple data and byte validity SRAMs. Furthermore, a 32-bit memory operation for address 0x0000:00fd crosses a cache line boundary. Therefore, the tags of *two* cache lines need to be investigated to generate the cache hit signal. To this end, each tag structure has two copies of the cache

tags. The set address is 0x01 for *addr_lo* and the set address is 0x02 for *addr_hi*. An alternative implementation is to have a single copy of the cache tags in each tag structure, and in the case of cache line crossing, two cycles are used to access both two cache lines. Although this alternative is cheaper (single copies of the cache tags would reduce area by 0.142 mm^2), it complicates the pipeline control, and produces more unpredictable execution behavior. Address bits 3 and 2 are 0b11 for *addr_lo* and are 0b00 for *addr_hi*, so data and byte validity SRAMs "11" and "00" are accessed. Data SRAM "11" is indexed with $addr_lo[14:4]$ = 0x00f and data SRAM "00" is indexed with $addr_hi[14:4]$ = 0x010. Validity SRAM "11" is indexed with $addr_lo[14:4]$ = 0x003 and validity SRAM "00" is indexed with $addr_hi[14:4]$ = 0x004.

Note that the data structure organization provides simultaneous (single cycle) access of up to 16 sequential bytes of a cache line. This allows for a 8 cycle cache line update by the refill and prefetch units, and for a 8 cycle cache line extraction by the copy back unit (a cache line is 128 bytes). By limiting the amount of cycles these units need for a cache line access, the interference with load and store operations is limited, reducing data cache stall cycles. The byte validity organization provides simultaneous access of up to 64 byte validity bits of a cache line. This allows for a 2-cycle cache line allocation by the refill unit (in the case of a write miss).

The data memory organization is partitioned into four SRAMs. Increasing the amount of SRAMs can increase the bandwidth to this structure. This would allow for even more efficient cache line update and extraction by the refill, prefetch and copy back units. However, layout and routing is complicated as the amount of SRAMs increases. Furthermore, doubling the amount of SRAMs (each with half the capacity) more than doubles the silicon area, due to the overhead of address decoders and sense amplifiers in SRAM design.

3.3.3 Memory arbitration

Figure 3.7 shows that separate memory arbiters control the access to the tag and data memory structures. Furthermore, each of the four SRAMs in the data memory structure has its dedicated arbiter. This allows for a low granularity access of the memory structures, resulting in high cache efficiency. For example, a 32-bit store in slot 4, a 32-bit load from address 0x0000:000fc in slot 5 and two CWB updates to addresses 0x0000:0004 and 0x0000:0008, can all be granted access to the required memory structures in the same cycle. The store operation requires access to the tag memory structure in slot 4, the load requires access to the tag memory structure in slot 5 and access to SRAM "11" in the data memory structure and the two CWB updates require access to SRAMs "01" and "10" in the data

memory structure.

Besides the granularity of the access control, cache efficiency is affected by arbiter priority setting. The following five separate entities that request the memory structures are distinguished:

- Issued operations
- CWB
- Refill unit
- Prefetch unit
- Copy back unit

All may request access to the data memory structure in the same cycle, and only one can be granted access (apart from simultaneous access by issued operations and CWB updates to mutually exclusion SRAMs).

The following priority setting is used for the arbitration of the data memory structure in *normal operation mode* (listed in decreasing priority):

1. Copy back unit
2. Refill unit
3. Issued operations
4. CWB
5. Prefetch unit

The rationale is as follows. A copy back operation has the highest priority, because a victimized cache line typically frees up a cache line location for a cache line refill that most likely stalls the processor. A refill operation has the second highest priority. This operation includes both the allocation and retrieving of a cache line. Both will stall the processor till completion. Next in line are the issued operations: they get the highest priority as long as no copy back or refill operation is required. Since store data is kept pending in a buffer, the CWB priority is lower than that of the issued operations. Lowest priority is given to the prefetch unit. Prefetches retrieve cache lines based on anticipated future use, and are typically not stalling the processor.

When the issued operations are granted access to the data memory structure, the low granularity access control grants the CWB access to those SRAMs that are not required by the issued operations.

The priority setting may be changed in the following exceptional situations:

- The CWB is full, and required by a store.
- A load has an address conflict with pending store data in the CWB.
- The prefetch unit raises its priority.

The first and second situations cause the CWB to have the highest priority. When the first situation occurs, a CWB entry needs to be freed up for a new store operation: at least one of the pending store data elements needs to be put in the data memory structure. Likewise, the second situation can only be resolved by putting the data of the conflicting pending store in the data memory structure (data forwarding from the CWB is not supported, for timing closure reasons). The details of the third situation and the effect on the priority setting are described in the next section.

3.3.4 Data prefetching

The TM3270 provides memory region based prefetching as described in Section 2.5. When the processor detects a load from an address A within a prefetch region x ([PFx_START_ADDR, PFx_END_ADDR]), a prefetch request for address A+PFx_STRIDE is sent to the prefetch unit.

The prefetch unit has a request buffer for up to six outstanding prefetch requests. When the prefetch request buffer is almost continuously at the maximum of its capacity, the effectiveness of prefetching may decrease because new prefetch requests will overflow the buffer. Therefore, when the amount of prefetch buffer entries exceeds a high water mark (more than four entries occupied) or when a prefetch buffer entry turns into a compulsory miss, the arbitration priority of the prefetch unit for the data memory structure is raised. This results in the following priority setting for the arbitration of the data memory structure in *prefetch operation mode* (listed in decreasing priority):

1. Copy back unit
2. Refill unit
3. Prefetch unit
4. Issued operations
5. CWB

The relative priority of issued operations and the prefetch unit has been reversed. This priority change accelerates prefetching, especially when almost every

VLIW instruction contains a memory operation. For these code sequences, the *normal operation mode* priority setting would grant access to the issued memory operations and the prefetch unit may *starve*: it can not update the prefetched data into the data cache, its single cache line data buffer gets full, and no new prefetch request to memory can start.

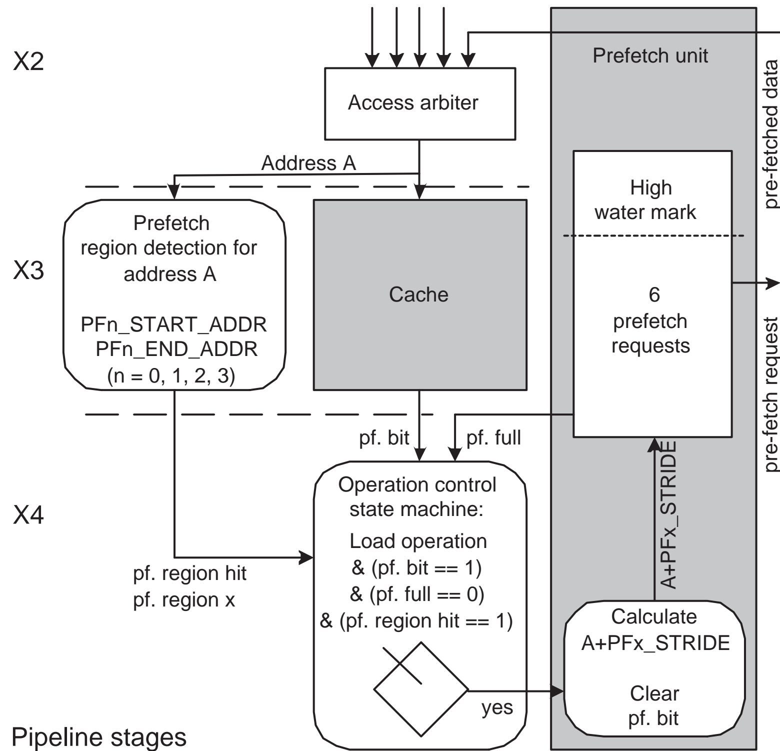

Figure 3.9: *Memory region based prefetching, an implementation perspective.*

It is not uncommon for almost every VLIW instruction to contain a load operation. Every load operation that accesses a prefetch memory region may potentially trigger a prefetch request. As a result, for every load from address A, a prefetch address A+PFx_STRIDE may have to be calculated, and the presence of this prefetch address in the cache needs to be checked. A traditional approach would require a dedicated tag SRAM to provide the required bandwidth to check the presence of prefetch addresses in the cache. We decided upon a different approach that does not require a dedicated tag SRAM, and is therefore cheaper to implement [58]. Whenever a cache line is created in the data cache, either by a refill or prefetch request, a cache line prefetch bit is set to '1'. This introduces an overhead

of a single bit to every cache line (Figure 3.8). A prefetch will only be considered for those load operations that access a prefetch region and have their prefetch bit equal to '1' (Figure 3.9). Furthermore, a prefetch request will only be sent to the prefetch unit, when its prefetch request buffer is not full; i.e. its six entries are not all occupied. Only when these conditions are met, will the prefetch bit of the cache line be set to '0'. Our approach has the following advantage: every cache line gives rise to at most one prefetch request, which prevents the possibility of multiple duplicate prefetch requests for the same cache line. The amount of checks for the presence of prefetch addresses in the cache is reduced. As a result, the existing tag memory structures can be used to check the presence of prefetch addresses in the cache, without any significant performance penalty.

Since prefetches are only requested when the prefetch request buffer is not full, prefetch requests will not get lost. When they do not make it into the prefetch unit, the prefetch bit will remain '1', and a future load to the same cache line will initiate the same prefetch request if the buffer has freed up one of its entries. This try-and-retry mechanism is especially useful for memory access patterns that initiate a burst of prefetch requests in a relatively short period of time that would overflow the prefetch request buffer.

3.4 Conclusions

In this chapter we described the TM3270 implementation: an overview of the processor pipeline and more detailed descriptions of the instruction fetch unit and the load/store unit. For the instruction fetch unit, we described how a compressed VLIW encoding and a sequential instruction cache design are implemented that result in low power consumption. The load/store unit implements a novel semi dual-ported cache design, providing high data bandwidth, at a limited area penalty when compared to a single-ported cache. The cache sustains a high store bandwidth by allowing two operations per VLIW instruction and a high load bandwidth by sustaining a single load operation per VLIW instruction with a bandwidth of twice the datapath size. All load and store operations support non-aligned memory access, without incurring any processor stall cycles. Furthermore, a new data prefetching technique is described.

Chapter 4

Realization

In the previous chapter we described the TM3270 implementation. This chapter describes the TM3270 realization: the means used to materialize the implementation. In particular, we describe the first TM3270 realization in a low power CMOS process technology, with a 90 nm feature size. The low power realization allows for successful application in portable battery-operated markets. Section 4.1 describes the physical realization. Section 4.2 discusses power consumption. Section 4.3 gives some performance numbers of the first realization. This section includes performance data of a standard definition H.264 video decoder, and quantifies the speedup of the new CABAC decoding operations described in Section 2.3.5. Section 4.4 presents a summary and some conclusions.

4.1 CMOS realization

The first realization of the TM3270 is in a low power CMOS process technology, with a 90 nm feature size and six metal layers. The design is fully synthesizable and uses off-the-shelf single ported SRAMs. The processor reaches a frequency of 350 MHz under worst case operating conditions (125 C, 1.08 V, worst case process corner), and a frequency of about 450 MHz under typical conditions (125 C, 1.2 V, typical process corner). Figure 4.1 gives the processor floorplan with a breakdown in its major modules. Table 4.1 gives the area of the individual modules. The TM3270 area is 8.08 mm^2 and the cache SRAMs constitute roughly 47% of this area. The instruction fetch unit area includes the area of the SRAMs that are used to implement the instruction cache tags, 64 Kbyte instruction memory structure and LRU replacement memory structure. The load/store unit area includes the area of the SRAMs that are used to implement the data cache tags, 128 Kbyte data memory structure and byte validity structure. The data cache LRU replacement

structure is implemented with standard cell logic.

Module	*Description*	*mm^2*
IFU	Instruction Fetch Unit	1.46
Decode	VLIW instruction decoding	0.05
Register-file	128 entry register-file	0.97
Execute	All functional units	1.53
LS	Load/Store unit	3.60
BIU	Bus Interface Unit	0.24
MMIO	Memory Mapped IO peripherals	0.23
Total		8.08

Table 4.1: *TM3270 area breakdown. The IFU and LS areas include the SRAMs that are used to implement the cache designs (64 Kbyte instruction cache and 128 Kbyte data cache).*

The TM3270 design has configurable cache sizes. The first implementation and realization as presented in this thesis instantiates a 64 Kbyte instruction cache and a 128 Kbyte data cache to address the requirements of standard definition video algorithms. As mentioned the cache SRAMs 47% constitute of the total area. This design decision results in a high cache hit rate, which reduces off-chip memory bandwidth requirements and reduces stall cycles associated to cache misses. Furthermore, a high cache hit rate is beneficial for *system* power consumption, as less power is consumed in off-chip memory traffic. For algorithms that are less computationally demanding or operate on smaller image sizes, a TM3270 instantiation with smaller cache sizes may be implemented. As an example, a place and routed TM3270 instantiation with a 32 Kbyte instruction cache and a 32 Kbyte data cache measures around 5.5 mm^2.

The TM3270 floorplan is realized as follows: the SRAM memories are hand-placed and the standard cell logic is placed by tools; i.e. no labor intensive hierarchical placement of the modules is used to come to a place and routed design. The synthesizability of the design and the semi-automated place and route approach make it relatively easy to port the design to a different process technology. once the place and routed TM3270 design is integrated into a SoC design, it is likely that it is one of the largest blocks in the SoC, which may cause SoC routing problems. Figure 4.2 illustrates the problem: when the TM3270 is placed on a SoC boundary it prevents a shortest path routing of SoC interface signals to I/O pins, potentially complicating timing closure on these signals[1]. The TM3270 has feed thru channels to ease SoC interface signal routing: rather than routing

[1]Note that although the TM3270 is a relatively large block in the SoC, it does not connect to I/O pins directly. The communication to off-chip SDRAM memory is through a SoC memory controller block.

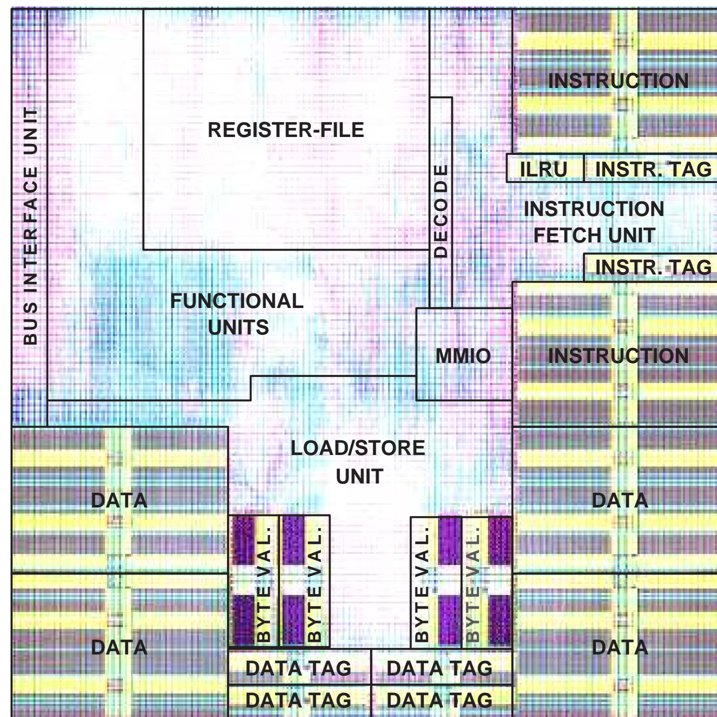

Figure 4.1: *TM3270 floorplan (64 Kbyte instruction cache and 128 Kbyte data cache).*

the interface signals around the TM3270, the interface signals are routed through the TM3270 design. A fixed amount of feed thru channels is provided as part of the place and routed TM3270, 80 channels are provided in each of the following directions: north-to-south, south-to-north, east-to-west and west-to-east. Each of the channels has a series of buffer and inverter standard cells to guarantee driving strength and signal integrity.

To improve *Design for Manufacturability* (DfM), multicut via insertion is applied in 72% of the design to increase the yield. *Design for Testability* (DfT) is addressed by Built-In Self Tests (BIST) for the SRAMs and by scan-chain based tests for the standard cell logic (with a 99.67% fault coverage based on a stuck-at fault model).

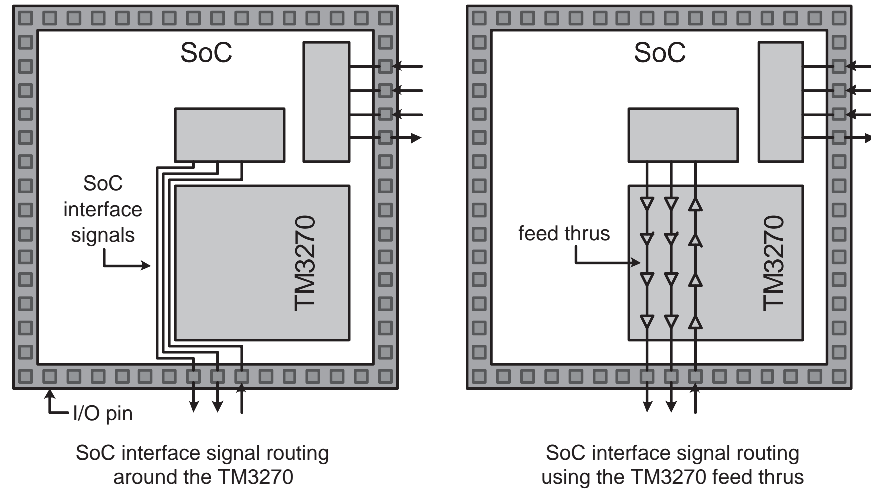

Figure 4.2: *Routing of SoC interface signals: around the TM3270 (left) and using the TM3270 feed thru routing channels (right).*

4.2 Power consumption

Low power consumption is one of the main requirements of the TM3270 design. For portable, battery-operated devices, low power has the obvious advantage of longer "playing time". However, also for devices with a power supply that plugs into the wall, low power consumption is desirable, since the use of dedicated cooling techniques adds cost. For the cost-driven consumer market, the use of cooling techniques should be kept to a minimal to keep overall system cost as low as possible. E.g., the use of heat sinks as a passive cooling technique to improve power dissipation adds to the IC package cost and the use of fans as an active cooling technique to increase the airflow adds to the overall system cost.

We distinguish both *static* and *dynamic* power consumption. Static power consumption is mainly determined by transistor leakage current. We use a low power *dual* (standard and high) threshold voltage CMOS process technology. Leakage current is proportional to the inverse exponential of the threshold voltage V_t. To keep the static power consumption small, both the standard threshold voltage *standard* V_t and the high threshold voltage *high* V_t are relatively high at 0.37 and 0.47 V respectively for the NMOS transistors. However, the high threshold voltages increase the transistor switching time, which has a negative impact on the device's maximum operating frequency. The TM3270 is realized with 78.4% *standard* V_t cells and 21.6% *high* V_t cells; the *high* V_t cells are used for the non-timing critical paths in the design.

Dynamic power consumption is defined by CV^2f, C is the switched capacitance, V is the supply voltage and f is the operating frequency of the device. The capacitance C is determined by process technology parameters and the device's activity level. For the TM3270 realization, the low power CMOS process technology is used for its low passive power consumption characteristics, but the technology parameters are fixed from a TM3270 design perspective. However, the device activity level is addressed as part of the TM3270 design.

At the *architecture level*, the decision for a statically scheduled VLIW approach has an advantage over alternatives such as a superscalar approach. A superscalar approach performs operand dependency analysis in hardware, whereas a VLIW approach moves this function to the compiler/scheduler. As a result, the analysis and the associated power consumption is removed from processor design. In line with the VLIW philosophy of keeping the hardware simple (and low power) and moving complexity to the compiler/scheduler, the decision was made to expose the processor's 5 jump delay slots at the architectural level, rather than to rely on hardware support for jump/branch-prediction.

At the *implementation level*, power reduction techniques can be found at different levels. At a relatively high level, the decision for a sequential instruction cache design, rather than a parallel instruction cache design, has an obvious power consumption advantage. At a lower level, *clock-gating* is used extensively to reduce power consumption, by turning off the clocks of unused registers in the design. As a result of clock-gating, a total of 172 functional clock domains are present. All of the TM3270's functional units are clock-gated: when a functional unit is not used by an operation, its clock is turned off. Functional unit clock-gating is performed at pipeline stage granularity; i.e. only those stages that are used by executed operations are clocked. As an example, consider the FALU functional unit, which has 4 pipeline stages. When a FSUB operation is issued in stage D in cycle i, the input registers of stage X1 of the FALU functional unit are clocked to receive the operation information (opcode, source operands, etc.). In the next cycle i+1, the input registers of stage X2 of the FALU functional unit are clocked to receive the in-between result of the FSUB operation. As the FSUB operation proceeds through the FALU pipeline, only the input registers of the following pipeline stage are clocked.

With 128 32-bit registers, 5 guard register read ports, 10 source register read ports and 5 destination register write ports, the register-file is a potentially power-hungry part of the design. Like the functional units, the register-file is heavily clock gated to reduce power consumption. Although the register-file has 10 source register read ports to sustain the issue rate of five operations per VLIW instruction, it is unlikely that all 10 read ports are required by the VLIW instruction: due to a lack of available ILP, not all VLIW instruction may contain five operations.

Furthermore, not all operations have two register operands: some operations only have a single register operand (e.g. FSQRT), or have an immediate operand (e.g. IIMM) for which no read port to the register-file is required. Each of the 10 source register read ports is implemented as a 128-to-1 multiplexer of 32 bits wide, the control input to this multiplexer is a 7-bit source register identifier and the data inputs are the 128 32-bit general-purpose registers. The activity level of this structure is mainly determined by the source identifier: this identifier may change every processor cycle and affects the flow through the structure, whereas only 5 of the general-purpose registers can change every cycle[2] and affect only a limited part of the flow through the structure. To reduce read port power consumption, the source identifiers of not required read ports are kept stable; i.e. the clock is turned off and the identifiers keep the value of the previous cycle. As a result, the activity level is only affected by changes in the values of general-purpose registers. To reduce write port power consumption, only those general-purpose registers addressed by the 5 write ports are clocked.

Power reduction is also applied to the off-the-shelf single ported SRAMs in the design. By turning off the chip-select input to these memories, power consumption is significantly reduced. As an example, the chip-select inputs of the four data memory structure SRAMs are individually controlled (Figure 3.8): only the to be accessed SRAMs are turned on.

The TM3270 supports a power-down mode, which it can enter when it is idle; i.e. it does not have any work to do. In this mode, almost all processor clocks are turned off[3] and the dynamic component of the power consumption is negligible.

The supply voltage V, in the dynamic power consumption equation CV^2f, is 1.2 V under typical conditions for our process technology, but functional correct operation is guaranteed for a supply voltage of 0.8 V. The asynchronous clock domain transfer in the bus interface unit that connects the processor to the SoC infrastructure has level shifters that allow for different supply voltages of the processor and SoC infrastructure. As a result, dynamic voltage scaling can be applied to set the processor's supply voltage V based on an application's computational requirements.

The operating frequency f, in the dynamic power consumption equation CV^2f, has a maximum of 450 MHz under typical conditions (1.2 V, typical process). The *required* frequency is determined by an application's computational requirements. Note that the video and audio application domain specific enhancements to the TM3270 ISA greatly improve the processor performance on multimedia applications. As a result, the required frequency of media-processors on multime-

[2]The register-file has 5 write ports.
[3]Only the clock for the power-down mode wake-up circuitry is turned on.

dia applications is typically lower than processor architectures without these ISA enhancements. The asynchronous clock domain transfer in the bus interface unit allows for operating frequencies independent of the frequency of the SoC infrastructure. As a result, frequency scaling can be applied to set the processor's operating frequency f based on an application's computational requirements, independent of the frequency of the SoC infrastructure.

As an example consider a MP3 decoder application (384 Kbits stereo decoding at 44.1 KHz). MP3 decoding is performed in 8 MHz with a OPI (operations per VLIW instruction) of 4.5 operations per VLIW instruction and a CPI (cycles per VLIW instruction) close to 1.0[4]. Synopsys' PowerCompiler tool [55] was used to measure the power consumption on a back annotated gate level netlist[5] (including SRAMs and the power grid). Table 4.2 gives a dynamic power consumption breakdown at an operating voltage of 1.2 V, the contribution of static power consumption is negligible for our low power CMOS process technology.

Module	*Description*	*mW/MHz*
IFU	Instruction Fetch Unit	0.272
Decode	VLIW instruction decoding	0.022
Register-file	128 entry register-file	0.107
Execute	All functional units	0.255
LS	Load/Store unit	0.266
BIU	Bus Interface Unit	0.002
MMIO	Memory Mapped IO peripherals	0.012
Total		0.935

Table 4.2: *TM3270 dynamic power consumption breakdown for the MP3 decoder application, at 1.2 V.*

MP3 decoding has a dynamic power consumption of 0.935 mW/MHz, and if all 450 MHz (125 C, 1.2 V, typical process corner) of processor performance were spent on MP3 decoding, 450 MHz * 0.935 mW/MHz = 421 mW is consumed. However, MP3 decoding is performed in 8 MHz and frequency scaling can be applied to have the processor operate at 8 MHz, consuming only 8 MHz * 0.935 mW/MHz = 7.48 mW. Voltage scaling can be applied to further reduce power consumption. The supply voltage may be reduced to 0.8 V: the lowest voltage that guarantees functional correct operation. Due to the quadratic dependency of dynamic power consumption on supply voltage, a reduction by a factor $1.2^2/0.8^2 = 2.25$ is achieved, which results in a power consumption of $\frac{1}{2.25}$ * 8 MHz * 0.935 mW/MHz = 3.32 mW.

[4] Mainly due to the effectiveness of data prefetching and the large 128 Kbyte data cache, which eliminates almost all data cache misses.

[5] The same gate level netlist as used for the final floorplan.

Power measurements on other applications showed that power consumption is more dependent on generic characteristics such as OPI and CPI, than on algorithmic specifics such as the type of operations performed: applications with similar OPI and CPI have a similar mW/MHz number. As the CPI increases (more stall cycles), the mW/MHz number decreases as more clock gating is performed. This illustrates one of the caveats of power analysis: power consumption should be measured for the full duration of an application and *not* in terms of mW/MHz.

Note that downscaling of processor frequency may reduce the computational requirements of an application. A CPI greater than 1.0 indicates stall cycles, which are most likely related to instruction and data cache misses. As the processor frequency is reduced (but the frequency of the SoC infrastructure is not changed), the cache miss penalty expressed in processor cycles and the amount of associated stall cycles is reduced. In other words, a MHz of processor performance becomes more useful as the processor operating frequency decreases.

4.3 Performance

For the performance evaluation of new processor designs, we use a Philips inhouse benchmark suite called MediaStone. It consists of about 50 applications and kernels from the media-processing domain. Section 4.3.1 compares TM3270 performance to that of it predecessor: the TM3260. The MediaStone benchmark suite applications are optimized for the TM3260; i.e. no optimizations are performed that use the new TM3270 features. Section 4.3.2 evaluates the performance impact of the new CABAC operations as discussed in Section 2.3.5. We also optimized several applications using the new TM3270 features: Chapters 5, 6 and 7 describe TM3270 optimization of the motion estimation, MPEG2 encoder and temporal upconversion applications. The performance impact of new operations, data prefetching and the allocate-on-write-miss policy are quantified in the context of these applications.

4.3.1 MediaStone

We made a selection of applications from the MediaStone benchmark suite, focusing on video-processing (Table 4.3) and compare TM3270 performance to that of it predecessor: the TM3260. The TM3260 finds commercial use in e.g. the PNX1500 IC [46]. The applications are optimized for the TM3260, and re-compiled for the TM3270 without modifications. Therefore, the performance results do not include improvements that could be achieved by applying TM3270 specific features (nonaligned memory access, advanced data prefetching, new operations, etc.). As such, the results are *a lower bound for achievable performance improvement.*

Kernel/application	*Description*
memset	Sets a 64 Kbyte memory region to a pre-defined value.
memcpy	Copies a 64 Kbyte memory region.
filter	From the EEMBC consumer suite.
rgb2yuv	From the EEMBC consumer suite.
rgb2cmyk	From the EEMBC consumer suite.
rgb2yiq	From the EEMBC consumer suite.
mpeg2_a	MPEG2 decoder, input sequence with highly disruptive motion vector field.
mpeg2_b	MPEG2 decoder.
mpeg2_c	MPEG2 decoder.
filmdet	Film detection algorithm, as used in TV sets.
majority_sel	De-interlacer algorithm, as used in TV sets.

Table 4.3: *Selection of video applications and kernels from the MediaStone benchmark suite.*

Table 4.4 lists the main characteristics of the TM3260 and TM3270 that cause difference in performance. The most notable are the operating frequencies and data cache capacity. To evaluate the impact of these characteristics, we measured four processor configurations. Configuration A represents the TM3260 and configuration D represents the TM3270. Configuration B represents the TM3270, operating at the TM3260 frequency of 240 MHz, and with TM3260 cache sizes. Configuration C represents the TM3270, operating at 350 MHz, and with TM3260 cache sizes. Note that to achieve the higher operating frequency of the TM3270 (350 MHz vs. 240 MHz), the amount of jump delay slots and the load latency is increased. As a result, the TM3270 has a deeper pipeline than the TM3260, which has a negative impact on the CPI. However, this is more than compensated by the TM3270 improved data cache design and its higher operating frequency, as is illustrated by the performance numbers (Figure 4.3). The performance measurements were performed with a 32-bit off-chip DDR SDRAM memory operating at 200 MHz.

Typically, the TM3260 (configuration A) has the lowest performance. However, for the MPEG2 application, configuration A outperforms configurations B and C. This is explained as follows. MPEG2 decoding is heavily dependent on the ability of the data cache to capture the working set. Although all configurations A, B and C have the same data cache capacity, the line size is different. The TM3270 doubles the line size to 128 bytes, resulting in more capacity misses for MPEG2 decoding, increasing the amount of stall cycles, and decreasing processor performance. Note that the TM3270 design decision for 128-byte line size was based on a 128 Kbyte data cache, as used for configuration D. Furthermore, the ability to perform two loads/VLIW instruction, the 3 jump delay slots and the 3-cycle load latency, give

Feature	*TM3260*	*TM3270*
Operating frequency	240 MHz	350 MHz
Instruction cache	64 Kbyte, 8 way set assoc.	64 Kbyte, 8 way set assoc.
	64 byte line size	128 byte line size
	parallel cache design,	sequential cache design,
	3 jump delay slots	5 jump delay slots
Data cache	16 Kbyte, 8 way set assoc.	128 Kbyte, 4 way set assoc.
	64 byte line size	128 byte line size
	fetch on write miss	allocate on write miss
	3 cycle load	4 cycle load
	dual-ported:	pseudo dual-ported:
	- two loads / VLIW instr.	- one load / VLIW instr.
	- two stores / VLIW instr.	- two stores / VLIW instr.

Table 4.4: *TM3260 and TM3270 main characteristics.*

the TM3260 an advantage over the TM3270. As illustrated by configuration D, the larger TM3270 data cache capacity more than makes up for this TM3270 disadvantage. The *memcpy* kernel shows the largest performance gain going from configuration A to B. The main reason is the TM3270's allocate-on-write-miss policy. The kernel is memory bound for both configurations. As the TM3270 generates less memory traffic, its performance is significantly higher.

On average, the TM3270 gives a performance gain of 2.29 over the TM3260. This number is achieved through re-compilation of applications optimized for the TM3260. Not all applications benefit to the same extent from a larger data cache. Whereas the MPEG2 application shows a large performance gain, the EEMBC kernels and TV algorithms show a modest performance gain. These applications benefit most from a higher operating frequency.

4.3.2 CABAC operations

We measured the performance of the CABAC decoding process (including overhead for decoder data structure maintenance and context computation) as part of a complete H.264 decoder. The H.264 standard allows for different decompositions of 16x16 macroblock into smaller 4x4 or 8x8 blocks [37]. As the macroblocks are decomposed into smaller blocks (e.g. sixteen 4x4 blocks) the control overhead of the decoder increases. To reflect this increased complexity, the performance was measured on a stream with 80% of the predicted macroblocks (for P and B frames) decomposed into sixteen 4x4 blocks. The bitstream has a 2.5 Mbits/s bitrate and represents a 4:2:0 PAL standard definition video stream (720*576 luminance pixels, at 25 frames/s.).

To illustrate the performance enhancement of the new CABAC operations

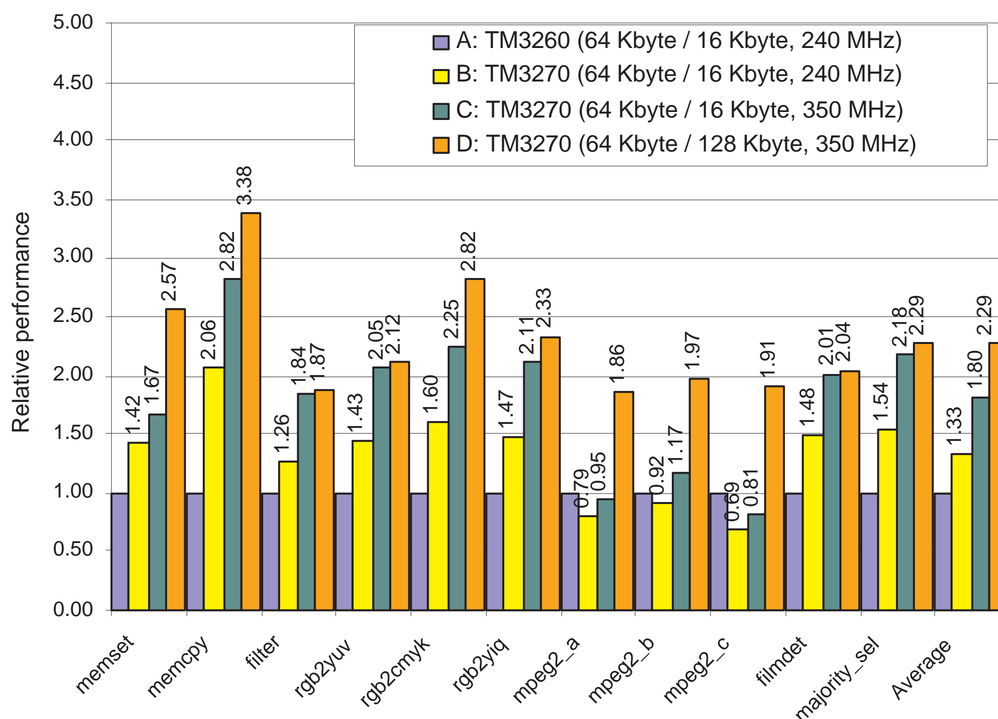

Figure 4.3: *TM3260 and TM3270 relative performance numbers on a series of video applications and kernels.*

(Section 2.3.5), the CABAC decoding performance is measured with and without the use of these new operations. Table 4.5 gives the results for the different frame types: the use of the new CABAC operations improves performance in the range of [1.5, 1.7]. Decoding of I-frames benefits most in both absolute and relative terms. In absolute terms, because I-frames are encoded with more bitstream bits than P- and B-frames. In relative terms, because I-frames have less overhead for decoder data structure maintenance and context computation, and as a result the new CABAC operations have a larger impact on the overall CABAC decoding process.

With the use of the new CABAC decoding operations, the TM3270 requirement of H.264 decoding at a sustained bitrate of 2.5 Mbits/s with a maximum dynamic performance complexity of 300 MHz is met, as illustrated by Figure 4.4. The figure gives the computational complexity of the individual frames, partitioned into the major decoding kernels: CABAC decoding, motion vector prediction, deblocking

Frame-type	*Average bits/frame*	*Cycles*	*Cycles/bit*	*Speedup*
Without CABAC operations				
I	615,599	14,748,701	24.0	-
P	149,915	4,942,957	33.0	-
B	34,604	1,287,888	37.2	-
With CABAC operations				
I	615,599	8,737,382	14.2	1.7
P	149,915	3,066,351	20.5	1.6
B	34,604	851,215	24.6	1.5

Table 4.5: *Dynamic performance complexity of the CABAC decoding process for I-, P- and B-frames of a PAL definition stream (2.5 Mbits/s): without and with the new CABAC decoding operations.*

control information calculation (this constitutes the calculation of the 4x4 block boundary strengths that are used by the actual deblocking), deblocking and the "other" partition, which collects all cycles that are not part of the previously mentioned kernels. Even with the new CABAC operations, a significant part of the computational budget is spent on the CABAC decoding process. To reduce the effect of individual peaks in performance complexity, as introduced by the decoding of computationally demanding I-frames, a four frame moving average $Average4$ is calculated to determine the processor performance requirements. Note that averaging (to spread the computational requirements) increases the memory requirements of the system solution and adds a delay to the video decoding chain. Around I-frames, the four-frame average reaches its maximum value of 250 MHz, well below the 300 MHz target.

4.4 Conclusions

In this chapter we described the TM3270 realization: a 8.08 mm^2 processor, with a power consumption of roughly 0.9 mW/Mhz. On average the TM3270 provides a speedup of 2.29 over its predecessor, the TM3260, for a series of video applications and kernels (Section 4.3.1). To achieve this speedup, the applications were re-compiled for the TM3270, without any optimizations that take advantage of the new TM3270 features. As a result, the speedup reflects a lower bound for achievable performance improvement. The new CABAC decoding operations improve the CABAC decoding process by a factor of 1.7 for I-frames (Section 4.3.2). The processor provides enough performance to allow for even the most demanding video applications, as illustrated by the dynamic performance complexity of a standard definition H.264 video decoder (a 2.5 Mbits/s standard definition stream

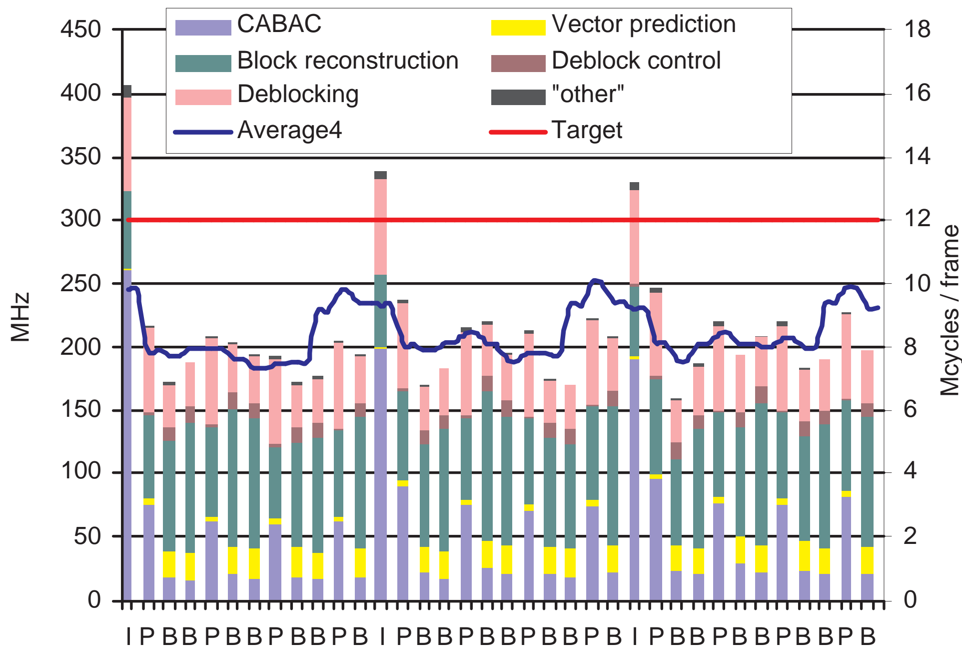

Figure 4.4: *Dynamic performance complexity of the H.264 decoder for a progressive PAL standard definition stream (2.5 Mbits/s). The left Y-axis represents the MHz requirement for decoding the 25 Hz stream, the right Y-axis represents the amount of cycles to decode a frame. The average "Average4" is a four-frame moving average.*

is decoded with a maximum frequency of 250 MHz, based on a four frame average).

To illustrate the performance potential of the TM3270, *using the new features*, the following three chapters evaluate the performance of three video algoithms: motion estimation (Chapter 5), MPEG2 encoding (Chapter 6) and temporal up-conversion (Chapter 7). Each of the algorithms is optimized using the TM3270 extensions to the TriMedia ISA and the benefits of non-aligned memory access and data prefetching are evaluated. These evaluations not only illustrate the TM3270 performance potential, but also provide a justification for the choice of the new features.

Chapter 5

Motion estimation

Motion estimation is used in industry-standard video encoders, such as MPEG2 (Chapter 6), MPEG4, H.264/AVC, and in proprietary video enhancement algorithms, such as temporal upconversion (Chapter 7). In both cases, motion estimation searches for motion-displaced spatial similarity between video images. The motion estimator typically uses a block-matching kernel that determines the similarity between two motion-displaced blocks of image pixels, taken from different video images. The Sum-of-Absolute-Differences (SAD) cost function is a popular means to determine similarity.

In the case of video encoders, the similarity is exploited to achieve a high compression factor. A small SAD value between two motion-displaced blocks implies a small difference between the two blocks, which is efficiently encoded in a small number of bits. In the case of temporal upconversion, the similarity is used to derive motion-accurate video images at temporal positions that were not present in the input video sequence. A small SAD value between two motion-displaced blocks indicates a *likely* movement of an object in the video sequence. The image pixels of the two blocks could be interpolated to calculate a block in a temporally interpolated video image. Whereas, video encoders are mainly interested in small block differences to allow for high compression, temporal upconversion requires an accurate portrayal of video object movement to derive new video images with good subjective quality. Although it is *likely* that a small SAD value indicates object movement, it is *not necessary.* To prevent interpolation based on motion-displaced blocks with a small SAD value that do not represent object movement, additional criteria can be added to evaluate candidate motion vectors. For example: as object sizes typically exceed block size, object movement should be represented by a certain consistency in the motion vector field of the blocks that cover the object. This consistency requirement could be added as a criterion, next to the SAD cost

function.

This chapter evaluates the performance of a motion estimation algorithm on the TM3270. We quantify the impact of new operations, data prefetching and off-chip memory latency on processor performance. In Section 5.1, we describe our motion estimation algorithm. In Section 5.2, we show how new TM3270 operations are used to improve the implementations of the block-matching kernel and discuss static performance complexity. In Section 5.3, we discuss the dynamic performance complexity of our motion estimation algorithm. Finally, in Section 5.4, we present the conclusions and suggestions for further optimization of the motion estimation algorithm. An earlier evaluation of motion estimation on the TM3270 processor can be found in [63].

5.1 Description of the algorithm

Many block-based motion estimation algorithms exist, and the suitability of a particular algorithm depends on the application at hand. We do not intend to introduce a new and better motion estimator, but rather intend to evaluate the performance of new TM3270-specific features on an existing motion estimation algorithm. For the motion estimation, we decided upon the 3-D Recursive Search (3DRS) algorithm [11]. This block-based algorithm performs a motion search using a limited set of candidate motion vectors, rather than an exhaustive motion search. It provides a high quality result at a relatively low performance complexity, making it an interesting candidate for a software implementation. The algorithm has found successful application in commercial ICs [10].

5.1.1 The estimator

The 3DRS algorithm is a spatial-temporal algorithm; i.e. candidate motion vectors for a block of image pixels are derived from the motion vectors of surrounding blocks in both space and time. Our version of the algorithm uses a block size of 8x8 image pixels. Let's assume a sequence of video images: a previous image $Image_p$, a current image $Image_c$ and a next image $Image_n$. The blocks in $Image_c$ are processed in a left-to-right, top-to-bottom sequence. Let $\overrightarrow{b_c} = \begin{pmatrix} b_c[x] \\ b_c[y] \end{pmatrix}$ denote the position of block b_c in $Image_c$, such that $\overrightarrow{b_c} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and $\overrightarrow{b_c} + \begin{pmatrix} 7 \\ 7 \end{pmatrix}$ identify the upper left and lower right pixel positions of block b_c. The horizontal (X) and vertical (Y) block positions of block b_c are integer multiples of 8 (the block size). Each block b_c in $Image_c$ is matched against 11 motion-displaced blocks b_n^i (i = 0,

..., 10) in $Image_n$, as indicated by candidate motion vectors $Mv^i(\overrightarrow{b_c})$, such that:

$$\overrightarrow{b_n^i} = \overrightarrow{b_c} + Mv^i(\overrightarrow{b_c}) = \begin{pmatrix} b_c[x] + Mv^i(\overrightarrow{b_c})[x] \\ b_c[y] + Mv^i(\overrightarrow{b_c})[y] \end{pmatrix}$$

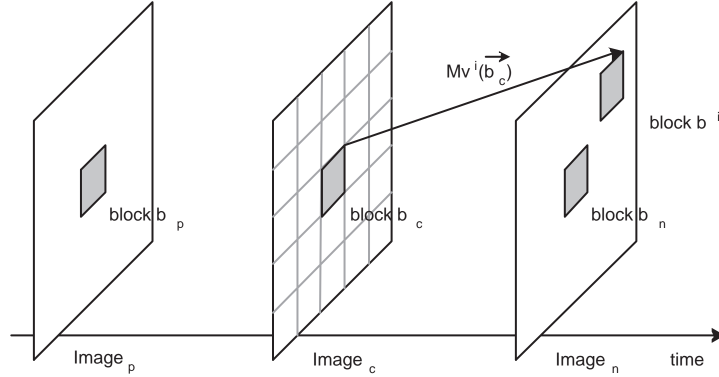

Figure 5.1: *Motion estimation: the current block b_c is matched against candidate blocks b_n^i in the next image $Image_n$.*

The motion vectors $Mv^i(\overrightarrow{b_c})$ have a $\frac{1}{4}$ pixel precision, and as a result, the block positions $\overrightarrow{b_n^i}$ have a $\frac{1}{4}$ pixel precision. For a block b, the integer position $\overrightarrow{int_b}$ is defined by $\lfloor \overrightarrow{b} \rfloor$, and the fractional offset $\overrightarrow{frac_b}$ is defined by $\overrightarrow{b} - \overrightarrow{int_b}$. At an integer position $\overrightarrow{b}$ the value of an image pixel is defined by $Image_n[\overrightarrow{b}]$, at a fractional position $\overrightarrow{b}$ the value is calculated using bi-linear interpolation:

$$\begin{aligned} Image_n[\overrightarrow{b}] = (\quad & (1 - frac_b[x])(1 - frac_b[y]) \quad Image_n[\overrightarrow{int_b}] \\ & + frac_b[x](1 - frac_b[y]) \quad Image_n[\overrightarrow{int_b} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}] \\ & + (1 - frac_b[x]) frac_b[y] \quad Image_n[\overrightarrow{int_b} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}] \\ & + frac_b[x] frac_b[y] \quad Image_n[\overrightarrow{int_b} + \begin{pmatrix} 1 \\ 1 \end{pmatrix}] + 2)/4 \end{aligned}$$

The calculation of a bi-linear interpolated 8x8 block at a horizontal and vertical fractional position requires a 9x9 block of pixel values.

The candidate motion vector that provides the best match (lowest SAD value) for the block at position $\overrightarrow{b_c}$, is denoted $BestMv(\overrightarrow{b_c})$. The motion vector candidates

are taken from the candidate sets CS_{zero} (the zero motion vector), $CS_{spatial}$ (the best motion vectors of already processed blocks in the current image $Image_c$), $CS_{temporal}$ (the best motion vectors of blocks in the previous image $Image_p$), and $CS_{noise_spatial}$ (noise updated best motion vectors of already processed blocks in the current image $Image_c$):

$$
\begin{aligned}
CS_{zero} &= \begin{pmatrix} 0 \\ 0 \end{pmatrix} \\
CS_{spatial} &= \left\{ \begin{array}{c} BestMv(\overrightarrow{b_c} + 8\begin{pmatrix} -1 \\ 0 \end{pmatrix}), \\ BestMv(\overrightarrow{b_c} + 8\begin{pmatrix} -1 \\ -1 \end{pmatrix}), \\ BestMv(\overrightarrow{b_c} + 8\begin{pmatrix} 1 \\ -1 \end{pmatrix}) \end{array} \right\} \\
CS_{temporal} &= \left\{ \begin{array}{c} BestMv(\overrightarrow{b_p}), \\ BestMv(\overrightarrow{b_p} + 8\begin{pmatrix} 1 \\ 0 \end{pmatrix}), \\ BestMv(\overrightarrow{b_p} + 8\begin{pmatrix} 2 \\ 0 \end{pmatrix}), \\ BestMv(\overrightarrow{b_p} + 8\begin{pmatrix} 1 \\ 1 \end{pmatrix}), \\ BestMv(\overrightarrow{b_p} + 8\begin{pmatrix} -1 \\ 1 \end{pmatrix}) \end{array} \right\} \\
CS_{noise_spatial} &= \left\{ \begin{array}{c} BestMv(\overrightarrow{b_c} + 8\begin{pmatrix} -2 \\ 0 \end{pmatrix}) + \overrightarrow{Noise_0}, \\ BestMv(\overrightarrow{b_c} + 8\begin{pmatrix} 0 \\ -1 \end{pmatrix}) + \overrightarrow{Noise_1} \end{array} \right\}
\end{aligned}
$$

Noise vectors $\overrightarrow{Noise_0}$ and $\overrightarrow{Noise_1}$ are cyclically selected from the list of noise

vectors NS:

$$NS = \left\{ \begin{array}{c} \begin{pmatrix} \frac{1}{4} \\ 0 \end{pmatrix}, \begin{pmatrix} -\frac{1}{4} \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \frac{1}{4} \end{pmatrix}, \begin{pmatrix} 0 \\ -\frac{1}{4} \end{pmatrix}, \\ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \\ \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \begin{pmatrix} -2 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 2 \end{pmatrix}, \begin{pmatrix} 0 \\ -2 \end{pmatrix}, \\ \begin{pmatrix} 4 \\ 0 \end{pmatrix}, \begin{pmatrix} -4 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 4 \end{pmatrix}, \begin{pmatrix} 0 \\ -4 \end{pmatrix} \end{array} \right\}$$

The horizontal component of a motion vector is unrestricted: motion-displaced blocks b_n can have any horizontal position in $Image_n$. The vertical component is restricted to the range [-40, $39\frac{3}{4}$]. Motion vectors that point to motion-displaced blocks b_n outside the $Image_n$ boundaries are clipped to the image boundaries.

5.1.2 Block-matching

Most of the performance complexity of our motion estimation algorithm is found in the block-matching kernel. The matching kernel determines the similarity between a block b_c in the current image $Image_c$ and a motion-displaced block b_n in the next image $Image_n$. We use the SAD cost-function is used to determine similarity: for two similar blocks b_c and b_n, $SAD(b_c, b_n)$ is a small value. The absolute differences may be calculated using the block pixels directly or indirectly, resulting in different block-matching approaches.

Traditional block-matching

The traditional block-matching approach performs the SAD function on the block pixels directly. The SAD value for blocks b_c and b_n is calculated as follows:

$$SAD(b_c, b_n) = \sum_{i=0}^{7} \sum_{j=0}^{7} |Image_c[\overrightarrow{b_c} + \begin{pmatrix} i \\ j \end{pmatrix}] - Image_n[\overrightarrow{b_n} + \begin{pmatrix} i \\ j \end{pmatrix}]|$$

The absolute differences of pixels at corresponding block positions are summed to determine block similarity. A total of 64 absolute differences are summed to calculate the SAD value.

Down-sampled block-matching

The performance complexity of the traditional block-matching approach may be reduced by limitation of the amount of values involved in the SAD calculation.

Rather than performing the SAD function on the block pixels directly, it is performed on down-sampled block data. E.g., for a horizontal down-sampling of a factor two, the SAD value for blocks b_c and b_n is calculated as follows:

$$SAD(b_c, b_n) =$$
$$\sum_{i=0}^{3}\sum_{j=0}^{7} | \quad (Image_c[\overrightarrow{b_c} + \begin{pmatrix} 2*i \\ j \end{pmatrix}] + Image_c[\overrightarrow{b_c} + \begin{pmatrix} 2*i+1 \\ j \end{pmatrix}] + 1)/2$$
$$-(Image_n[\overrightarrow{b_n} + \begin{pmatrix} 2*i \\ j \end{pmatrix}] + Image_n[\overrightarrow{b_n} + \begin{pmatrix} 2*i+1 \\ j \end{pmatrix}] + 1)/2|$$

A total of 32 absolute differences are summed to calculate the SAD value. However, additional calculations are required to calculate the down-sampled values. Therefore, this approach is only useful when the performance complexity of the absolute difference calculations exceeds that of the down-sampling. In Section 5.2.2 we show how new TM3270 operations are used to perform horizontal down-scaling by a factor two at no additional performance complexity. As a result of horizontal down-sampling, the quality of the block match and the motion estimator is degraded.

Note that the presented horizontal down-sampling approach is different from an approach in which block matching is performed on a down-sampled image. A horizontally down-sampled image (by a factor two) only contains half the image pixel values of the original image; i.e. the original image pixels values are lost. In our down-sampled block-matching approach, the down-sampling is performed as part of the block-matching function and all of the original image pixel values are input to the SAD calculation.

A reduction in performance complexity is achieved by not supporting vertical fractional offsets (at a loss of quality in the block match and the motion estimator). Ignoring the vertical fractional offset simplifies the fractional image pixel value calculation to a linear interpolation:

$$Image_n[\overrightarrow{b}] = (\quad (1 - frac_b[x]) \quad Image_n[\overrightarrow{int_b}]$$
$$+frac_b[x] \quad Image_n[\overrightarrow{int_b} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}] + 1)/2$$

5.2 Block-matching implementations

In this section we show how new TM3270 operations are used to efficiently implement the block-matching approaches as described in the previous section. Our

motion estimation algorithm matches each 8x8 block b_c in $Image_c$ against 11 motion-displaced blocks b_n^i (i = 0, ..., 10) in $Image_n$, using the $MatchBlock$ function. The blocks b_c are processed in a left-to-right, top-to-bottom sequence. We assume that image pixel values are represented by a single byte; i.e. 8 bits[1]. The C-like interface of the $MatchBlock$ function is given by:

```
int MatchBlock (
      uint8*          c_image,                // current image
      uint8*          n_image,                // next image
      intdual16       position_curr,          // current position in image
      intdual16       position_max,           // maximum position in image
      intdual16       mv,                     // motion vector
      int             image_stride)           // image stride/width
```

The function retrieves b_c and b_n block data from memory, and returns the calculated SAD value.

5.2.1 Traditional block-matching

We created five different implementations of the traditional block-matching approach. The implementations provide the same functionality, but differ in the extent to which they use TM3270 features: non-aligned memory access and new operations (Table 5.1).

Implem.	Non-aligned	SUPER_LD32	LD_FRAC8	SUPER_QUADUSCALEMIXUI
		Usage		
A	no	no	no	no
B	yes	no	no	no
C	yes	yes	no	no
D	yes	yes	hor. fract. pos.	no
E	yes	yes	hor. fract. pos.	vert. fract. pos.
		Operation count		
A	–	0	0	0
B	–	0	0	0
C	–	9	0	0
D	–	9	34	0
E	–	0	16	18

Table 5.1: *Traditional block-matching implementations: the used features and new operation counts to calculate block b_n.*

[1]Video images are typically represented by three values: one luminance value, and two chrominance values. For the sake of simplicity, we assume that block matching is performed on the luminance value only.

Implementation A is the reference implementation: it does not use any of the new features. Implementations B, C, D, and E gradually apply new TM3270 features to improve performance. Implementation A does not use non-aligned load operations and requires dedicated operations to properly align pixel values before fractional pixel calculation can commence for block b_n. The following code sequence shows how pixel values with a *fractional* horizontal offset and an *integer* vertical offset are calculated:

```
alignment  = n_image_position & 3;
t_x0123_y0 = *n_image_position++;   // aligned 32-bit access: lower two address bits ignored
t_x4567_y0 = *n_image_position++;   // aligned 32-bit access
t_x89ab_y0 = *n_image_position;     // aligned 32-bit access

n_x0123_y0 = (t_x0123_y0 << (alignment*8)) | (t_x4567_y0 >> (32 - (alignment*8)));
n_x4567_y0 = (t_x4567_y0 << (alignment*8)) | (t_x89ab_y0 >> (32 - (alignment*8)));
n_x89ab_y0 = (t_x4567_y0 << (alignment*8));

switch (x_frac) {       // Horizontal fractional interpolation.
case 0:                 // fractional offset of 0:
   break;
case 1:                 // fractional offset of 1/4:
   n_x0123_y0 = QUADAVG (n_x0123_y0,
                         QUADAVG (n_x0123_y0, FUNSHIFT1 (n_x0123_y0, n_x4567_y0)));
   n_x4567_y0 = QUADAVG (n_x4567_y0,
                         QUADAVG (n_x4567_y0, FUNSHIFT1 (n_x4567_y0, n_x89ab_y0)));
   break;
case 2:                 // fractional offset of 2/4:
   n_x0123_y0 = QUADAVG (n_x0123_y0, FUNSHIFT1 (n_x0123_y0, n_x4567_y0));
   n_x4567_y0 = QUADAVG (n_x4567_y0, FUNSHIFT1 (n_x4567_y0, n_x89ab_y0));
   break;
default:                // fractional offset of 3/4:
   n_x0123_y0 = QUADAVG (FUNSHIFT1 (n_x0123_y0, n_x4567_y0),
                         QUADAVG (n_x0123_y0, FUNSHIFT1 (n_x0123_y0, n_x4567_y0)));
   n_x4567_y0 = QUADAVG (FUNSHIFT1 (n_x4567_y0, n_x89ab_y0),
                         QUADAVG (n_x4567_y0, FUNSHIFT1 (n_x4567_y0, n_x89ab_y0)));
   break;
}
```

Note that the original TriMedia ISA does not support operations to perform a weighted interpolation at quarter resolution on byte elements. As a result, a switch statement is used, with a dedicated code sequence for each fractional offset, and the interpolation makes (repeated) use of the QUADAVG operation (four-way 8-bit average).

Implementation B uses non-aligned load operations to retrieve pixels values with proper alignment, thereby eliminating the need for dedicated alignment operations (Figure 5.2). The following code sequence shows how pixel values with a *fractional* horizontal offset and an *integer* vertical offset are calculated:

```
n_x0123_y0 = *n_image_position++;       // non-aligned 32-bit access
n_x4567_y0 = *n_image_position++;       // non-aligned 32-bit access
n_x89ab_y0 = *n_image_position;         // non-aligned 32-bit access

switch (x_frac) {       // Horizontal fractional interpolation.
case 0:                 // fractional offset of 0:
   break;
case 1:                 // fractional offset of 1/4:
   n_x0123_y0 = QUADAVG (n_x0123_y0,
                         QUADAVG (n_x0123_y0, FUNSHIFT1 (n_x0123_y0, n_x4567_y0)));
   n_x4567_y0 = QUADAVG (n_x4567_y0,
                         QUADAVG (n_x4567_y0, FUNSHIFT1 (n_x4567_y0, n_x89ab_y0)));
   break;
case 2:                 // fractional offset of 2/4:
   n_x0123_y0 = QUADAVG (n_x0123_y0, FUNSHIFT1 (n_x0123_y0, n_x4567_y0));
   n_x4567_y0 = QUADAVG (n_x4567_y0, FUNSHIFT1 (n_x4567_y0, n_x89ab_y0));
   break;
default:                // fractional offset of 3/4:
   n_x0123_y0 = QUADAVG (FUNSHIFT1 (n_x0123_y0, n_x4567_y0),
                         QUADAVG (n_x0123_y0, FUNSHIFT1 (n_x0123_y0, n_x4567_y0)));
   n_x4567_y0 = QUADAVG (FUNSHIFT1 (n_x4567_y0, n_x89ab_y0),
                         QUADAVG (n_x4567_y0, FUNSHIFT1 (n_x4567_y0, n_x89ab_y0)));
   break;
}
```

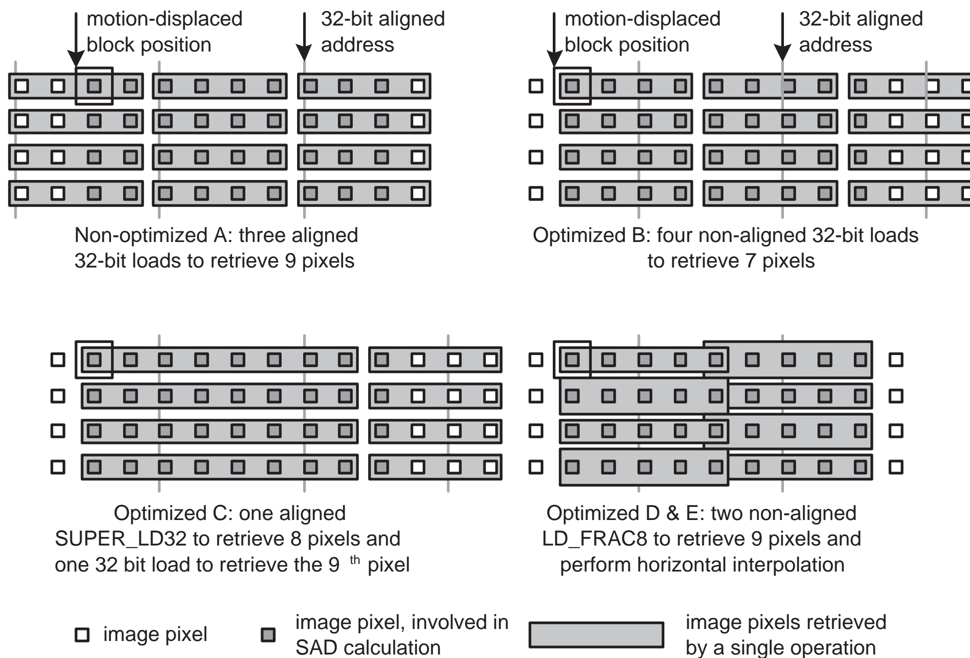

Figure 5.2: *Different implementations of the traditional block-matching kernel, only the first four rows of a 8x8 block are depicted.*

Implementation C uses the two-slot SUPER_LD32 load operation to retrieve eight horizontal neighboring pixel values. For the calculation of eight horizontal neighboring pixels in block b_n with a horizontal fractional offset, nine pixel values of $Image_n$ are required. For implementations A and B, three traditional load operations are used to retrieve the nine pixels. Implementation C, replaces two of these load operations with a single SUPER_LD32 operations. The following code sequence shows how pixel values with a *fractional* horizontal offset and an *integer* vertical offset are calculated (the SUPER_LD32 operation is in fact generated by the compiler/scheduler, rather than hand-instantiated by the programmer):

```
n_x0123_y0 = *n_image_position++;                         // non-aligned 32-bit access
SUPER_LD32 (&n_x4567_y0, &n_x89ab_y0, n_image_position);  // non-aligned 64-bit access

switch (x_frac) {       // Horizontal fractional interpolation.
case 0:                 // fractional offset of 0:
   break;
case 1:                 // fractional offset of 1/4:
   n_x0123_y0 = QUADAVG (n_x0123_y0,
                         QUADAVG (n_x0123_y0, FUNSHIFT1 (n_x0123_y0, n_x4567_y0)));
   n_x4567_y0 = QUADAVG (n_x4567_y0,
                         QUADAVG (n_x4567_y0, FUNSHIFT1 (n_x4567_y0, n_x89ab_y0)));
   break;
case 2:                 // fractional offset of 2/4:
   n_x0123_y0 = QUADAVG (n_x0123_y0, FUNSHIFT1 (n_x0123_y0, n_x4567_y0));
   n_x4567_y0 = QUADAVG (n_x4567_y0, FUNSHIFT1 (n_x4567_y0, n_x89ab_y0));
   break;
default:                // fractional offset of 3/4:
   n_x0123_y0 = QUADAVG (FUNSHIFT1 (n_x0123_y0, n_x4567_y0),
                         QUADAVG (n_x0123_y0, FUNSHIFT1 (n_x0123_y0, n_x4567_y0)));
   n_x4567_y0 = QUADAVG (FUNSHIFT1 (n_x4567_y0, n_x89ab_y0),
                         QUADAVG (n_x4567_y0, FUNSHIFT1 (n_x4567_y0, n_x89ab_y0)));
   break;
}
```

Implementation D uses the collapsed LD_FRAC8 load operation to retrieve image pixel values, whereas implementations A, B and C use traditional load and/or SUPER_LD32 operations. A single LD_FRAC8 operation is used to retrieve five pixel values and performs horizontal fractional pixel calculation on-the-fly, thereby eliminating the need for dedicated operations to calculate horizontal fractional positions. To calculate all eight horizontal fractional pixels, two of these operations are used. The following code sequence shows how pixel values with a *fractional* horizontal offset and an *integer* vertical offset are calculated:

```
x_frac = x_frac * 4;      // Calculate horizontal fractional position at 1/16 accuracy
x_frac = PACK16LSB (PACKBYTES (x_frac, x_frac), PACKBYTES (x_frac, x_frac));

n_x0123_y0 = LD_FRAC8 (n_image_position,   x_frac);
n_x4567_y0 = LD_FRAC8 (n_image_position+4, x_frac);
```

Implementation E uses the SUPER_QUADUSCALEMIXUI operation to calculate pixel values at a vertical fractional offset. The following code sequence shows how pixel values with an *integer* horizontal offset and a *fractional* vertical offset are calculated:

```
y_frac = y_frac * 16;      // Calculate vertical fractional position at 1/64 accuracy
y_frac       = PACK16LSB (PACKBYTES (y_frac, y_frac), PACKBYTES (y_frac, y_frac));
y_frac_compl = QUADSUB (0x40404040, y_frac);

SUPER_LD32 (&n_x01234_y0, &n_x4567_y0, n_image_position);              // non-aligned access
SUPER_LD32 (&n_x01234_y1, &n_x4567_y1, n_image_position+image_stride); // non-aligned access

n_x0123_y0 = SUPER_QUADUSCALEMIXUI (n_x0123_y0, y_frac_compl, n_x0123_y1, y_frac);
n_x4567_y0 = SUPER_QUADUSCALEMIXUI (n_x4567_y0, y_frac_compl, n_x4567_y1, y_frac);
```

5.2.2 Down-sampled block-matching

The two down-sampled block-matching approaches, as discussed in Section 5.1.2, reduce performance complexity by limitation of the amount of inputs to the SAD calculation, and by not supporting vertical fractional offsets (Table 5.2).

Implem.	Non-aligned	SUPER_LD32	LD_PACKFRAC8	SUPER_ QUADUSCALEMIXUI
		Usage		
F	yes	yes	hor. fract. pos.	vert. fract. pos.
G	yes	yes	hor. fract. pos.	no
		Operation count		
F	–	0	8	9
G	–	0	8	

Table 5.2: *Down-sampled block-matching implementations: the used features and new operation counts to calculate block* b_n*.*

Implementations F and G use the collapsed LD_PACKFRAC8 operation to retrieve pixel values and to perform horizontal down-sampling by a factor two. The operation is used to retrieve the pixel values for both blocks b_c and b_n (Figure 5.3). For block b_c, both implementations use 8 LD_PACKFRAC8 operations. For block b_n, implementation F uses 9 LD_PACKFRAC8 operations to allow for the calculation of vertical fractional pixel values and implementation G uses only 8 LD_PACKFRAC8 operations, as vertical fractional offsets are not supported. The

following code sequence shows how pixel values with a *fractional* horizontal offset and an *integer* vertical offset are calculated:

```
x_frac = x_frac * 4;      // Calculate horizontal fractional position at 1/16 accuracy
x_frac = PACK16LSB (PACKBYTES (x_frac, x_frac), PACKBYTES (x_frac, x_frac));

n_x01234567_y0 = LD_PACKFRAC8 (n_image_position, x_frac);
```

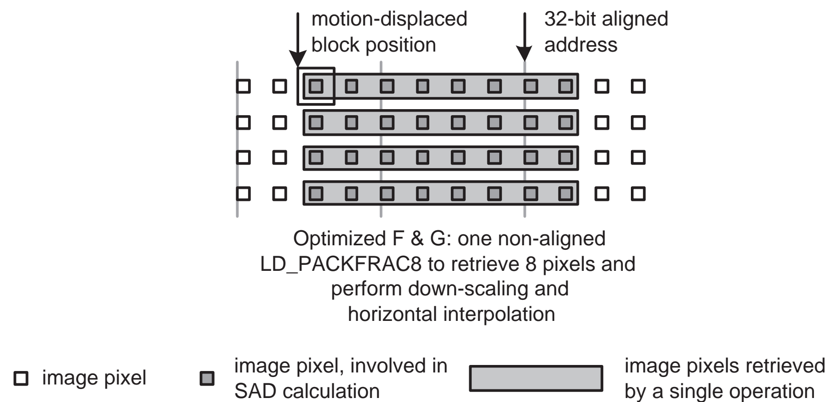

Figure 5.3: *Implementation of the down-sampled block-matching kernel, only the first four rows of a 8x8 block are depicted.*

5.2.3 Static performance complexity

The *MatchBlock* function was compiled and scheduled for the TM3270, for each of the implementations A through G. Table 5.3 gives an overview of the static performance complexity in terms of VLIW schedule lengths and number of operations.

Implem.	Quality level	Worst case VLIW schedule	Operations	Ops. / VLIW instr.
A	+	99	424	4.28
B	+	97	361	3.72
C	+	89	355	3.99
D	+	58	201	3.47
E	+	45	142	3.16
F	−	36	84	2.33
G	−−	30	59	1.97

Table 5.3: *Static performance complexity for the MatchBlock function.*

Implementations A through E provide the same functionality, implementations F and G provide similar functionality at degraded quality levels, as a result of down-sampling and removing support for vertical fractional offsets. The VLIW schedule lengths and operation counts include the overhead of retrieving function parameters from the stack.

The operations column gives an indication of the static complexity, based on the *scheduled* operations. Implementations A through C have dedicated code sequences to calculate pixel values at a specific fractional horizontal position and implementations A through D have dedicated code sequences to calculate pixel values at a specific fractional vertical position. The compiler/scheduler generates guarded operations to optimally schedule code with multiple possible execution paths. When scheduled operations have a guard value of '0', they are not *executed*.

The difference between implementations A and B shows the impact of non-aligned memory access. Implementation B uses non-aligned access: the additional operations of implementation A to align the pixel values are not required. Implementation C shows the impact of the SUPER_LD32 operation. Implementation D shows the impact of the SUPER_QUADUSCALEMIXUI operation, which is used to efficiently implement fractional pixel calculations. Implementation E shows the impact of the LD_FRAC8 operation. Overall, implementation E has a 55% shorter schedule length as implementation A, at a same quality level. At a degraded quality level, implementation F shows how a reduction in the amount of SAD input values can reduce performance complexity. Similarly, implementation G shows how removing support for vertical fractional offsets can reduce performance complexity even further. As the performance complexity of the implementations improves, as indicated by reduced VLIW schedule lengths and operation counts, the amount of operations per VLIW instruction (issue slot utilization) reduces. The reduction in issue slot utilization is explained as follows:

- The new TM3270 operations improve the ISA efficiency, calculations are performed with less operations and the amount of operation level parallelism decreases. As a result, it becomes harder to fill the five issue slots of a VLIW instruction with independent operations.

- The operation count represents the amount of *scheduled* operations, rather than the amount of *executed* operations. Implementations A through D have dedicated code sequences to calculate pixel values with different fractional offsets. The compiler/scheduler generates VLIW schedules with guarded operations, and multiple dedicated code sequences may be mapped on a single VLIW schedule based on similarity of the code sequences. A VLIW sequence

distinguishes between different code sequences using guarded execution. In the case of fractional pixel calculation the guards are dependent on the fractional offset. As a result, those operations that do not contribute to the calculation of a specific fractional offset have a guard value of '0', and are not executed.

5.3 Dynamic performance complexity

In this section we discuss the dynamic performance complexity of our 3DRS algorithm (as described in Section 5.1) for each of the seven implementations (A through G) of the *MatchBlock* function on a standard definition (720*480) NTSC video sequence. The performance is measured in our cycle-accurate performance evaluation environment (Section 1.3.2). The dynamic performance complexity includes the static complexity in terms of the amount of VLIW instructions and the execution behavior in terms of processor stall cycles. Processor stall cycles are mainly caused by data cache misses.

For implementations E, F, and G we were able to successfully inline the *Match Block* function in the motion estimation function (for implementations A, B, C and D inlining resulted in excessive spilling, effectively degrading processor performance). Inlining reduces function-call, and -return overhead. Rather than retrieving the same b_c block data for each of the 11 block matches, the block data is retrieved only once. As a result, a large amount of load operations is removed from the generated code. Furthermore, inlining provides the compiler/schedules with the operation parallelism of 11 *MatchBlock* functions. As a result, the issue slot utilization is improved when compared to the issue slot utilization for a single block match. All implementations fit in the 64 Kbyte instruction cache, so apart from initial compulsory cache misses, no instruction cache misses and associated cache stall cycles were observed.

As the 3DRS algorithm iterates over a video sequence, it tends to converge to a smooth motion vector field. Although we implemented the algorithm as presented (including the control overhead of tracking the best motion vector candidate for each block b_c), we decided not to rely on the spatial and temporal convergence of the algorithm:

- The spatial convergence is non-existent: we operate on images with randomly initialized image pixel values in the [0, 255] byte value range.
- The temporal convergence is non-existent: we initialize the temporal motion vector candidate set $CS_{temporal}$ with random motion vectors, with a horizontal component in the range [-720, $719\frac{3}{4}$] and a vertical component in the

range [-40, $39\frac{3}{4}$].

As a result, our performance evaluation results reflect worse case, rather than typical case execution behavior.

When memory region based prefetching is turned on, its settings are as follows (Figure 5.4). A first prefetch memory region includes the current image $Image_c$. The associated stride value is set to $8 * image_stride$: when processing a pixel at position $\overrightarrow{b_c}$ in $Image_c$ the pixel at position $\overrightarrow{b_c} + \begin{pmatrix} 0 \\ 8 \end{pmatrix}$ is prefetched. A second prefetch memory region includes the next image $Image_n$. The associated stride value is set to $40 * image_stride$: when processing a pixel at position $\overrightarrow{b_n}$ in $Image_n$ the pixel at position $\overrightarrow{b_n} + \begin{pmatrix} 0 \\ 40 \end{pmatrix}$ is prefetched. The value 40 is the vertical search range of the motion estimation algorithm. These settings make it likely that the image pixels for blocks b_c and b_n are found in the data cache when needed by the algorithm.

5.3.1 Comparing the implementations

The performance of implementations A through G is measured, with prefetching turned on and 0 additional memory delay cycles. This setup reflects a SoC use case scenario in which only the TM3270 processor consumes off-chip memory bandwidth. Although this is an unlikely scenario, it gives an upper bound of achievable processor performance. Table 5.4 gives an overview of the dynamic performance complexity.

Implem.	Cycles	VLIW instr.	Stall cycles	Ops.	Ops. / VLIW instr.	Cycles / VLIW instr.
A	4,753,461	4,609,434	144,027	18,631,133	4.04	1.03
B	4,596,421	4,460,227	136,194	14,295,652	3.21	1.03
C	4,153,657	3,985,028	168,629	14,310,563	3.59	1.04
D	2,921,350	2,828,541	92,809	10,309,443	3.64	1.03
E	2,135,625	2,080,539	55,086	7,764,781	3.73	1.03
F	1,895,990	1,751,987	144,003	5,534,595	3.16	1.08
G	804,555	638,792	165,763	2,865,714	4.49	1.26

Table 5.4: *Dynamic performance complexity: motion estimation results (prefetching on, 0 additional delay cycles).*

Implementations A and E show the compound impact of using non-aligned memory access and new operations: implementation A uses 4,753,461 cycles and implementation E uses 2,135,625 cycles, a reduction of 65.1%. Implementation

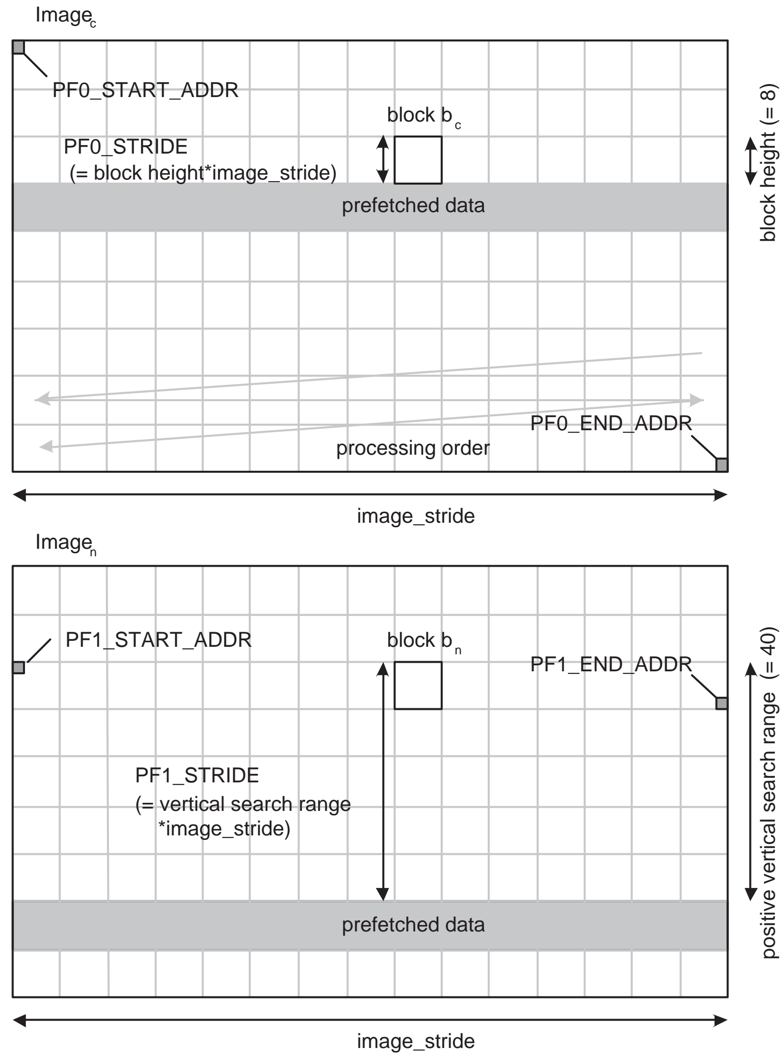

Figure 5.4: *Prefetch memory region settings for* $Image_c$ *and* $Image_n$*.*

G (with horizontal down-sampling, and without support for vertical fractional offsets) uses only 804,555 cycles, a further reduction of 62.3% over implementation E (and a reduction of 83.1% over implementation A). Implementations D through G have higher issue slot utilization for the motion estimation algorithm, than for a single block match, as a result of function inlining. All implementations have a CPI close to the theoretical optimum of 1.0: efficient prefetching limits the amount of data cache misses to a minimum. Most of the observed data cache misses were conflict misses (the two images $Image_c$ and $Image_n$, the motion vectors, and the stack compete for the four cache lines in each set of our four way set-associative data cache).

5.3.2 Memory latency

We use the delay block in our performance evaluation environment (Section 1.3.2) to measure the impact of SDRAM latency on processor performance. We measured the performance of implementations A through G with prefetching turned on and the additional memory delay cycles in the range [0, 150] (one memory delay cycle represents 2.25 processor cycles). Since the amount of VLIW instruction is unaffected by the memory latency, we focus on the amount of processor stall cycles as a function of additional memory latency (Figure 5.5).

As memory latency increases, the amount of stall cycles due to cache misses increases, which has a negative impact on processor performance. All of the implementations have linear stall cycle curves, however, the lowest complexity implementation G has a steeper slope. For the higher complexity implementations A, B, C, D, E, and F, the increase in stall cycles is a result of the larger miss penalty of data cache conflict misses. Compulsory misses hardly occur, because prefetching is overlapped with computation and data is prefetched into the cache before actual use. For implementation G, the increase in stall cycles is a result of both conflict and compulsory misses. Because of the lower computational complexity of implementation G, prefetching is no longer overlapped with computation. As a result, prefetches have turned into compulsary misses, resulting in additional stall cycles. This explains the steeper slope of the implementation G stall cycle curve.

Table 5.5 gives the dynamic performance complexity at 150 additional memory delay cycles. The cycle count difference between the down-sampling implementations F and G has become significantly smaller; the difference is 2,312,954 - 2,113,646 = 199,308 at 150 additional memory delay cycles, whereas the difference is 1,895,990 - 804,555 = 1,091,435, at 0 additional memory delay cycles, a reduction of 81.7%. This illustrates that as an application becomes more memory bound, the static performance complexity becomes less important. At 150 additional memory delay cycles, implementations F and G require a similar amount of

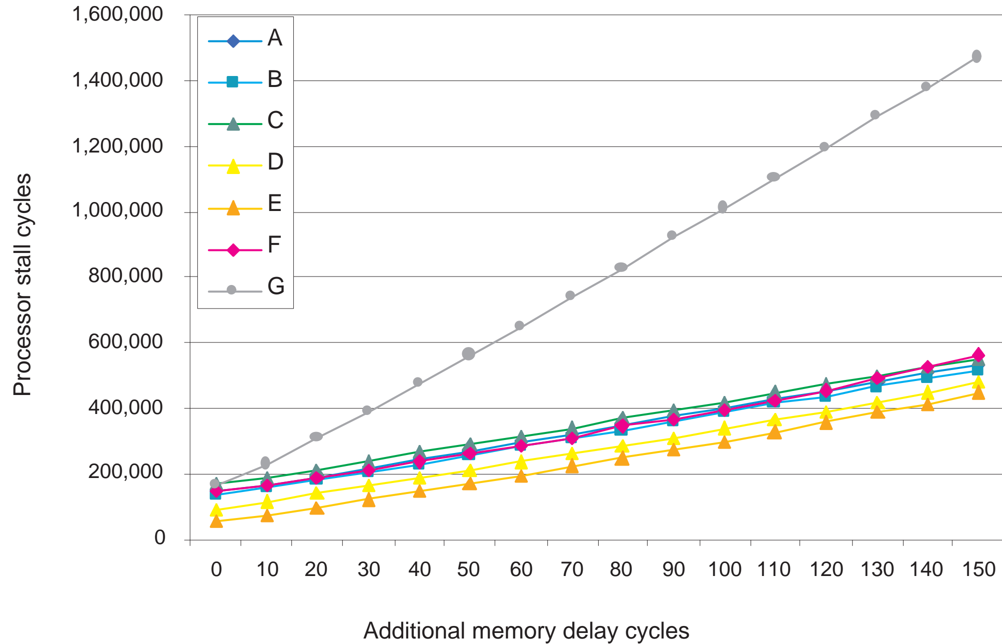

Figure 5.5: *Processor stall cycles as a function of additional memory delay cycles (prefetching on).*

cycles (2,312,954 and 2,113,646 cycles), but implementation F provides a better quality result (due to the support of vertical fractional offsets).

5.3.3 Data prefetching

Processor-performance benefits from prefetching of data from the off-chip SDRAM into the processor's data cache. When prefetching is turned off, the amount of compulsory data cache misses increases significantly. To determine the impact of prefetching, we measured the performance of implementations A through G with prefetching turned off and the additional memory delay cycles in the range [0, 150] (Figure 5.6). As memory latency increases, the amount of stall cycles increases. All stall cycle curves are similar, which is expected as all implementations operate on the same amount of input data ($Image_c$ and $Image_n$).

Especially at larger memory latencies, the benefit of prefetching is significant (Table 5.6 versus Table 5.5). When prefetching is turned off, the dynamic performance complexity of implementation E increases from 2,527,134 to 4,135,684

Implem.	Cycles	VLIW instr.	Stall cycles	Ops.	Ops. / VLIW instr.	Cycles / VLIW instr.
A	5,142,063	4,609,434	532,629	18,631,133	3.62	1.12
B	4,974,828	4,460,227	514,601	14,295,652	3.21	1.12
C	4,535,788	3,985,028	550,760	14,310,563	3.59	1.14
D	3,308,739	2,828,541	480,198	10,309,443	3.64	1.17
E	2,527,134	2,080,539	446,595	7,764,781	3.73	1.21
F	2,312,954	1,751,987	560,967	5,534,595	3.16	1.32
G	2,113,646	638,792	1,474,854	2,865,714	4.49	3.31

Table 5.5: *Dynamic performance complexity: motion estimation results (prefetching on, 150 additional delay cycles).*

Implem.	Cycles	VLIW instr.	Stall cycles	Ops.	Ops. / VLIW instr.	Cycles / VLIW instr.
A	6,844,294	4,604,391	2,239,903	18,577,172	4.03	1.49
B	6,589,782	4,460,224	2,129,558	14,284,471	3.20	1.48
C	6,138,029	3,985,025	2,153,004	14,299,382	3.59	1.54
D	4,948,272	2,828,539	2,119,733	10,346,081	3.66	1.75
E	4,135,684	2,080,536	2,055,148	7,742,859	3.72	1.99
F	3,916,556	1,751,984	2,164,572	5,512,613	3.15	2.24
G	2,722,733	638,782	2,083,951	2,881,583	4.51	4.26

Table 5.6: *Dynamic performance complexity: motion estimation results (prefetching off, 150 additional delay cycles).*

cycles, an increase of 63.6% (at 150 additional memory delay cycles).

5.4 Conclusions

The programmability of a media-processor based solution allows for flexibility in the implementation of a motion estimation algorithm. We have shown how the new operations contribute to improved performance complexity and how implementations E, F, and G trade off performance complexity and quality level of the motion estimator. Reductions in performance complexity can be achieved by applying e.g. sub-sampling: limitation of the amount of values in the SAD calculation by not using all of a block's pixel values, as applied for implementations F and G.

The use of non-aligned memory access and new operations improves the dynamic performance complexity of the motion estimator by 65.1% (Section 5.3.1, implementation A versus implementation E, prefetching on, 0 additional delay cycles). Implementation E requires 2,135,625 cycles per image; motion estimation on 30 standard definition images per second would require 64.1 MHz (14.2% of

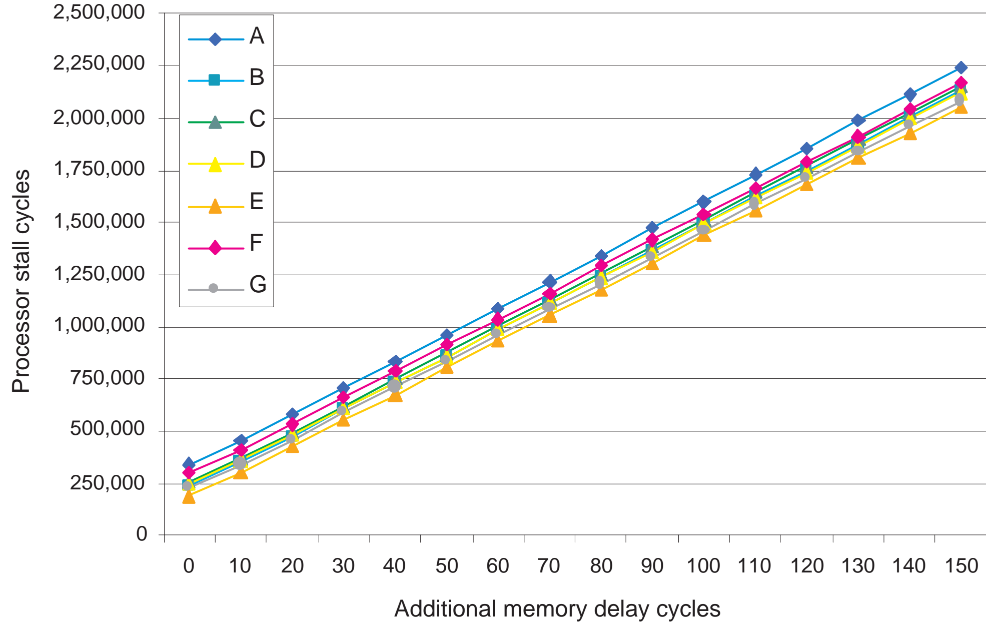

Figure 5.6: *Processor stall cycles as a function of additional memory delay cycles (prefetching off).*

the 450 MHz processor frequency). As memory latency increases, the benefit of prefetching increases: the dynamic performance complexity of implementation E is improved by 63.6% (Section 5.3.3, 150 additional memory delay cycles).

Four-way 8-bit operation	*Two-way 16-bit operation*
LD_FRAC8	LD_FRAC16
LD_PACKFRAC8	LD_PACKFRAC16
SUPER_QUADUSCALEMIXUI	SUPER_DUALISCALEMIX

Table 5.7: *Four-way 8-bit operations and their two-way 16-bit counterparts.*

The need for increased image quality has resulted in image formats with 10- and 12-bit luminance value representations. In this chapter, the motion estimation algorithm assumed 8-bit image pixel values. To accommodate the higher resolution pixel values, the TM3270 supports all new pixel processing operations with a SIMD subword size of both 8- and 16-bit. Table 5.7 lists the four-way 8-bit operations as presented in this chapter, and their two-way 16-bit counterparts. Note that

the four-way 8-bit operations process four subwords, whereas the two-way 16-bit operations only process two subwords.

Chapter 6

MPEG2 encoder

In the previous chapter we discussed TM3270 performance on a motion estimation algorithm. Motion estimation constitutes a significant amount of the computational complexity of video encoders. In this chapter we will discuss TM3270 performance on MPEG2 encoding.

Multiple video standards exist that address the compression of video to reduce transmission bandwidth and storage requirements of the compressed video signal, typically expressed in *bitrate*: the amount of bits per second of encoded video material. Currently, MPEG2 is the de facto standard for video compression. But newer standards have been introduced that enhance the techniques of MPEG2 to further reduce bitrate requirements or to improve video quality at a given bitrate. However, with the introduction of new standards, older standards do not necessarily become obsolete, since backward compatibility needs to be guaranteed to allow for playback of existing encoded video material. The requirement to support multiple standards makes a programmable platform, such as the TM3270 media-processor, an interesting solution.

Each of the video standards offers a collection of techniques to allow for efficient video compression. The video encoder has the freedom to decide *what* techniques to apply and *how* to apply them. The increasing amount of compression techniques and their interdependencies in terms of their effect on bitrate and video quality results in a combinatorial explosion of possibilities. When real-time video encoding is required, computational complexity forces us to make a decision on how to apply which compression techniques. This decision is far from trivial and drives a large research community to improve video encoder efficiency. The flexibility of a programmable implementation platform allows for fast application of the continuous improvements in encoder efficiency, resulting in a short time-to-market. Note that the video decoder does not have the encoder's freedom to decide what

techniques to apply. Its operation is determined by the compressed video signal, and it needs to support all of the standard's compression techniques.

This chapter evaluates the performance of a MPEG2 encoder on the TM3270. In Section 6.1, we give an overview of the MPEG2 encoder and identify the parts that are optimized using new TM3270 operations. In Section 6.2, we show how new operations are used to improve the MPEG2 motion estimator and discuss the static performance complexity. In Section 6.3, we show how new operations are used to improve the MPEG2 texture pipeline and discuss the static performance complexity. In Section 6.4, we briefly discuss the application of non-aligned to improve bitstream generation performance. In Section 6.5, we discuss the dynamic performance complexity of the complete MPEG2 encoder. Finally, in Section 6.6 we present a summary and conclusions. The work presented in this chapter was earlier published as [64].

6.1 Description of the algorithm

This section gives an overview of the MPEG2 encoder that was used to evaluate new TM3270 operations. The starting point was a plain-vanilla C-implementation of a MPEG2 encoder [18], and we invested 6 man weeks to optimize the implementation for the TM3270. We have not undertaken any optimizations that compromise MPEG2 compliancy. Most of the optimizations involve the use of new operations to reduce the encoder's computational complexity.

Figure 6.1 gives an overview of the MPEG2 encoder, and its functionality is summarized as follows. Macroblocks of 16x16 image pixels are processed in a left-to-right, top-to-bottom image order. First, motion estimation is performed at macroblock granularity. This function decides upon the encoding mode (intra-coded or predicted), and produces up to two motion vectors (no vectors for intra-coded macroblocks, one vector for uni-directional predicted macroblocks and two vectors for bi-directional predicted macroblocks). Next, the texture pipeline function is performed for each block of 8x8 image pixels in a macroblock. The encoder uses a 4:2:0 video format: each 16x16 macroblock has four luminance blocks and two chrominance block, resulting in a total of six blocks per macroblock. The texture pipeline consists of a sequence of kernels that perform image reconstruction based on the source image and up to two reference images, using the motion vectors a determined by the motion estimator. Furthermore, the texture pipeline produces a sequence of (run, length) pairs for every 8x8 block. The (run, length) sequence is variable length encoded and a compressed video bitstream is generated. This last step is performed after the texture pipeline has produced the (run, length) sequences for all of the six blocks in a macroblock.

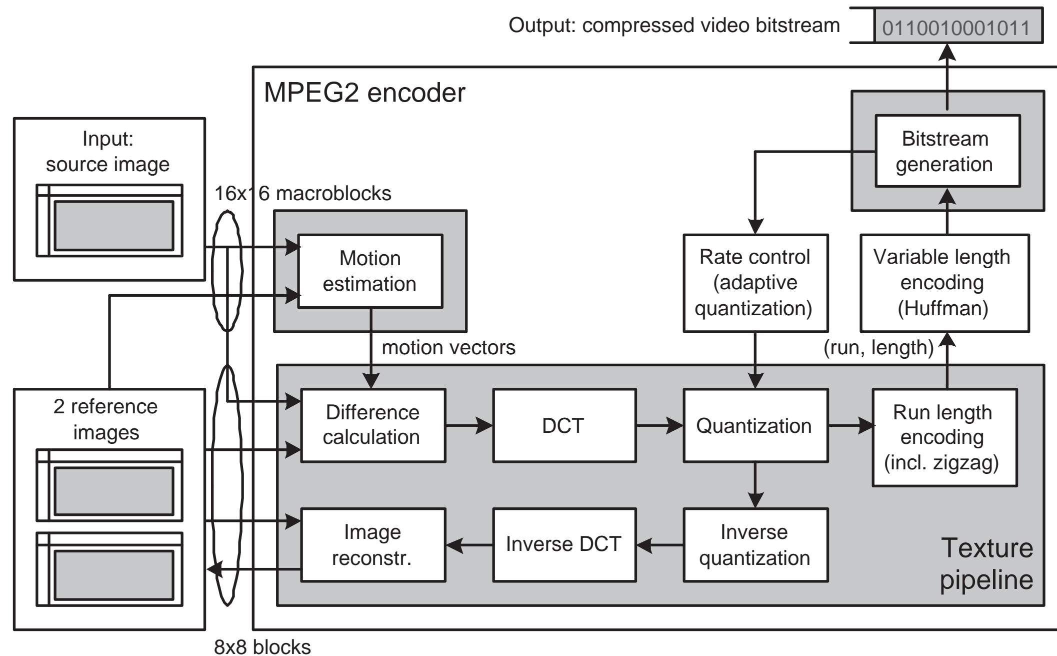

Figure 6.1: *MPEG2 encoder overview, with a breakdown of its main functions.*

6.2 Motion estimator

A significant amount of the computational complexity of a video encoder is found in the motion estimator. We decided upon the 3-D Recursive Search (3DRS) algorithm [11] to implement the motion estimator. Chapter 5 gives an example application of this algorithm for a 8x8 block based motion estimator. The motion estimator performs multiple matches, to determine the best match between a macroblock in the input image with a motion-displaced macroblock in a reference image. Most of the computational complexity of the algorithm is found in the macroblock matching function. The MPEG2 standard allows for motion vectors with $\frac{1}{2}$ pixel precision in both the horizontal and vertical direction. To reduce the computational complexity of fractional pixel calculation, we restricted the vertical motion vector component to full pixel precision (the horizontal component has a $\frac{1}{2}$ pixel precision).

6.2.1 Macroblock matching

The block-matching kernel determines the similarity of a 16x16 macroblock in the current image with motion-displaced macroblock in a reference image. The Sum-of-Absolute-Difference (SAD) cost function is used as matching criterion. For the estimator we use two optimized versions of the block-matching kernel. Version A with a relatively high computational complexity that provides a high quality matching result and version B with a relatively low computational complexity that provides a lower quality matching result. Version B uses down-scaling and sub-sampling of macroblock pixel data. As a result, the amount of inputs to the SAD calculation is significantly reduced.

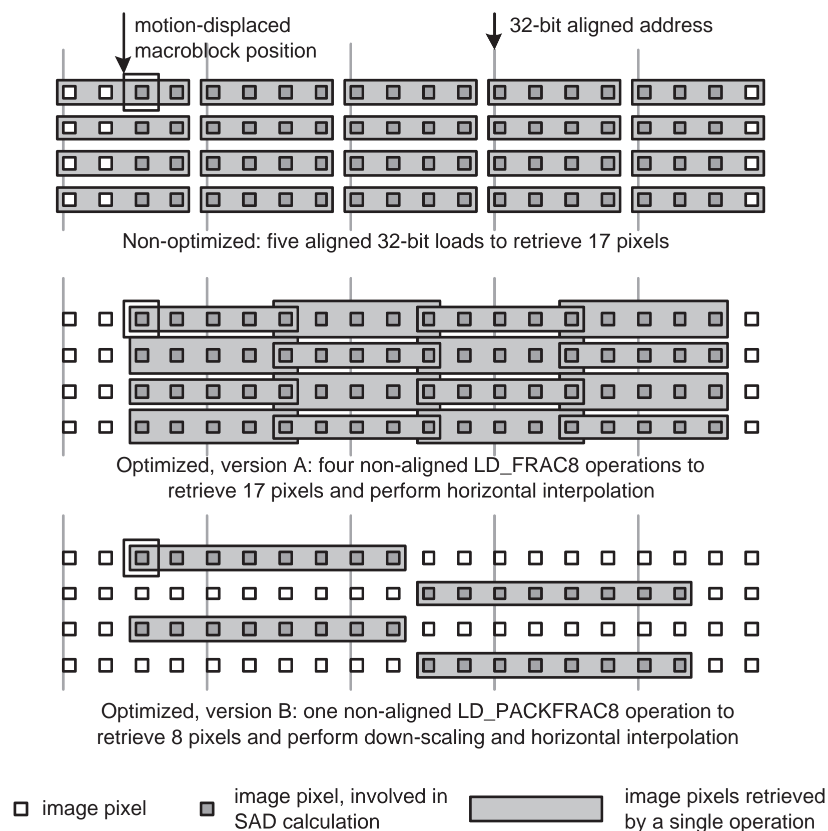

Figure 6.2: *Optimized and non-optimized (versions A and B) implementations of the block-matching kernel. Depicted is the retrieval of the first four rows of macroblock pixels from a reference image.*

Figure 6.2 illustrates three implementations of the block-matching kernel as

applied on the first four rows of a 16x16 macroblock in a reference image (one non-optimized implementation and two optimized implementations (versions A and B)). The non-optimized implementation and optimized implementation A are functionally equivalent and use all of the macroblock pixels for the SAD calculation (256 absolute differences are summed). The non-optimized implementation uses five 32-bit aligned load operations to retrieve the 17 pixels that are required to calculate the 16 horizontally interpolated pixels of a macroblock row. Optimized implementation A uses four non-aligned LD_FRAC8 operation to retrieve the 17 pixels and to calculate the 16 horizontally interpolated pixels of a macroblock row. Optimized implementation B uses only half of the macroblock pixels (sub-sampling). The optimized implementation uses a single LD_PACKFRAC8 operation to retrieve 8 pixels and to perform down-scaling and horizontal pixel calculation for a macroblock row. Sub-sampling and down-scaling result in a lower quality matching result for optimized implementation B. However the complexity of the SAD calculation is reduced by a factor of four (only 64 absolute differences are summed).

Retrieving data for the 16x16 macroblock in the *current image* is simplified when the image is located at 32-bit aligned address and when the image width is a multiple of 4 bytes, such that every image line starts a 32-bit aligned address. Given the macroblock width of 16 bytes, every upper-left pixel of a macroblock is located at a 32-bit boundary. Since no fractional horizontal pixel calculation is required for the macroblock in the current image, four aligned 32-bit load operations are used to retrieve the 16 pixels in a macroblock row in the non-optimized implementation. The optimized version A uses two SUPER_LD32R operations to retrieve the 16 pixels in a macroblock row. The optimized version B uses a single LD_PACKFRAC8 operation to retrieve 8 pixels and to perform down-scaling for a macroblock row.

Table 6.1 compares the macroblock matching computational complexity of the non-optimized implementation and the two optimized implementations (versions A and B). Optimized implementation A achieves a speedup of 1.56 with respect to the non-optimized implementation, as a result of the use of the new LD_FRAC8 and SUPER_LD32R operations and the non-aligned memory access. This speedup is achieved with the same quality matching result. At a lower quality matching result, optimized implementation B achieves a speedup of 2.32 with respect to the non-optimized implementation.

6.2.2 The estimator

The 3DRS algorithm performs multiple block matches, to determine the best match between a 16x16 macroblock in the input image and a motion-displaced

Implement.	LD_FRAC8	LD_PACK FRAC8	SUPER_ LD32R	Total ops.	VLIW instr.	Speedup
Non-opt.	0	0	0	449	169	-
Optimized (version A)	64	0	32*2	333	108	1.56
Optimized (version B)	0	32	0	169	73	2.32

Table 6.1: *Block-matching implementations. Speedup is based on VLIW instruction count with respect to the non-optimized implementation. Each SUPER_LD32R operation is counted twice, as it occupies two issue slots.*

macroblock in a reference image. The estimator's computational complexity depends on the amount of macroblock matches and the computational complexity of a single macro block match. Our implementation of the 3DRS algorithm evaluates 17 motion vector candidates for each uni-directionally predicted macroblock (P-frame). In a initial step, 12 motion vector candidates are evaluated using the optimized implementation B, as described in the previous section. The motion vectors are taken from candidate sets CS_{zero} (the zero motion vector), $CS_{spatial}$ (4 motion vectors of already processed blocks in the current image $Image_c$), $CS_{temporal}$ (5 motion vectors of blocks in the previous image $Image_p$), and $CS_{noise_spatial}$ (2 noise updated motion vectors of already processed blocks in the current image $Image_c$) (Section 5.1.1). In a second, refinement step, 5 additional motion vector candidates are evaluated using the optimized implementation A, based on the best motion vector as determined by the initial step: the best motion vector is re-evaluated using the higher quality block-matching implementation A, the best motion vector horizontally offsetted by a $\frac{1}{2}$ pixel and the best motion vector vertically offsetted by a full pixel. Figure 6.3 illustrates the motion estimator's initial and refinement steps.

The optimized block-matching kernels are function inlined in the motion estimator function, to allow for compiler/scheduler optimizations. The macroblock in the current image is only retrieved once for the 12 block matches using implementation B performed in the initial step, rather than separately for every block match. Likewise, the macroblock in the current image is only retrieved once for the 5 block matches using implementation A performed in the refinement step. Furthermore, function inlining removes function call overhead.

The motion estimator function includes the control overhead for the motion vector administration of the 3DRS algorithm and the clipping of motion vectors to ensure that they do not point outside the reference image. The clipping of motion vectors uses the new two-slot SUPER_DUALIMEDIAN operation: one of

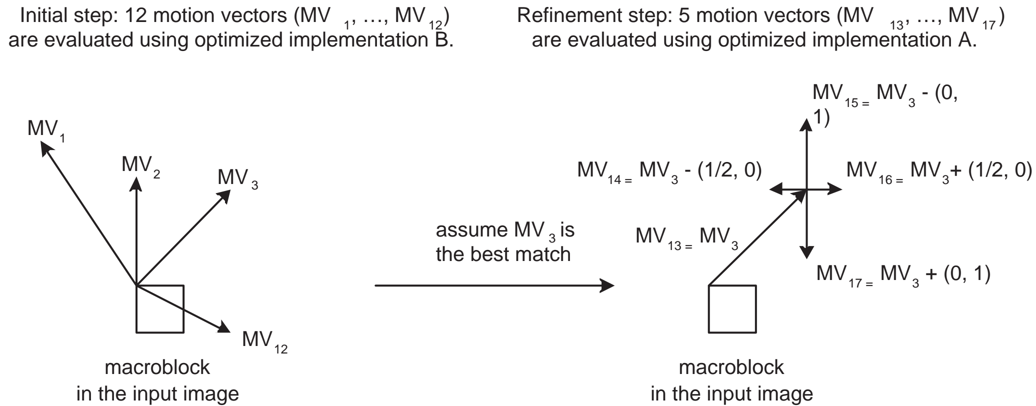

Figure 6.3: *The motion estimator for a uni-directionally predicted macroblock: in an initial step 12 motion vector candidates are evaluated using the low quality block match implementation B, in a refinement step 5 additional motion vector candidates are evaluated using the high quality block match implementation A.*

the operation's inputs is the motion vector (using a two-way 16-bit representation), and the other two inputs are the upper-left and lower right image pixel positions. Table 6.2 gives an overview of the static performance complexity of the motion estimator function. Without inlining of the block-matching functions, the VLIW schedule length is at least the sum of the block-matching functions (12 copies of implementation B and 5 copies of implementation A). Note that this schedule length is optimistic; as it does not include the actual function call overhead and the motion estimator control overhead. Inlining of the block-matching functions achieves a speedup of at least 2.48.

Implementation	VLIW instructions	Speedup
Optimized, no inlining	$\geq$ 12*73 + 5*108 = 1416	-
Optimized, inlining	570	2.48

Table 6.2: *Static performance complexity of the motion estimator function for a uni-directionally predicted macroblock, with and without inlining of the block-matching functions.*

For bi-directionally predicted macroblocks (B-frame), motion estimation is performed for two reference images. We restricted the amount of motion vectors for each reference image to 11 motion vector candidates to limit the motion estimation computational complexity. In a initial step, 6 motion vector candidates are

evaluated using the optimized implementation B. In a second, refinement step, 5 additional motion vector candidates are evaluated using the optimized implementation A, based on the best motion vector as determined by the initial step. Table 6.3 gives an overview of the static performance complexity of the motion estimator function for a single reference image. Inlining of the block-matching functions achieves a speedup of at least 2.02.

Implementation	VLIW instructions	Speedup
Optimized, no inlining	$\geq$ 6*73 + 5*108 = 953	-
Optimized, inlining	471	2.02

Table 6.3: *Static performance complexity of the motion estimator function for a bi-directionally predicted macroblock (for a single reference image), with and without inlining of the block-matching functions.*

6.3 Texture pipeline

In this section we present the static performance complexity of optimized and non-optimized versions of the texture pipeline for blocks in a bi-directionally predicted macroblock; i.e. two reference images are used. The encoder uses a 4:2:0 video format: each macroblock of 16x16 pixels has four luminance blocks and two chrominance blocks, resulting in a total of six 8x8 blocks per macroblock. The texture pipeline is performed for each of these blocks of 8x8 pixels. The texture pipeline consists of a sequence of kernels, as shown in Figure 6.4. Note that the texture pipeline performance complexity for bi-directionally predicted macroblocks is higher than that of intra-coded macroblocks and uni-directionally predicted macroblocks: intra-coded macroblocks do not require the difference calculation kernel and have a simpler image reconstruction kernel, uni-directionally predicted macroblocks have a single reference image which results in a simpler difference calculation kernel. The following subsections present the performance complexity of optimized and non-optimized versions of the individual kernels. Subsection 6.3.8 presents the performance complexity of the complete texture pipeline. Similar to the performance complexity analysis of the motion estimator, we restrict our texture pipeline analysis to motion vectors with a vertical component of full pixel precision (the horizontal component has a $\frac{1}{2}$ pixel precision). This restriction simplifies the implementation of the difference calculation kernel.

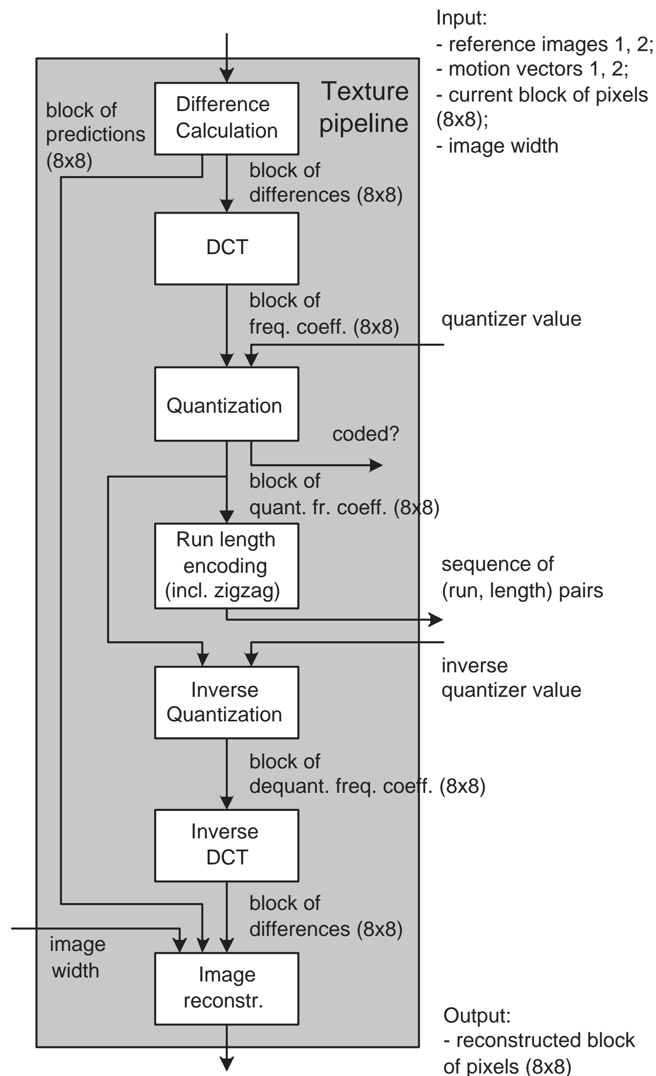

Figure 6.4: *The texture pipeline for a 8x8 block. The pipeline consists of a sequence of kernels: difference calculation, discrete cosine transform, quantization, run length encoding, inverse quantization, inverse discrete cosine transform and image reconstruction.*

6.3.1 Difference calculation

The difference calculation kernel produces a 8x8 block of unsigned 8-bit prediction values (the predicted block), based on the image pixels of two reference images at

positions indicated by the respective motion vectors. The 8x8 block of prediction values is used by the image reconstruction kernel. Furthermore, the difference calculation kernel produces an 8x8 block of signed 9-bit difference values that represent the difference between the prediction values and the image pixels of the to be encoded 8x8 block in the current image.

```
void DifferenceCalculation (
          uint8*     c_block,        // current block
          uint8*     r_image1,       // reference image 1
          uint8*     r_image2,       // reference image 2
          intdual16  c_position,     // current position in image
                                        (for motion vector offset)
          intdual16  mv1,            // motion vector for ref. image 1
          intdual16  mv2,            // motion vector for ref. image 2
          int        image_width,    // image width
          uint8*     pred_block,     // prediction block
          int16*     diff_block)     // difference block
```

The optimized implementation uses 32 collapsed LD_FRAC8 operations to retrieve the two 8x8 blocks of reference image pixels, and performs horizontal fractional pixel calculation on the fly. The non-optimized implementation uses 24 32-bit aligned load operations to retrieve the required pixels for a single reference block, resulting in a total of 48 load operations for two reference blocks: an increase in load bandwidth of 50%. The optimized implementation uses 8 two-slot SUPER_LD32R operations to retrieve the 8x8 block of current image pixels; the non-optimized implementation uses 16 32-bit aligned load operations (assuming a 32-bit alignment of the current block). Table 6.4 compares the computational complexity of the non-optimized and optimized implementations: optimization achieves a speedup of 1.26.

Implementation	LD_FRAC8	SUPER_LD32R	Total operations	VLIW instr.	Speedup
Non-opt.	0	0	306	78	-
Optimized	32	8*2	263	62	1.26

Table 6.4: *Static performance complexity of the difference calculation kernel. Each SUPER_LD32R operation is counted twice, as it occupies two issue slots.*

Intermezzo. The MPEG4 and H.264/AVC standards prescribe a non-linear multi-taps filter for the calculation of reference data at fractional positions. This prohibits the use of the LD_FRAC8 operation. However, the required interpolation is efficiently implemented with the new tow-slot SUPER_USCALEFIR8UI and SUPER_IFIR8UI operations. Given eight horizontal neighboring image pixels r_0, r_1, ..., r_7, with r_i located at horizontal position i, the H.264/AVC standard calculates p at horizontal position $3\frac{1}{2}$ using a 6-taps filter:

$$p = 2r_1 - 10r_2 + 40r_3 + 40r_4 - 10r_5 + 2r_6 + 32$$
$$p = Max(Min(p >> 6, 0), 255)$$

The MPEG4 standard calculates p at horizontal position $3\frac{1}{2}$ using a 8-taps filter:

$$\begin{aligned} p = & -2r_0 + 6r_1 - 12r_2 + 40r_3 + 40r_4 - 12r_5 + 6r_6 - 2r_7 + rounding \\ & (with\ rounding\ either\ 31\ or\ 32) \\ p = & Max(Min(p >> 6, 0), 255) \end{aligned}$$

For H.264/AVC the SUPER_USCALEFIR8UI operation performs the required calculation: filtering, rounding, scaling, and clipping to the range of an unsigned 8-bit integer. For MPEG4/AVC with a rounding value of 32, the same operation can be used. For a rounding factor of 31, the SUPER_IFIR8UI can be used to calculate the 8-taps filter, additional operations are required to perform rounding, scaling, and clipping.

6.3.2 Discrete cosine transform

The discrete cosine transform (DCT) kernel produces a 8x8 block of signed 16-bit frequency coefficients, based on the 8x8 block of difference values as produced by the difference calculation kernel.

```
void Dct (int16*     diff_block,       // difference block
          int16*     freq_coeff_block) // frequency coefficients block
```

The 8x8 two-dimensional (2D) DCT is row-column separated into 8-points 1D transforms. We use the Chen algorithm [5] for both the optimized and non-optimized implementations. The algorithm makes frequent use of butterfly and rotate operators, which are defined by:

$$\begin{aligned} Butterfly(input0, input1): \quad & output0 = input0 + input1 \\ & output1 = input0 - input1 \\ Rotate(input0, input1, cos(a), sin(a)): \quad & output0 = input0 * cos(a) - input1 * sin(a) \\ & output1 = input0 * sin(a) + input1 * cos(a) \end{aligned}$$

The new TM3270 operations allow for the calculation of the 2D DCT with signed 16-bit arithmetic. These operations use two-way 16-bit SIMD arithmetic.

As a result, two independent 1D DCTs can be calculated in parallel. The butterfly operation uses the two-way 16-bit addition and subtraction operations: DSPIDUALADD and DSPIDUALSUB (both operations are present in the original TriMedia ISA). Two independent butterfly structures are calculated in parallel using one DSPIDUALADD and one DSPIDUALSUB operation. The rotate operation uses the new two-way 16-bit SUPER_DUALISCALEMIX operation. Two of these two-slot operations calculate two independent rotate structures in parallel. Since, the operation scales its in-between result by a factor 2^{14}, the partial results of the DCT calculation can be kept within a signed 16-bit representation. Furthermore, the operation's rounding improves the accuracy of the calculation. The new DUALISCALEUI_RZ operations is similar in terms of rounding and scaling, and is used for the multiplication of partial DCT results. The compiler/scheduler keeps the working set of this kernel in registers; i.e. no spill code is generated. Table 6.5 compares the computational complexity of the non-optimized and optimized implementations: optimization achieves a speedup of 1.74.

Implement.	SUPER_ LD32R	SUPER_ DUAL ISCALEMIX	DUAL ISCALEUI_ RZ	Total ops.	VLIW instr.	Speedup
Non-opt.	0	0	0	802	179	-
Optimized	16*2	48*2	32	380	103	1.74

Table 6.5: *Static performance complexity of the discrete cosine transform kernel. Each two-slot operation is counted twice, as it occupies two issue slots.*

Intermezzo. To avoid accuracy problems, the H.264/AVC standard prescribes an integer transform for the spatial-to-frequency domain translation and vice verse [37], rather than an integer approximation of a floating point transform as used by most other video standards. It is efficiently implemented with the signed 16-bit arithmetic DSPIDUALADD and DSPIDUALSUB operations, and the new SUPER_DUALIMIX operation.

6.3.3 Quantization

The quantization kernel produces a 8x8 block of signed 16-bit quantized frequency coefficients, based on the 8x8 block of frequency coefficients as produced by the DCT kernel a quantizer value. The quantizer value is the product of the quantization matrix value and the macroblock quantizer value. For uni- and bi-directionally predicted marcoblocks, our implementation supports only a single quantization value for a complete 8x8 block; i.e. each of the block's 64 frequency coefficients is quantized with the same quantizer value. For intra-coded macroblocks, our imple-

mentation supports a separate quantizer value for each of the block's 64 frequency coefficients. As a result, the computational complexity of this kernel is higher for intra-coded blocks, than for predicted blocks. The performance numbers in this section are for predicted blocks.

```
void Quantization (
    int16*     freq_coeff_block,       // frequency coefficients block
    int        quantizer,              // quantizer value
    int16*     quant_freq_coeff_block) // quantized freq. coeff. block
```

The MPEG2 standard defines the inverse quantization (Section 6.3.5) for a predicted block by:

$$\begin{aligned} dequant_coeff &= ((2 * quant_coeff + k) * quantizer)/32 \\ quantizer &= macroblock_quantizer * matrix_quantizer \\ k &= Sign(quant_coeff) \\ &/ : division\ with\ truncation\ to\ 0. \end{aligned}$$

For the inverse quantization, an exact implementation of the above definition is required to ensure accuracy. However, for the quantization we decided upon a low complexity approximation. The approximation performs a single multiplication of the frequency coefficient with a pre-computed value based on the quantizer value. It uses the new two-way 16-bit DUALISCALEUI_RZ operation (multiplication with scaling, and rounding to zero). This operation performs the quantization of two frequency coefficients. For a total of 64 coefficients, 32 DUALISCALEUI_RZ operations are required. The non-optimized implementation provides the same functionality, but uses 32-bit arithmetic. Table 6.6 compares the computational complexity of the non-optimized and optimized implementations: optimization achieves a speedup of 3.06.

Implementation	SUPER_ LD32R	DUAL ISCALEUI_RZ	Total ops.	VLIW instr.	Speedup
Non-opt.	0	0	455	98	-
Optimized	16*2	32	131	32	3.06

Table 6.6: *Static performance complexity of the quantization kernel. Each two-slot operation is counted twice, as it occupies two issue slots.*

The quantization kernel also produces a *coded* value (Figure 6.4). This value is '1' when at least one of the quantized coefficients has a non-zero value. When this is not the case (*coded* value is '0'), a possibility exists to shorten the execution path through the texture pipeline, as described in Section 6.3.8.

6.3.4 Run length encoding

The run length encoding kernel is based on a block of quantized frequency coefficients as produced by the quantization kernel. Run length encoding checks for coefficient values of ”0”, to produce $(run, length)$ pairs of quantized coefficients run, with a $length$ value indicating the amount of preceding ”0” coefficients in zigzag order, as prescribed by the MPEG2 standard.

```
void RunLengthEncoding (
    int16*     quant_freq_coeff_block, // quantized freq. coeff. block
    int16*     run_length_pairs)       // series of (run, length) pairs
```

The kernel does not take advantage of any of the new operations: the optimized and non-optimized implementations are the same (Table 6.7).

Implementation	Total Operations	VLIW instructions	Speedup
Non-opt.	357	76	-
Optimized	357	76	1.00

Table 6.7: *Static performance complexity of the run length encoding kernel.*

6.3.5 Inverse quantization

The inverse quantization kernel produces a 8x8 block of signed 16-bit dequantized frequency coefficients, based on the 8x8 block of quantized frequency coefficients as produced by the quantization kernel and an inverse quantizer value (the same value as used as quantizer value by the quantization kernel). The quantizer value is the product of the quantization matrix value and the macroblock quantizer value. For uni- and bi-directionally predicted marcoblocks, our implementation supports only a single quantization value for each 8x8 block; i.e. each of the block’s 64 frequency coefficients is quantized with the same quantizer value. For intra-coded macroblocks, our implementation supports a separate quantizer value for each of the block’s 64 frequency coefficients. As a result, the computational complexity of this kernel is higher for intra-coded blocks, than for predicted blocks. The performance numbers in this section are for predicted blocks.

```
void Dequantization (
    int16*     quant_freq_coeff_block,   // quantized freq. coeff. block
    int        quantizer,                // (inverse) quantizer value
    int16*     dequant_freq_coeff_block) // dequantized freq. coeff. block
```

The MPEG2 standard defines the inverse quantization for a predicted block by:

$$\begin{aligned} dequant_coeff &= ((2 * quant_coeff + k) * quantizer)/32 \\ quantizer &= macroblock_quantizer * matrix_quantizer \\ k &= Sign(quant_coeff) \\ & / : division\ with\ truncation\ to\ 0. \end{aligned}$$

The optimized implementation is performed with two-way 16-bit arithmetic, and uses the new two-way 16-bit DUALADDSUB operation. This operation is used to add the sign bit to the doubled *dual_quant_coeff*:

```
(2 * dual_quant_coeff + k) = DUALADDSUB (DSPIDUALADD (dual_quant_coeff, dual_quant_coeff),
                                          0x00010001)
```

Table 6.8 compares the computational complexity of the non-optimized and optimized implementations: optimization achieves a speedup of 1.35.

Implementation	SUPER_ LD32R	DUALADDSUB	Total instr.	VLIW	Speedup
Non-opt.	0	0	333	74	-
Optimized	16*2	32	236	55	1.35

Table 6.8: *Static performance complexity of the quantization kernel. Each two-slot operation is counted twice, as it occupies two issue slots.*

6.3.6 Inverse discrete cosine transform

The inverse discrete cosine transform kernel (IDCT) produces a 8x8 block of signed 16-bit difference values, based on the 8x8 block of dequantized frequency coefficients as produced by the inverse quantization kernel.

```
void Idct (
      int16*  dequant_freq_coeff_block, // dequantized freq. coeff. block
      int16*  diff_block)               // difference block
```

The 8x8 2D IDCT is row-column separated into 8-points 1D transforms. We use a version of the Loeffler algorithm [35]. Like the forward DCT, the algorithm makes frequent use of butterfly and rotate operators. The optimized implementation is performed with two-way 16-bit arithmetic. The rounding and scaling capabilities of SUPER_DUALISCALEMIX and DUALISCALEUI_RZ allow for a

standard compliant accuracy. The compiler keeps the working set of this kernel in registers; i.e. no spill code was generated. Table 6.9 compares the computational complexity of the non-optimized and optimized implementations: optimization achieves a speedup of 1.49.

Implement.	SUPER_ LD32R	SUPER_ DUAL ISCALEMIX	DUAL_ ISCALEUI_ RZ	Total ops.	VLIW instr.	Speedup
Non-opt.	0	0	0	719	162	-
Optimized	16*2	48*2	48	397	109	1.49

Table 6.9: *Static performance complexity of the inverse discrete cosine transform kernel. Each two-slot operation is counted twice, as it occupies two issue slots.*

6.3.7 Image reconstruction kernel

The image reconstruction kernel produces a 8x8 block of unsigned 8-bit reconstructed image pixels for the current image, based on the prediction values as produced by the difference calculation kernel and based on the difference values as produced by the IDCT kernel.

```
void ImageReconstruction (
          uint8*     pred_block,      // prediction block (input)
          int16*     diff_block,      // difference block (input)
          int        image_width,     // image width
          uint8*     c_block)         // current block (output)
```

The initial steps of the image reconstruction are with 16-bit arithmetic, and the final step clips the intermediate results to an unsigned 8-bit integer range. The optimized implementation uses the SUPER_LD32R operation to retrieve the prediction and difference values. Table 6.10 compares the computational complexity of the non-optimized and optimized implementations: optimization achieves a speedup of 1.33.

Implementation	SUPER_ LD32R	Total Operations	VLIW instructions	Speedup
Non-opt.	0	190	57	-
Optimized	24*2	204	43	1.33

Table 6.10: *Static performance complexity of the image reconstruction kernel. Each two-slot operation is counted twice, as it occupies two issue slots.*

6.3.8 Putting it all together

The previous sections discussed the implementations of the individual texture pipeline kernels. Given the pipelined organization of the texture pipeline (Figure 6.4), it is possible to combine these functions into a single *TexturePipeline* function through function inlining. This gives the compiler/scheduler the opportunity to remove function-call and -return overhead and to pass in-between kernel results through registers, rather than through memory, which eliminates a large amount of load and store operations that move in-between results between the processor registers and memory. Furthermore, the compiler/scheduler is presented with more operation level parallelism, which might allow for a higher operations/VLIW instruction ratio, thereby reducing the overall VLIW schedule length.

```
void TexturePipeline (
           uint8*     c_block_in,       // current block (input)
           uint8*     r_image1,         // reference image 1
           uint8*     r_image2,         // reference image 2
           intdual16  c_position,       // current position in image
           intdual16  mv1,              // motion vector for ref. image 1
           intdual16  mv2,              // motion vector for ref. image 2
           int        image_width,      // image width
           int        quantizer,        // quantizer value
           int16*     run_length_pairs, // series of (run, length) pairs
           uint8*     c_block_out)      // current block (output)
{
   DifferenceCalculation ();
   Dct ();
   Quantization ();
   RunLengthEncoding ();
   Dequantization ();
   Idct ();
   ImageReconstruction ();
}
```

Table 6.11 repeats the computational complexity results of the optimized kernels and compares the computational complexity of two texture pipeline implementations: the first implementation sums the complexity results of the kernels and the second gives the results after function inlining. Note that the computational complexity result of the first implementation is optimistic, since it only includes the complexity results of the kernels, but not the function-call overhead of these kernels in the *TexturePipeline* function. As a result of function inlining the overall schedule length is reduced from 480 to 358 VLIW instructions, a speedup of 1.34. Function inlining is possible for both the optimized and non-optimized kernel implementations. However, the benefit for the optimized kernel implementations is higher as a result of reduced register pressure. Without the use of new operations, the register pressure of the non-optimized kernel implementations is relatively high and function inlining causes spilling, introducing a large amount of

additional load and store operations to spill and reload in-between results to and from the stack. With the use of new operation, function inlining of the optimized kernel implementations hardly causes spilling: the texture pipeline for intra-coded and uni-directionally predicted blocks do not spill, and the texture pipeline for bi-directionally predicted blocks only spills three in-between results. It is expected that optimization of the scheduler's register allocator can completely eliminate spilling for a bi-directionally predicted block.

Function	VLIW instructions	Speedup
Difference calculation	62	-
DCT	103	-
Quantization	32	-
Run length encoding	76	-
Inverse quantization	55	-
Inverse DCT	109	-
Image reconstruction	43	-
TexturePipeline, no inlining	$\geq$ 480	-
TexturePipeline, inlining	358	1.34

Table 6.11: *Static performance complexity of the texture pipeline for a bi-directionally predicted block, with and without function inlining.*

We investigated two further optimizations of the $TexturePipeline$ function:

1. Use of the *coded* value as produced by the quantization kernel to eliminate the unused part of the texture pipeline functionality.
2. Reduce the amount of encoded quantized frequency coefficients to reduce texture pipeline functionality.

The *coded* value is '1' when at least one of the quantized coefficient has a non-zero value, otherwise the *coded* is '0'. When the *coded* value is '0', it is not necessary to perform a large part of the texture pipeline: the run length encoding, inverse quantization and IDCT kernels do not need to be executed. Furthermore, the computational complexity of the image reconstruction kernel is reduced: the reconstructed 8x8 block in the current image is the predicted 8x8 block as produced by the difference calculation kernel. These so called non-coded blocks become more frequent as the target bitrate decreases. We created a texture pipeline implementation that takes advantage of the *coded* value: when the *coded* value is '0' a faster execution path through the $TexturePipeline$ function is achieved. When the *coded* value is '1', the new implementation has a somewhat longer normal execution path than the original implementation, as a result of scheduler overhead related to the conditional jump on the *coded* value. As a result,

this optimization is only effective for lower bitrate applications. This optimization is only performed for predicted macroblocks; i.e. the texture pipeline for intra-coded marcoblocks has no fast execution path, since it is unlikely that none of intra-coded block coefficients has a non-zero value. Table 6.12 gives the complexity results: the fast execution path achieves a speedup of 1.83.

Function	VLIW instructions	Speedup
TexturePipeline, inlining	358	-
TexturePipeline, inlining (*coded*)		
- fast execution path	196	1.83
- normal execution path	383	0.93
TexturePipeline, inlining (*coded*, 48 coeff.)		
- fast execution path	180	1.99
- normal execution path	339	1.06

Table 6.12: *Static performance complexity of the texture pipeline for a bi-directionally predicted block for two (incremental) optimizations: use of the coded value and reducing the amount of encoded quantized frequency coefficients.*

In [17] it is shown that for low bitrate applications, the encoding of only a small amount of quantized frequency coefficients in the low frequency domain results in limited quality degradation. By reducing the amount of frequency coefficients, the computational complexity of the DCT, quantization, run length encoding, inverse quantization, IDCT and image reconstruction kernels is reduced. We created a texture pipeline implementation that only encodes the first 48 quantized frequency coefficients in zigzag order, the 16 higher frequency domain coefficients are assumed to be "0". This optimization is only performed for predicted macroblocks; i.e. for intra-coded marcoblocks all 64 coefficients are encoded. Table 6.12 gives the complexity results of the new implementation, which takes advantage of the *coded* value, and only encodes 48 coefficients: the fast execution path achieves a speedup of 1.99.

6.4 Bitstream generation

The bitstream generator writes bit patterns to a sequential stream located in memory. It makes frequent use of the *PutBits* function, which takes a bit pattern *pattern* of a certain size *size*, and writes the pattern to the stream:

```
static uint8* OutputPtr;
static int    OutputState;             // left aligned
static int    OutputStateBitCount;

void PutBits (int pattern, int size)
// write rightmost size (0 <= size <= 25) bits of pattern to Output
{
   int mask;
   int bits_produced;

   mask                = (1 << size) - 1;

   OutputStateBitCount = OutputStateBitCount + size;
   OutputState         =   OutputState
                         | ((pattern & mask) << (32 - OutputStateBitCount));

   * (int *) OutputPtr = OutputState;          // non-aligned store

   bits_produced       = OutputStateBitCount & 0x38;

   OutputPtr           = OutputPtr + (bits_produced >> 3);
   OutputState         = OutputState << bits_produced;
   OutputStateBitCount = OutputStateBitCount & 0x7;
}
```

The global state of the *PutBits* function is captured by three variables: *Output Ptr* is a byte address to the current position in the stream, *OutputState* holds at most 7 bits that have not yet been written to the stream (in its most significant bit positions) and *OutputStateBitCount* gives the bit size of *OutputState*. The maximum size of *pattern* is 25 bits, to ensure that the combined size of *pattern* and *OutputState* does not exceed the 32-bit integer size. Note that almost all of the MPEG2 code words are 24 or less bits in size, which means that the *PutBits* function can be used to write almost all of the codewords to the bitstream. The *PutBits* function does not use any of the new operations, but it does take advantage of the TM3270's non-aligned memory access. After the bits of *pattern* and *OutputState* have been properly aligned and merged, the result is written to memory with a single non-aligned 32-bit store operation. Without support for non-aligned memory access, the 32-bit store operation to byte address *OutputPtr* would have to be replaced by a series of four 8-bit store operations:

```
   OutputPtr[0] = OutputState >> 24;
   OutputPtr[1] = OutputState >> 16;
   OutputPtr[2] = OutputState >> 8;
   OutputPtr[3] = OutputState;
```

6.5 Dynamic performance complexity

In this section we present the dynamic performance complexity of our MPEG2 encoder with the optimizations as described in the previous sections. The performance is measured in our cycle-accurate performance evaluation environment (Section 1.3.2). For those measurements for which prefetching is turned on, next-sequential cache line prefetching is performed.

We encoded the "Foreman" sequence at CIF resolution (352*288 pixels) at 25 frames per second, with a target bitrate of 500,000 bits per second. The rate control kernel (Figure 6.1) controls the bitrate through the macroblock quantizer value. The encoder uses a 4:2:0 video format: each macroblock of 16x16 pixels has four luminance blocks and two chrominance blocks, resulting in a total of six 8x8 blocks per macroblock. The group-of-pictures (GOP) encoding pattern is given by: I-B-B-P-B-B-P-B-B-P-B-B (in frame display order), and repeats itself every 12 frames. The estimator evaluates 17 motion vector candidates for each macroblock in a P-frame and 11 motion vector candidates for each macroblock in a B-frame (Section 6.2.2). The texture pipeline uses the two performance optimizations as described in Section 6.3.8: use of the *coded* value and encoding of only 48 quantized frequency coefficients. Bitstream generation uses the *PutBits* function as described in Section 6.4.

Figure 6.5 gives an overview of the dynamic performance complexity. For each frame type, a cycle breakdown into the major MPEG2 kernels is given (the "other" part collects all cycles that are not part of the major MPEG2 kernels). The average numbers are calculated based on the frame type frequencies as defined by the GOP pattern. I-frames have the lowest complexity, as they do not require motion estimation. The amount of cycles in the bitstream generation kernel is related to the amount of bits that are used to encode a certain frame type: I-frames use more bits per frame than P- and B-frames. For B-frames, motion estimation is performed for two reference images, which explains the high complexity of the motion estimator kernel. However, B-frames use the smallest amount of bits to encode a frame, which results in a low complexity of the bitstream generation kernel. On average 1.438,220 cycles are used to encode a frame (including processor stall cycles). At 25 frames per second, this results in a 35.95 MHz processor load.

To evaluate the impact of prefetching, we measured MPEG2 encoder performance with prefetching turned on and off, and for different additional memory delay cycles. Prefetching and additional memory delay cycles do not affect the amount of executed VLIW instruction, but only the amount of processor stall cycles. Figure 6.5 gives the results: at 0 additional memory delay cycles, prefetching reduces the amount of stall cycles from 190,458 to 171,618, a reduction of 9.9%. At 100 additional memory delay cycles, prefetching reduces the amount of stall

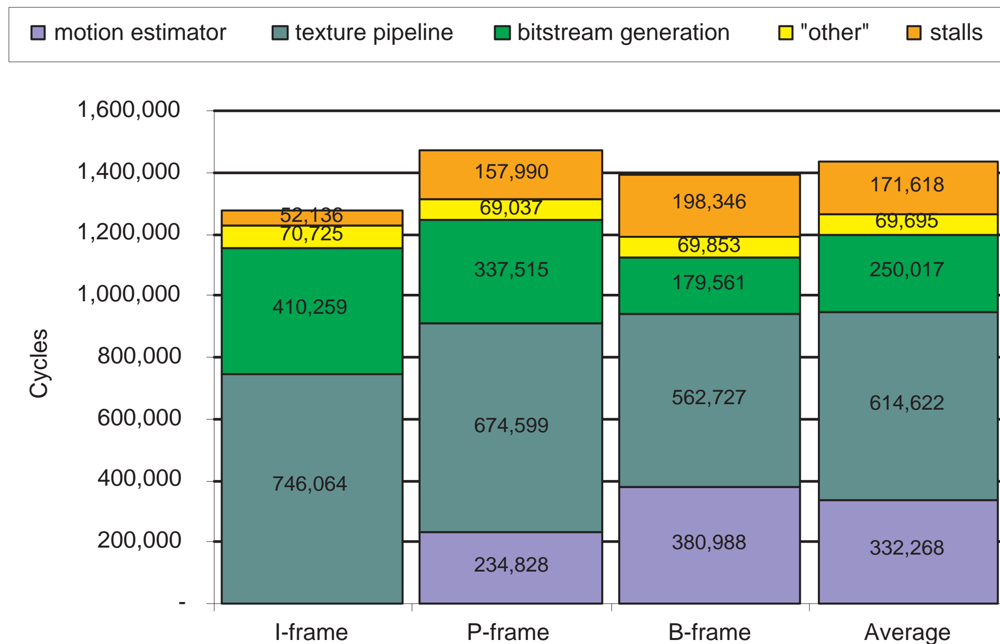

Figure 6.5: *Dynamic performance complexity of the MPEG2 encoder for I-, P- and B-frames, for prefetching on and 0 additional memory delay cycles. The average is calculated based on the GOP pattern: I-B-B-P-B-B-P-B-B-P-B-B.*

cycles from 932,718 to 557,633, a reduction of 40.2%.

6.6 Conclusions

It is not the intend of this chapter to build the best quality MPEG2 encoder, but rather to evaluate the performance improvement of new TM3270 features (new operations and non-aligned memory access) and to give an indication of achievable performance on the TM3270. When the image quality is deemed unsatisfactory, the flexibility of a programmable solution allows for changes to improve quality, e.g. different algorithms [34] could be used to implement the motion estimator.

The presented MPEG2 encoder requires 35.95 MHz to encode a CIF sequence. At an operating frequency of 450 MHz, the TM3270 media-processor is capable of encoding more than twelve CIF video sequences in parallel, an interesting feature for e.g. video observance systems in which multiple cameras can be connected to a video encoder IC with a single TM3270.

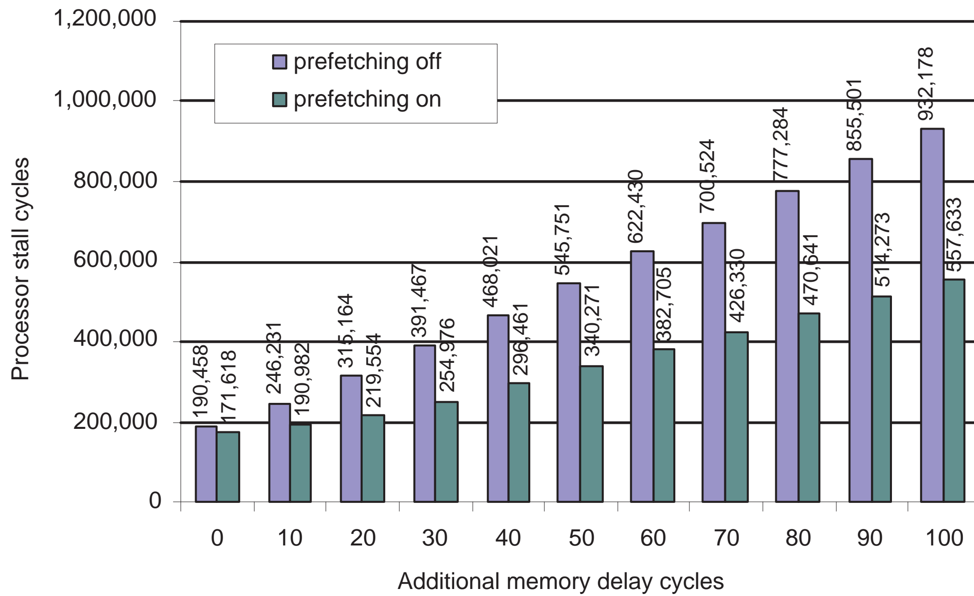

Figure 6.6: *Processor stall cycles as a function of additional memory delay cycles, for prefetching on and prefetching off.*

We have shown how that collapsed load operations significantly reduce the static performance complexity of the block-matching kernel: a speedup of 1.56 for 16x16 block match (Section 6.2.1, optimized implementation A)). Other new operations are used to speed up the texture pipeline kernels with a factor of up to 3.06 (Section 6.3.8: maximum speedup is achieved for the quantization kernel). The use of non-aligned memory access improves the performance of the bitstream generation (Section 6.4). Furthermore, we have shown that data prefetching can improve performance by 40.2% for larger off-chip SDRAM latencies (Section 6.5, 100 additional memory delay cycles).

Chapter 7

Temporal upconversion

In Chapter 5 we discussed TM3270 performance on a motion estimation algorithm. In this chapter we will discuss TM3270 performance on a motion-compensated temporal upconversion algorithm, which requires the motion vector data as produced by a motion estimation algorithm.

Temporal upconversion is a video enhancement algorithm that adapts the frequency of a video sequence to the frequency of a display device (Figure 7.1). E.g. movie video material is encoded at a progressive frame rate of 24 Hz, and typically displayed on 50 Hz display devices in Europe (PAL standard) and on 60 Hz display devices in the USA (NTSC standard). A straightforward approach to temporal upconversion is frame repetition, in Europe every movie image is repeated twice to generate a 48 Hz video sequence[1] and in the USA every movie image is repeated two or three times to generate a 60 Hz video sequence. These upconversion approaches are known as 2:2 pull down for the European flavor, and 3:2 pull down for the flavor as used in the USA. Frame repetition provides satisfactory image quality for stationary image parts, but results in jerky motion for moving image parts. These motion artifacts are known as *motion judder* or *film judder*. To prevent these artifacts, the latest of temporal upconversion algorithms apply motion-compensation techniques. Rather than repeating source images on the display device, new images are calculated using a motion vector field that intends to represent the motion of video objects.

The application of temporal upconversion is not limited to movie video material. The PAL and NTSC standards prescribe the interlaced video material at an interlaced field rate of 50 and 60 Hz respectively, resulting in a perfect match with the traditional 50 and 60 Hz display devices. However, higher frequency display

[1] 48 Hz is considered close enough to the 50 Hz of the display device, and as a result movies are shorter when viewed on a 50 Hz display device than in a movie theater.

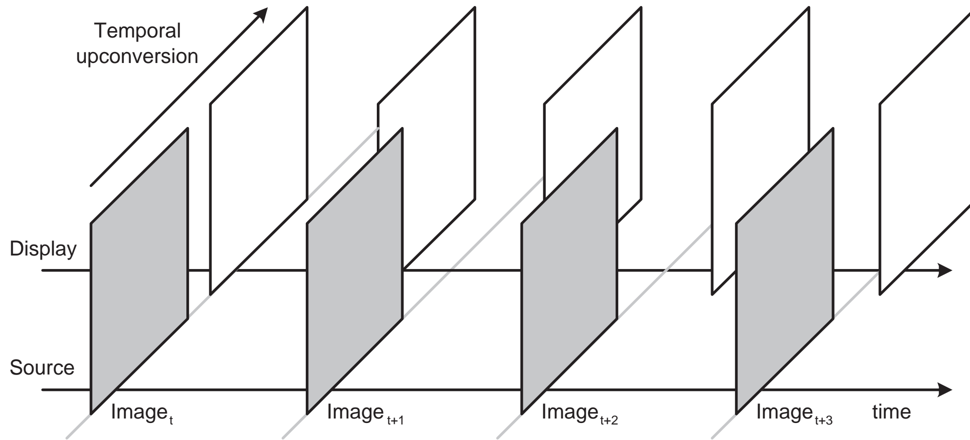

Figure 7.1: *Temporal upconversion. A sequence of images at the display frequency (white color) is calculated from a sequence of images at the source frequency.*

devices have become available with the introduction of 100 Hz CRT displays and LCD display in the television market to offer a better viewing experience.

This chapter evaluates the performance of a motion-compensated temporal upconversion algorithm on the TM3270. We quantify the contribution of new operations, data prefetching, data cache write miss policy and off-chip memory latency on processor performance. In Section 7.1, we describe the temporal upconversion algorithm. In Section 7.2, we show how new TM3270 operations are used to improve the implementations of the temporal upconversion algorithm. In Section 7.3, we discuss the dynamic performance complexity of our temporal upconversion algorithm. Finally, in Section 7.4, we present the conclusions. An earlier evaluation of the temporal upconversion on the TM3270 processor can be found in [65].

7.1 Description of the algorithm

We do not intend to introduce a new and better temporal upconverter, but rather to evaluate the performance of new TM3270-specific features on an existing algorithm. For the temporal upconverter, we decided upon an enhanced version of the cascaded median upconverter [42]. The cascaded median upconverter uses pixels of the two temporally neighboring images from the source video sequence ($Image_p$ and $Image_n$) to calculate a new upconverted image ($Image_c$) (Figure 7.2). We use a block-based implementation of the algorithm, with a block size of 4x4 image pixels. The blocks in $Image_c$ are generated in a left-to-right, top-to-bottom

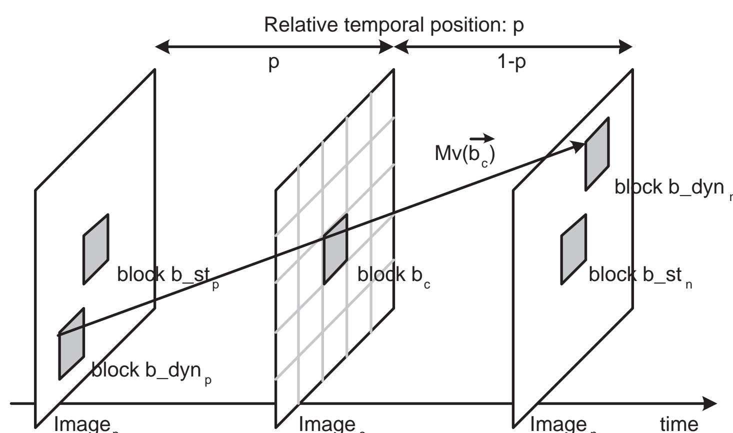

Figure 7.2: *Motion-compensated temporal upconversion. Static and dynamic pixels from the neighboring source images.*

sequence. The position of b_c in $Image_c$ is denoted by $\overrightarrow{b_c}$ such that $\overrightarrow{b_c} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and $\overrightarrow{b_c} + \begin{pmatrix} 3 \\ 3 \end{pmatrix}$ identify the upper left and lower right pixel positions of block b_c. The horizontal (X) and vertical (Y) block positions are integer multiples of 4 (the block size). We assume the availability of a motion vector $Mv(\overrightarrow{b_c})$ for every b_c in $Image_c$, which intends to represent motion from $Image_p$ to $Image_n$ through block b_c in $Image_c$. The horizontal component of a motion vector is unrestricted, the vertical component is restricted to the range [-40, $39\frac{3}{4}$] (the motion vectors $Mv(\overrightarrow{b_c})$ have a $\frac{1}{4}$ pixel precision). Motion vectors that point to blocks outside the $Image_n$ boundaries are clipped to the image boundaries.

The cascaded median upconverter uses non motion-compensated (static) and motion-compensated (dynamic) pixels of the two temporally surrounding images from the source video sequence. Static pixels are taken from colocated positions in the two temporally neighboring images: b_st_p and b_st_n. Dynamic pixels are taken from motion-compensated positions in the two temporally neighboring images: b_dyn_p and b_dyn_n, as indicated by the motion vector $Mv(\overrightarrow{b_c})$ and a fraction p that indicates the relative temporal position of $Image_c$ with respect to $Image_p$ and $Image_n$. Given the position $\overrightarrow{b_c}$, the positions $\overrightarrow{b_dyn_p}$ and $\overrightarrow{b_dyn_n}$ are calculated as follows:

$$\overrightarrow{b_dyn_p} = \overrightarrow{b_c} - pMv(\overrightarrow{b_c}) = \begin{pmatrix} b_c[x] - pMv(\overrightarrow{b_c})[x] \\ b_c[y] - pMv(\overrightarrow{b_c})[y] \end{pmatrix}$$

$$\overrightarrow{b_dyn_n} = \overrightarrow{b_c} + (1-p)Mv(\overrightarrow{b_c}) = \begin{pmatrix} b_c[x] + (1-p)Mv(\overrightarrow{b_c})[x] \\ b_c[y] + (1-p)Mv(\overrightarrow{b_c})[y] \end{pmatrix}$$

The positions $\overrightarrow{b_dyn_p}$ and $\overrightarrow{b_dyn_p}$ are calculated at a $\frac{1}{4}$ pixel precision. Image pixel values are represented by a single byte[2]. For a block b, the integer position $\overrightarrow{int_b}$ is defined by $\lfloor \overrightarrow{b} \rfloor$, and the fractional offset $\overrightarrow{frac_b}$ is defined by $\overrightarrow{b} - \overrightarrow{int_b}$. At an integer position $\overrightarrow{b}$ the value of an image pixel is defined by $Image_n[\overrightarrow{b}]$, at a fractional position $\overrightarrow{b}$ the value is calculated using bi-linear interpolation:

$$\begin{aligned} Image_n[\overrightarrow{b}] = (\quad & (1-frac_b[x])(1-frac_b[y]) && Image_n[\overrightarrow{int_b}] \\ & +frac_b[x](1-frac_b[y]) && Image_n[\overrightarrow{int_b} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}] \\ & +(1-frac_b[x])frac_b[y] && Image_n[\overrightarrow{int_b} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}] \\ & +frac_b[x]frac_b[y] && Image_n[\overrightarrow{int_b} + \begin{pmatrix} 1 \\ 1 \end{pmatrix}] + 2)/4 \end{aligned}$$

The calculation of a bi-linear interpolated 4x4 block at a horizontal and vertical fractional position requires a 5x5 block of pixel values.

The cascaded median upconverter combines a *static median*, a *dynamic median* and a *mixer* (Figure 7.3). This combination takes advantage of the individual filters' strengths and reduces the impact of the individual filters' weaknesses. A detailed description of the algorithm can be found in [42]. We further enhanced the quality of the cascaded median upconverter by the addition of a *fader*.

The *static average* and *dynamic average* functions calculate a weighted average based on the temporal position p. For a small value of p, the pixels from $Image_p$ have a relatively large weight, and the pixels from $Image_n$ have a relatively small weight.

$$Average(input1, input2, p) = (1-p) * input1 + p * input2$$

The *mixer* function calculates a weighted average based on a smoothness factor s. The upconversion algorithm calculates the smoothness factor based on the current block's motion vector $Mv(\overrightarrow{b_c})$ and the motion vectors of spatially surrounding

[2]Video images are typically represented by three values: one luminance value, and two chrominance values. For the sake of simplicity, we assume that temporal upconversion is performed on the luminance value only.

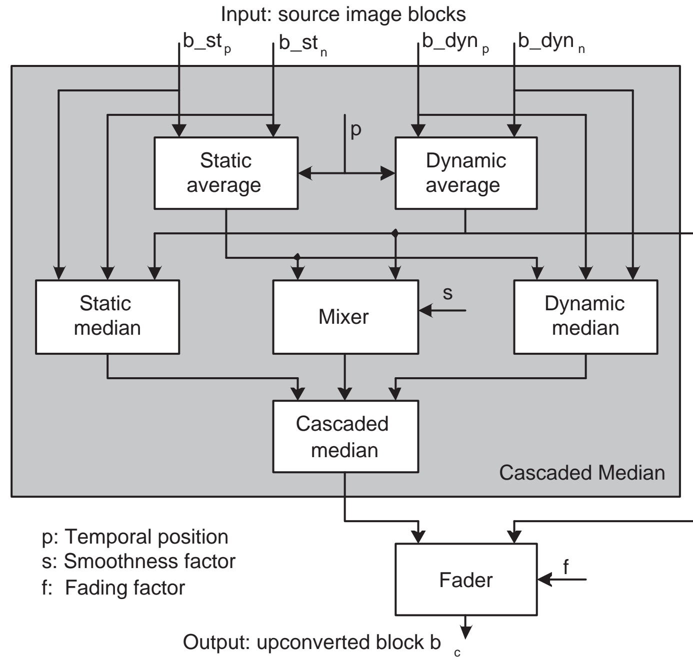

Figure 7.3: *Enhanced cascaded median upconverter.*

blocks. The smaller the sum of the differences between the current motion vector and the surrounding motion vectors, the higher the relative weight of the dynamic average input to the mixer. The smoothness factor represents the local consistency/smoothness of the motion vector field.

$$Mixer(input1, input2, s) = (1 - s) * input1 + s * input2$$

The *fader* function calculates a weighted average based on a fading factor f. The upconversion algorithm calculates the fading factor based on the current block's motion vector $Mv(\overrightarrow{b_c})$, and the motion vectors of spatially surrounding blocks. A large difference between the current motion vector and one of the surrounding motion vectors (a large block-to-block motion vector field discontinuity), results in a small relative weight for the dynamic average input to the fader.

$$Fader(input1, input2, f) = (1 - f) * input1 + f * input2$$

The *median* function calculates a three input median:

$$Median(input1, input2, input3) =$$

$$Min(Max(Min(input1, input2), input3), Max(input1, input2))$$

7.2 Implementation

In this section we show how new TM3270 operations are used to efficiently implement the temporal upconversion algorithm as described in the previous section. The algorithm generates the individual 4x4 blocks in $Image_c$ in a left-to-right, top-to-bottom sequence, using the *CalculateBlock* function, which calculates a single block in $Image_c$. We assume that image pixel values are represented by a single byte; i.e. 8 bits. The C-like interface of this function is given by:

```
void CalculateBlock (
          uint8*     p_image,       // previous image
          uint8*     c_image,       // current image
          uint8*     n_image,       // next image
          intdual16  position_curr, // current position in image
          intdual16  position_max,  // maximum position in image
          intdual16  mv,            // motion vector
          int        image_stride,  // image stride/width
          int        temporal,      // temporal position
          int        smoothness,    // smoothness factor
          int        fading)        // fading factor
```

7.2.1 Six implementations

We created six different implementations of the *CalculateBlock* function. The implementations provide the same functionality, but differ in the extent to which they use TM3270 features: non-aligned memory access and new operations (Table 7.1).

Implementation A is the reference implementation: it does not use any of the new features. Implementations B, C, D, E, and F gradually apply new TM3270 features to improve performance. Whereas implementation A has dedicated operations to properly align pixel values before fractional pixel calculation can commence, implementation B uses non-aligned load operations to retrieve pixel values with the proper alignment, thereby eliminating the need for dedicated alignment operations. Implementation C uses the collapsed LD_FRAC8 operation. For the calculation of four horizontal neighboring pixels with a horizontal fractional offset, five pixel values from an image are required. For implementations A and B, two traditional load operations are used to retrieve the five pixel values. Implementation C uses a single LD_FRAC8 load operation to retrieve the five pixel values and performs horizontal fractional pixel calculation on-the-fly, thereby reducing the amount of load operations and eliminating the need for dedicated operations to calculate horizontal fractional positions. Implementation D uses the

Implem.	Non -aligned	LD_FRAC8	SUPER_ QUADUSCALEMIXUI	SUPER_ QUADUMEDIAN
		Usage		
A	no	no	no	no
B	yes	no	no	no
C	yes	hor. fract. pos.	no	no
D	yes	hor. fract. pos.	vert. fract. pos.	no
E	yes	hor. fract. pos.	vert. fract. pos. average, mixer, and fader	no
F	yes	hor. fract. pos.	vert. fract. pos. average, mixer, and fader	median filter
		Operation count		
A	–	0	0	0
B	–	0	0	0
C	–	4	0	0
D	–	4	8	0
E	–	4	24	0
F	–	4	24	12

Table 7.1: *Six different implementations of the CalculateBlock function: the used features and the operation counts of new operations.*

two-slot SUPER_QUADUSCALEMIXUI operation to efficiently implement the vertical fractional pixel calculation. A single SUPER_QUADUSCALEMIXUI operation calculates four vertical fractional pixels. Implementation E extends the use of the SUPER_QUADUSCALEMIXUI operation. It uses the operation for vertical fractional pixel calculation and to efficiently implement the weighted average operation as required by the *average*, *mixer* and *fader* functions. Implementation F uses the two-slot SUPER_QUADUMEDIAN operation to efficiently implement a 3-taps median operation as required by the *median* function. Without this new operation, the *median* function is implemented by using the traditional QUADUMIN and QUADUMAX operations:

```
Median (input1, input2, input3) = QUADUMIN (QUADUMAX (QUADUMIN (input1, input2),
                                                      input3),
                                            QUADUMAX (input1, input2))
```

The traditional implementation uses two QUADUMIN and two QUADUMAX operations, resulting in a total of four used issue slots. Both operations have a latency of two cycles, resulting in a compound latency of six cycles for the *median* function. The SUPER_QUADUSCALEMIXUI implementation requires only two issue slots and the compound latency is reduced to two cycles.

The use of new operations to more efficiently implement a certain functionality has the obvious advantage of a reduction in the amount of required operations. As

a result, the functionality is performed at a lower static performance complexity. An additional advantage is the reduction of register pressure. An implementation should preferably keep all intermediate results of the calculation in processor registers (this eliminates the need for spilling of register values to and from the stack, which requires additional load and store operations). As new operations perform larger parts of the calculation, less processor registers are required to hold intermediate results, which reduces the register pressure. E.g. the traditional 3-taps median implementation has three intermediate results, whereas the SUPER_QUADUSCALEMIXUI implementation has no intermediate results.

7.2.2 Static performance complexity

The *CalculateBlock* function was compiled and scheduled for the TM3270, for each of the implementations A through F. Table 7.2 gives an overview of the static performance complexity in terms of VLIW schedule length and number of operations. The VLIW schedule lengths and operation counts include the overhead of retrieving function parameters from the stack.

Implem.	VLIW schedule	Operations	Ops. / VLIW instr.
A	92	375	4.08
B	81	320	3.95
C	65	244	3.75
D	62	202	3.26
E	59	186	3.15
F	60	162	2.70

Table 7.2: *Static performance complexity for the CalculateBlock function.*

The differences between implementations A and B represent the impact of non-aligned memory access. Implementation A has dedicated operations to properly align pixel values before fractional pixel calculation can commence, implementations B does not need these operations. Implementation F uses all of the new TM3270 features and shows a significant reduction in terms of operations. However, its VLIW schedule length is larger than that of implementation E. This is explained by the infancy of the current scheduler: the assignment of two-slot operations to issue slots is more complicated than the assignment of single slot operations (this limitation is addressed in later versions of the scheduler). As the performance complexity of the implementations improves, as indicated by reduced VLIW schedule lengths and operation counts, the amount of operations per VLIW instruction (issue slot utilization) reduces. The reduction in issue slot utilization is explained by the improved ISA efficiency: the use of new TM3270 operations

reduces the amount of operations and it becomes harder to fill the five issue slots of a VLIW instruction with independent operations.

7.3 Dynamic performance complexity

In this section we discuss the dynamic performance complexity of our temporal upconversion algorithm (as described in Section 7.1) for each of the six implementations (A through F) of the *CalculateBlock* function on a standard definition (720*480) NTSC video sequence. The performance is measured in our cycle-accurate performance evaluation environment (Section 1.3.2). The dynamic performance complexity includes the static complexity in terms of the amount of issued VLIW instructions and the execution behavior in terms of processor stall cycles. Processor stall cycles are mainly caused by data cache misses.

The compiler/scheduler was able to successfully inline the *CalculateBlock* function in the temporal upconversion function for all of the implementations. Inlining reduces function-call, and -return overhead. Furthermore, the inner loop of the temporal upconversion function, which iterates over the blocks b_c, was unrolled four times. This provides the compiler/scheduler with the operation parallelism of 4 *CalculateBlock* functions, at the cost of an increased code footprint. All implementations fit in the 64 Kbyte instruction cache, so apart from initial compulsory instruction cache misses, no instruction cache misses and associated cache stall cycles were observed.

Since video object sizes typically exceed our 4x4 block size, object movement should be represented by a certain consistency in the motion vector field of the blocks that cover the object. Therefore, motion-compensated dynamic block references b_dyn_p and b_dyn_n for neighboring blocks b_c exhibit high spatial locality. We decided not to rely on this locality, and use a motion vector field with random motion vectors, with a horizontal component in the range [-720, $719\frac{3}{4}$] and a vertical component in the range [-40, $39\frac{3}{4}$]. As a result, our performance evaluation results reflect worse case, rather than typical case execution behavior.

When memory region based prefetching is turned on, its settings are dependent on the temporal position p and the vertical motion vector search range. Figure 7.2 shows that as the temporal position p becomes smaller, the displacement between blocks b_dyn_p and b_c becomes smaller and the displacement between blocks b_dyn_n and b_c becomes larger. For a restricted vertical search range of [-40, $39\frac{3}{4}$] and a temporal position p the maximum vertical displacement between blocks b_dyn_p and b_c is given by $40 * p$. A first prefetch memory region includes the previous image $Image_p$. The associated stride value is set to $40{*}p{*}image_stride$: when processing

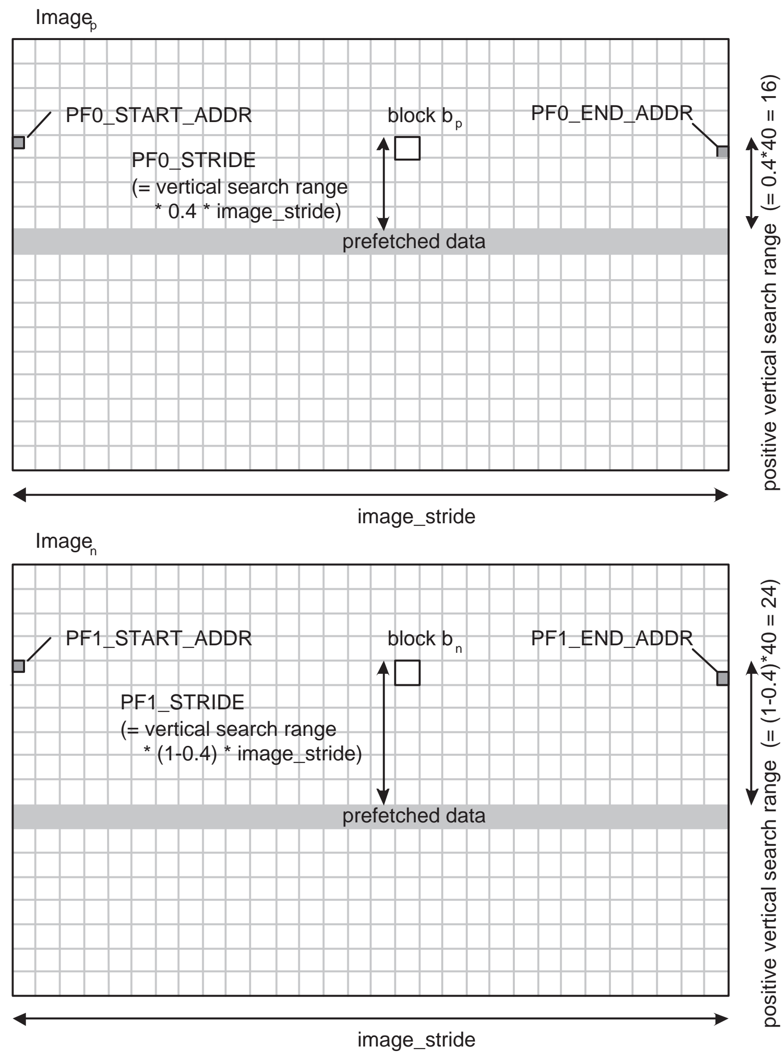

Figure 7.4: *Prefetch memory region settings for* $Image_p$ *and* $Image_n$*, for a temporal position p of 0.4.*

a pixel at position $\overrightarrow{b_p}$ in $Image_p$ the pixel at position $\overrightarrow{b_p} + \begin{pmatrix} 0 \\ 40 * p \end{pmatrix}$ is prefetched (Figure 7.4). A second prefetch memory region includes the next image $Image_n$. The associated stride value is set to $40 * (1 - p) * image_stride$: when processing a pixel at position $\overrightarrow{b_n}$ in $Image_n$ the pixel at position $\overrightarrow{b_n} + \begin{pmatrix} 0 \\ 40 * (1 - p) \end{pmatrix}$ is prefetched. These settings make it likely that the image pixels for blocks b_dyn_p and b_dyn_n are found in the data cache when needed by the algorithm. $Image_c$ is created by the temporal upconversion algorithm. When the allocate on write miss policy of the data cache is enabled, no prefetching is required for this image. When the allocate on write miss policy is disabled, and a fetch on write miss policy is used, a third prefetch memory region is used that includes $Image_c$. The associated stride value is set to $4 * image_stride$: when processing a pixel at position $\overrightarrow{b_c}$ in $Image_c$ the pixel at position $\overrightarrow{b_c} + \begin{pmatrix} 0 \\ 4 \end{pmatrix}$ is prefetched (Figure 7.5).

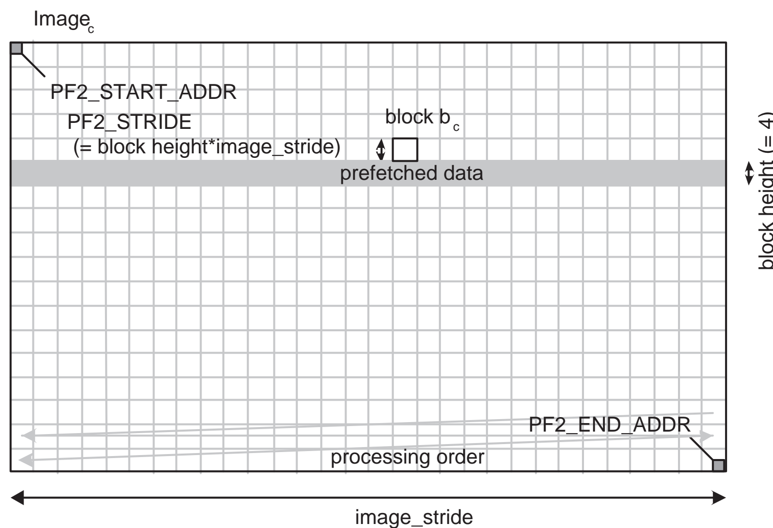

Figure 7.5: *Prefetch memory region settings for $Image_c$ (when a fetch on write miss policy is used).*

7.3.1 Comparing the implementations

The performance of implementations A through F is measured, with prefetching turned on, an allocate on write miss policy and 0 additional memory delay cycles.

This setup reflects a SoC use case scenario in which only the TM3270 processor consumes off-chip memory bandwidth. Table 7.3 gives an overview of the dynamic performance complexity.

Implem.	Cycles	VLIW instr.	Stall cycles	Ops.	Ops. / VLIW instr.	Cycles / VLIW instr.
A	1,982,854	1,918,957	63,897	8,622,483	4.49	1.03
B	1,736,874	1,670,314	66,560	7,423,559	4.44	1.04
C	1,290,174	1,222,480	67,694	5,658,964	4.63	1.06
D	1,086,778	1,017,399	69,379	4,843,324	4.76	1.07
E	1,025,611	957,161	68,450	4,571,643	4.78	1.07
F	953,825	886,829	66,996	4,161,819	4.69	1.08

Table 7.3: *Dynamic performance complexity: temporal upconversion results (prefetching on, allocate on write miss policy, 0 additional delay cycles).*

The implementations have a cycles / VLIW instruction ratio (CPI) in the range of [1.03, 1.08], which is close to the theoretical optimum of 1.0. This reflects the efficiency of prefetching: image data has been prefetched from the off-chip memory into the data cache before the actual use of the data by the algorithm. The implementations have a operations / VLIW instruction ratio (OPI) in the range of [4.44, 4.78], which is close to the theoretical optimum of 5.0 (the TM3270 has five issue slots). This reflects the high amount of operation level parallelism that is available in the algorithm as a result of inlining the *CalculateBlock* function, the loop unrolling and the scheduler's ability to exploit this parallelism. Whereas the OPI of implementations D, E, and F is relatively low (Table 7.2: 3.26, 3.15 and 2.70 respectively) at the *CalculateBlock* function level, the OPI at the temporal upconversion function level is significantly higher (Table 7.3: 4.76, 4.78, and 4.69 respectively). The use of non-aligned memory access, without any further optimizations in terms of new operations, reduces the dynamic performance complexity from 1,982,854 cycles for implementation A to 1,736,874 cycles for implementation B, a reduction of 12.4%. The use of both non-aligned memory access and the new operations, reduces the complexity from 1,982,854 cycles for implementation A to 953,825 cycles for implementations F, a reduction of 51.9%.

7.3.2 Memory latency

We use the delay block in our performance evaluation environment (Section 1.3.2) to measure the impact of SDRAM latency on processor performance. We measured the performance of implementations A through F with prefetching turned on, an allocate on write miss policy and the additional memory delay cycles in the range [0, 150] (one memory delay cycle represents 2.25 processor cycles). Since the

amount of VLIW instruction is unaffected by the memory latency, we focus on the amount of processor stall cycles as a function of additional memory latency (Figure 7.6).

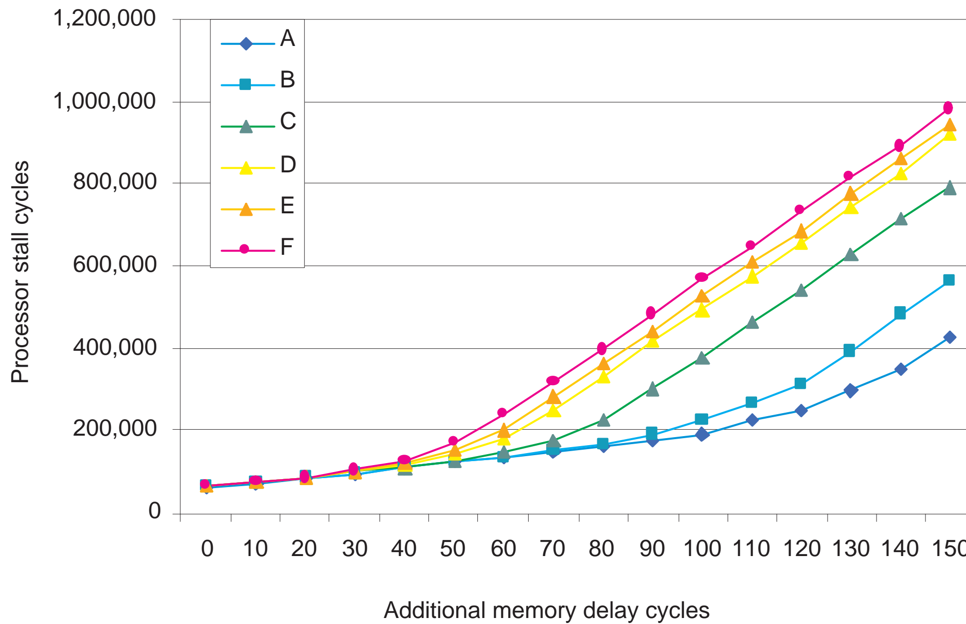

Figure 7.6: *Processor stall cycles as a function of additional memory delay cycles (prefetching on, allocate on write miss policy).*

As memory latency increases, the amount of stall cycles due to data cache misses increases, which has a negative impact on processor performance. All of the stall cycle curves have a discontinuity in their slope. The lower complexity implementations reach the discontinuity at less additional memory delay cycles, than the higher complexity implementations. We distinguish the slopes at the left and the right side of the discontinuity.

At the left side of the discontinuity, the implementations are compute bound; i.e. performance is *mainly* dependent on the amount of VLIW instructions. The stall cycle increase is a result of the increase of stall cycles due to a *limited* amount of data cache conflict misses. The implementations have a similar memory reference pattern and a similar conflict miss pattern, which explains the similarity of the stall cycles curves left of the discontinuities.

At the right side of the discontinuity, the implementations gradually become

more memory bound; i.e. performance becomes dependent on the availability of data in the data cache. This availability of data is heavily dependent on the efficiency of prefetching. As the memory latency increases, the efficiency of prefetching decreases, resulting in more compulsory misses. This decrease in prefetching efficiency becomes apparent when the time to prefetch data into the data cache exceeds the time to apply the required operations on the data. At this point, prefetches change into compulsory misses. This change is a gradual process. Prefetches start in time, but are not able to complete in time: new compulsory misses are typically on cache lines with prefetches in progress. The lower complexity implementations require less operations and are faced earlier by a decrease in prefetch efficiency, than the higher complexity implementations, which explains why the lower complexity implementation reach the discontinuity at less additional memory delay cycles.

Implem.	Cycles	VLIW instr.	Stall cycles	Ops.	Ops. / VLIW instr.	Cycles / VLIW instr.
A	2,347,309	1,918,957	428,352	8,622,483	4.49	1.22
B	2,234,374	1,670,314	564,060	7,423,559	4.44	1.34
C	2,016,461	1,222,480	793,981	5,658,964	4.63	1.65
D	1,937,042	1,017,399	919,643	4,843,324	4.76	1.90
E	1,900,875	957,161	943,714	4,571,643	4.78	1.99
F	1,868,948	886,829	982,119	4,161,819	4.69	2.11

Table 7.4: *Dynamic performance complexity: temporal upconversion results (prefetching on, allocate on write miss policy, 150 additional delay cycles).*

At 150 additional memory delay cycles, prefetching efficiency has significantly decreased, Table 7.4 shows the dynamic performance complexity. Compared to the dynamic performance complexity at 0 additional memory delay cycles (Table 7.3), the cycle count differences between the implementations have become smaller. This illustrates the general principle that implementation optimization based on static performance complexity has less impact when the implementations become more memory bound.

7.3.3 Data prefetching

To determine the impact of prefetching on processor performance, we measured the performance of implementations A through F with prefetching turned off, an allocate on write miss policy and the additional memory delay cycles in the range [0, 150] (Figure 7.7).

As memory latency increases, the amount of stall cycles due to data cache misses increases, which has a negative impact on processor performance. Data

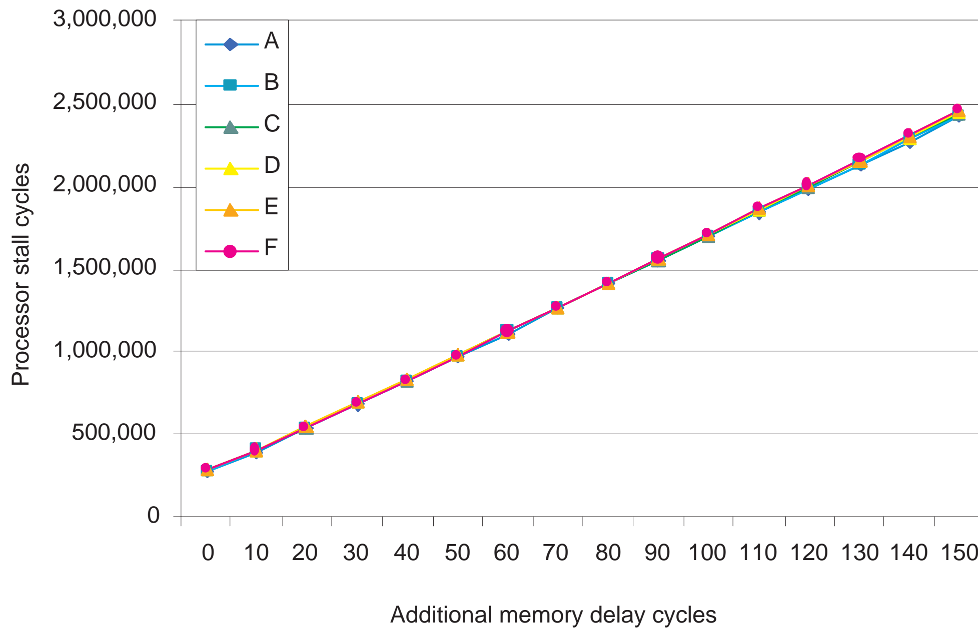

Figure 7.7: *Processor stall cycles as a function of additional memory delay cycles (prefetching off, allocate on write miss policy).*

cache misses include both compulsory and conflict misses, the working set of the implementations fit in the 128 Kbyte data sache, so no capacity misses were observed. The implementations have a similar memory reference pattern and a similar miss pattern, which explains the similarity of the stall cycles curves.

Although the implementations have become memory bound at higher SDRAM latencies, prefetching can still significantly contribute to processor performance. At 150 additional memory delay cycles, implementation F takes 3,348,267 cycles without prefetching and 1,868,948 cycles with prefetching (Table 7.5), a reduction of 44.2%. This significant improvement is explained as follows: both the data cache refill and prefetch unit can have a single outstanding bus transaction to the off-chip memory. Especially for higher memory latencies, the ability to have two, rather than one, outstanding bus transactions significantly improves processor performance.

Implem.	Cycles	VLIW instr.	Stall cycles	Ops.	Ops. / VLIW instr.	Cycles / VLIW instr.
A	4,340,223	1,918,223	2,422,000	8,616,429	4.49	2.26
B	4,098,852	1,669,821	2,429,031	7,412,224	4.44	2.45
C	3,664,106	1,222,223	2,441,883	5,657,830	4.63	3.00
D	3,466,666	1,017,143	2,449,523	4,826,470	4.75	3.41
E	3,415,849	957,021	2,458,828	4,586,944	4.79	3.57
F	3,348,267	886,814	2,461,453	4,096,493	4.62	3.78

Table 7.5: *Dynamic performance complexity: temporal upconversion results (prefetching off, allocate on write miss policy, 150 additional delay cycles).*

7.3.4 Write miss policy

To determine the impact of the write miss policy on processor performance, we measured the performance of implementations A through F with prefetching turned on, a fetch on write miss policy and the additional memory delay cycles in the range [0, 150] (Figure 7.8).

Implem.	Cycles	VLIW instr.	Stall cycles	Ops.	Ops. / VLIW instr.	Cycles / VLIW instr.
A	2,781,915	1,913,441	868,474	8,644,096	4.52	1.45
B	2,658,772	1,664,922	993,850	7,445,175	4.47	1.60
C	2,432,044	1,222,603	1,209,441	5,669,897	4.64	1.99
D	2,335,280	1,017,404	1,317,876	4,843,938	4.76	2.30
E	2,302,736	957,285	1,345,451	4,582,816	4.79	2.41
F	2,267,398	886,831	1,380,567	4,178,034	4.71	2.56

Table 7.6: *Dynamic performance complexity: temporal upconversion results (prefetching on, fetch on write miss policy, 150 additional delay cycles).*

The fetch on write miss policy fetches a cache line on a write miss, whereas the allocate on write miss policy allocates a cache line on a write miss. We use memory region based prefetching to limit the amount of write misses (Figure 7.5). Fetching a cache line increases the write miss penalty and memory bandwidth. The increase in memory bandwidth makes the implementations more memory bound, as reflected by the discontinuities in the stall cycles curves, which occur at less additional memory delay cycles (compared to Figure 7.6). Furthermore, the right sides of the stall cycle curves are steeper when the fetch on write miss policy is used.

Especially for higher memory latencies, the write miss policy significantly impacts processor performance. At 150 additional memory delay cycles, implementation F takes 2,267,398 cycles with a fetch on write miss policy (Table 7.6) and

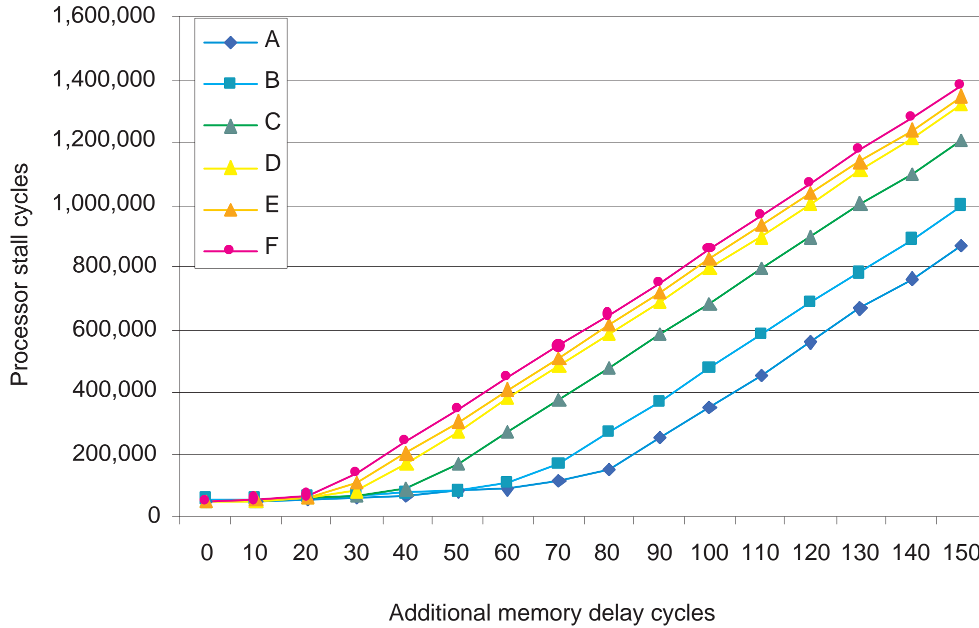

Figure 7.8: *Processor stall cycles as a function of additional memory delay cycles (prefetching on, fetch on write miss policy).*

1,868,948 cycles with an allocate on write miss policy (Table 7.4), a reduction of 17.6%.

7.4 Conclusions

The use of non-aligned memory access and new operations improves the dynamic performance complexity of the temporal upconversion algorithm by 51.9% (Section 7.3.1, implementation A versus implementation F, allocate on write miss policy, prefetching on, 0 additional delay cycles). Implementation F requires 953,825 cycles per image; a temporal upconversion to 60 standard definition images per second would require 57.2 MHz (12.7% of the 450 MHz processor frequency). Prefetching improves the performance of implementation F by 44.2% at 150 additional memory delay cycles (Section 7.3.3). The allocate on write miss policy improves the performance of implementation F by 17.6%, when compared to a fetch on write miss policy (Section 7.3.4, 150 additional memory delay cycles).

Four-way 8-bit operation	*Two-way 16-bit operation*
LD_FRAC8	LD_FRAC16
SUPER_QUADUSCALEMIXUI	SUPER_DUALISCALEMIX
SUPER_QUADUMEDIAN	SUPER_DUALIMEDIAN

Table 7.7: *Four-way 8-bit operations, and their two-way 16-bit counterparts.*

The need for increased image quality has resulted in image formats with 10- and 12-bit luminance value representations. In this chapter, the temporal upconverter algorithm assumed 8-bit image pixel values. To accommodate the higher resolution pixel values, the TM3270 supports all new pixel processing operations with a SIMD subword size of both 8- and 16-bit. Table 7.7 lists the four-way 8-bit operations as presented in this chapter, and their two-way 16-bit counterparts. Note that the four-way 8-bit operations process four subwords, whereas the two-way 16-bit operations only process two subwords.

Chapter 8

Conclusions

We have described the TM3270 media-processor design and evaluated the speedup of new operations and other functional enhancements to the TriMedia architecture. The TM3270 provides a media processing platform on which multiple video and audio processing tasks can implemented, at a power consumption and silicon area efficiency that allows for successful application in both the connected and portable markets. Furthermore, its programmability provides flexibility, which can be exploited to address future media standards. The TN3270 media-processor was recently taped out as part of a SoC for the mobile phone market; initial silicon samples indicate full functionality. The performance contributions of the TM3270 enhancements have been evaluated at an application level, rather than at a kernel level. To this end, a significant effort has been made to optimize three complete video applications, using the enhancements. To ensure realistic cycle behavior, the TM3270 Verilog HDL model is used to guarantee a 100% accurate representation of processor and cache behavior. Furthermore, a performance evaluation environment was created to represent realistic System-on-Chip processor behavior.

In the following three sections we summarize the main conclusions of the individual chapters, briefly describe the main contributions of the thesis and propose directions for further research.

8.1 Summary of conclusions

In Chapter 2 we described the architecture of the TM3270 TriMedia media-processor. In terms of extensions to the TriMedia ISA, we described two-slot operations, collapsed load operations, multiplication operations with rounding and clipping support and CABAC decoding operations. Besides these ISA extensions, we described an instruction cache replacement policy that prevents cache trashing for code with

temporal locality. Furthermore, we described a memory region based prefetching technique, a combined software/hardware technique that prefetches data into the data cache with limited overhead to the programmer.

Chapter 3 described the implementation of the TM3270. We gave an overview of the processor pipeline and more detailed descriptions of the instruction fetch unit and the load/store unit. For the instruction fetch unit, we described how a compressed VLIW encoding and a sequential instruction cache design are implemented to allow for a low power design. The load/store unit implements a novel semi dual-ported cache design, providing high data bandwidth, at a limited area penalty when compared to a single-ported cache. The cache sustains a high store bandwidth by allowing two operations per VLIW instruction and a high load bandwidth by sustaining a single load operation per VLIW instruction with a bandwidth of twice the datapath size. All load and store operations support non-aligned memory access, without incurring any processor stall cycles. Furthermore, a new data prefetching technique is described.

Chapter 4 described the realization of the TM3270 in a low power CMOS process technology, with a 90 nm feature size. The processor provides enough performance to allow for even the most demanding video applications, as illustrated by the dynamic performance complexity of a standard resolution H.264 video decoder (Section 4.3.2). On average the TM3270 provides a speedup of 2.29 over its predecessor, the TM3260, for a series of video applications and kernels through re-compilation and without any modifications to the applications (Section 4.3.1).

Chapter 5 evaluated the TM3270 performance on a motion estimation application. The use of non-aligned memory access and new operations improves the dynamic performance complexity of the motion estimator by 65.1% (Section 5.3.1). The use of data prefetching improves the complexity by a factor of more than two for larger off-chip SDRAM latencies (Section 5.3.3).

Chapter 6 evaluated the TM3270 performance on a MPEG2 video encoder. Collapsed load operations improve the static performance complexity of the block matching kernel by a factor of 1.56 for a 16x16 match (Section 6.2.1). Other new operation improve the complexity of the texture pipeline kernels with a factor of up to 3.06 (Section 6.3.8). Furthermore, we have shown that data prefetching can improve performance by 40.2% for larger off-chip SDRAM latencies (Section 6.5).

Chapter 7 evaluated the TM3270 performance on a motion-compensated temporal upconverter. The use of non-aligned memory access and new operations improves the dynamic performance complexity of the temporal upconversion algorithm by 51.9% (Section 7.3.1). Data prefetching can improve performance by 44.2% for larger off-chip SDRAM latencies (Section 7.3.3). Furthermore, we have shown that a allocate on write miss policy improves performance by 17.6%, when compared to a fetch on write miss policy (Section 7.3.4).

8.2 Main contributions

The main contributions of this thesis are summarized as follows:

- A media-processor design that provides *enough performance to address the requirements of standard and some high definition video processing algorithms* in the connected market, such as high-end TV sets. At the same time, its *low power consumption* enables successful application in portable battery operated markets. The processor's pipeline partitioning and the design of individual units, such as the instruction fetch unit and the load/store unit, are a result of a *trade-off between performance, power and silicon area.*
  - The instruction fetch unit implements a sequential instruction cache design to limit power consumption and supports *a cache line replacement policy that prevents cache trashing as a result of code sequences with limited temporal locality.*
  - The load/store unit design provides high performance through a *semi multi-ported cache*, providing high data bandwidth to the data cache, at a limited area penalty when compared to a single-ported cache. The cache sustains a high store bandwidth by allowing two operations per VLIW instruction and a high load bandwidth by sustaining a single load operation per VLIW instruction with a bandwidth of twice the datapath size. All load and store operations support *non-aligned memory access, without incurring any processor stall cycles.* To our knowledge, the particular implementation of the data cache is unprecedented. Furthermore, *a new data prefetching technique* is introduced. From an architectural perspective the technique provides limited overhead to the programmer and from an implementation perspective it adds limited overhead to the design in terms of silicon area.

- An extension of the TriMedia ISA with a series of new operations:
  - *Collapsed load operations* combine the functionality of a traditional load operation with that of a 2-taps filter function.
  - *Two-slot operations*, as introduced in [68], find their first application in the TM3270 design.
  - *CABAC decoding operations* address the specific requirements of the H.264 standard's Context-based Adaptive Binary Arithmetic Coding (CABAC) decoding process.

- Evaluation of the media-processor design, using a series of video algorithms. The use of complete algorithms as performance benchmarks allows us to quantify the speedup of the individual design innovations at an application level, rather than measuring the speedup of individual media kernels. Furthermore, the evaluation is based on a cycle accurate description of the processor and its SoC environment, which guarantees high accuracy of the presented quantative data.

All of these contributions add to a common goal: a balanced processor design in terms of silicon area and power consumption, which enables audio and standard resolution video processing for both the connected and portable markets. Note that although the innovations have been described in the context of the TM3270 design, their application extends to other media-processors and general-purpose processors.

8.3 Further research

In this thesis we described the TM3270: not a prototype, but a completed design that has taped out as part of a SoC for the mobile phone market. However, advances in media processing applications may eventually require more powerful media processing platforms. At the same time, flexibility, power consumption and silicon area will remain key factors that determine the success of a new platform in the cost-driven embedded consumer market. In this section we suggest some directions of further research for media processing platforms.

For media-processor design we suggest the following:

- Re-address the TM3270 implementation from a power consumption perspective. Although the TM3270 design has acceptable power consumption for the portable market, we feel that there is room for improvement. Currently, a relative large portion of the power consumption is in the SRAMs that implement the processor caches. Techniques such as small level 0 caches, way prediction and the use of low power, but slower SRAMs to implement a deeper pipelined (longer latency) data cache have all been evaluated for general purpose processors. These evaluations typically focus on the average execution behaviour of the technique. However, media-processors have more realtime processing requirements, and as a result, techniques with impressive average performance but with incidental significant performance drops are unacceptable. Interesting techniques should result in *predictable* processor execution behaviour.

- Further enhance the load/store unit design with functionality that addresses media processing data movement. The TM3270 uses memory region based data prefetching to anticipate the need of data before it is actually used by the application. As the hardware takes care of the prefetching, no explicit prefetch operations in software are required. However, once the data is brought into the cache, it will stay there until removed, either explicitly through invalidation or dirty-copy-back operations or implicitly as a result of the LRU replacement policy. When no explicit operations are used to remove the data from the cache, it may stay in the cache even till after it is still required by the application. In this case *the data occupies cache capacity that could have been used for other data structures*. When an application uses data only once, it is to be preferred that the data is invalidated in the cache (and the associated cache line made the least-recently-used cache line), freeing up cache capacity for other data structures. Likewise, when an application produces data and does not access it again, it is to be preferred that the data is copied back to memory and invalidated in the cache. The described behavior could be implemented similar to the region based prefetching. Performance evaluation is required to evaluate the processor performance benefit of better utilizing the available cache capacity.

- With an increasing amount of processor silicon area spent in processor overhead functions such as instruction and data caches, the relative cost of the computational resources deceases. To improve media-processor performance, an increase of computational resources may have become attractive, even when the amount of application parallelism only allows for a limited return on investment. E.g., the datapath could be doubled to 64-bit or the issue bandwidth could be increased. In the case of a 64-bit datapath, source-level backward compatibility is an important design constraint. In the case of increasing the issue bandwidth, clustering is most likely necessary, complicating scheduler technology and potentially degrading performance as a result of inter-cluster communication. New techniques should be evaluated anticipating/using the latest of video codec applications.

The previous suggestions for media-processor design will bring limited advances in terms of power consumption, efficiency and performance level. As mentioned earlier in this thesis (Section 1.1), the relative strengths and weaknesses of different media processing approaches suggests that the best solution may be a combination of approaches. We expect that VLIW-based media-processors provide an attractive component for any combination of approaches, as they provide more raw general-purpose performance than other types of embedded processors and have functional enhancements that address media processing.

To extend a media-processor with dedicated hardware or reconfigurable acceleration, both communication and synchronization need to be addressed. We distinguish a load/store based interface and a functional unit based interface. A load/store based interface allows the control and observation of a relatively independent acceleration block. The existing media-processor bus interface does most likely not provide the latency and throughput requirements required for such an approach. A dedicated acceleration bus interface, with associated accelerator load and store operations could be added to provide the required functionality. The associated load and store operations are scheduled by the media-processor compiler/scheduler. A functional unit based interface allows for the addition of accelerators as if they were media-processor functional units. New media-processor operations are implemented in dedicated hardware or reconfigurable logic, the operations provide high communication bandwdith through the media-processor's register-file and provide synchronization as the operations are scheduled by the media-processor compiler/scheduler.

In [51], M. Sima present the extension of a TriMedia processor with a run-time reconfigurable FPGA based approach. A functional unit based interface is used to combine the approaches. However, the interface is restricted to a single VLIW issue slot of the media-processor, limiting the communication bandwidth between the media-processor and the FPGA based accelerator. As indicated in [26] this restriction may limit the performance potential of the approach. We suggest the evaluation of FPGA based accelerators that utilize the register-file bandwidth of all five issue slots. This increases the communication bandwidth and allows for the creation of more complex operations. Besides, the cost in terms of silicon area of FPGA based approaches is currently too high to enable application in low cost SoCs, and needs to be addressed

Independent of whether the acceleration is performed by dedicated hardware or through reconfigurability, the partitioning of an application in parts performed by the media-processor and parts performed by accelerators needs to be addressed. Tools to identify or perform partitioning, taking into account communication and synchronization between media-processor and accelerators, are required to enable an efficient solution. Furthermore, there is a need for tooling that can perform or assist in the mapping of application parts to the accelerators.

Bibliography

[1] C. Basoglu, W. Lee, and J. O'Donnell. The Equator MAP-CA DSP: an end-to-end broadband signal processor VLIW. *IEEE Transactions on Circuits and Systems for Video Technology*, 12(8):646–659, August 2002.

[2] R. Bhargava, L. John, B.L. Evans, and R. Radhakrishnan. Evaluating MMX technology using DSP and multimedia applications. In *MICRO '98: Proceedings of the 31st International Symposium on Microarchitecture*, pages 37–46, December 2003.

[3] G.A. Blaauw and F.P. Brooks jr. *Computer Architecture: Concepts and Evolution*. Addison Wesley, 1997.

[4] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *ASPLOS '91: Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[5] W. Chen, C. Harrison, and S.C. Fralick. A fast computational algorithm for the discrete cosine transform. *IEEE Transactions on Communications*, COM-25(9):1004–1011, September 1977.

[6] D. Cheresiz, B. Juurlink, S. Vassiliadis, and H.A.G. Wijshoff. The CSI multimedia architecture. *IEEE Transactions on Very Large Scale Integration Systems*, 13(1):1–13, January 2005.

[7] J. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Morris, M. Schuette, and A. Saidi. The reconfigurable streaming vector processor (RSVP). In *MICRO '03: Proceedings of the 36th International Symposium on Microarchitecture*, pages 141–150, December 2003.

[8] J. Corbal, M. Valero, and R. Espasa. Exploiting a new level of DLP in multimedia applications. In *MICRO '99: Proceedings of the 32nd International Symposium on Microarchitecture*, pages 141–150, November 1999.

[9] A. Dasu and S. Panchanathan. A survey of media processing approaches. *IEEE Transactions on Circuits and Systems for Video Technology*, 12(8):633–645, August 2002.

[10] G. de Haan. IC for motion compensated deinterlacing, noise reduction and picture rate conversion. *IEEE Transactions on Consumer Electronics*, 45(3):617–624, August 1999.

[11] G. de Haan, P.W.A.C. Biezen, H. Huijgen, and O.A. Ojo. True motion estimation with 3-D recursive search block-matching. *IEEE Transactions on Circuits and Systems for Video Technology*, 3(5):368–379, October 1993.

[12] K. Diefendorff, P.K. Dubey, R. Hochsprung, and H. Scales. Altivec extensions to PowerPC accelerates media processing. *IEEE MICRO Magazine*, 20(2):85–95, March-April 2000.

[13] R.J. Eickemeyer and S. Vassiliadis. A load instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37(4):547–563, July 2003.

[14] S. Dutta et al. Architecture and design of a Talisman-compatible multimedia processor. *IEEE Transactions on Circuits and Systems for Video Technology*, 9(4):565–579, June 1999.

[15] J.A. Fisher. Very long instruction word architectures and the ELI-512. In *ISCA '83: Proceedings of the 10th International Symposium on Computer Architecture*, pages 140–150, June 1983.

[16] J. Fu, J. Patel, and B. Janssens. Stride directed prefetching in scalar processors. In *MICRO '92: Proceedings of the 25th International Symposium on Microarchitecture*, pages 102–110, December 1992.

[17] B. Girod and K.W. Stuhlmuller. A content-dependent fast DCT for low bit-rate video coding. In *ICIP '98: Proceedings of the International Conference on Image Processing*, pages 80–84, October 1998.

[18] MPEG Software Simulation Group. *mpeg2play.* http://www.mpeg.org/MPEG/MSSG/.

[19] T. Halfhill. Philips powers up for video. *Microprocessor Report*, November 2005.

[20] T. Halfhill. Tensilica previews video engine. *Microprocessor Report*, November 2005.

[21] J. Hoogerbrugge and L. Augusteijn. Instruction scheduling for TriMedia. *Journal of Instruction-Level Parallelism*, 1(1), February 1999.

[22] J. Hoogerbrugge and J.W. van de Waerdt. *EP-1576465 A1 (patent): Counter based stride prediction table data prefetching.* September 2005.

[23] M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro. H.264/AVC baseline profile decoder complexity analysis. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):704–716, July 2003.

[24] P.Y.T. Hsu and E.S. Davidson. Highly concurrent scalar processing. In *ISCA '86: Proceedings of the 13th International Symposium on Computer Architecture*, pages 386–395, June 1986.

[25] R.D. Isaac. The future of CMOS technology. *IBM Journal of Research and Development*, 44(3):369–378, May 2000.

[26] A.K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster. An FPGA-based VLIW processor with custom hardware execution. In *FPGA '05: Proceedings of the 13th International Symposium on Field-Programmable Gate Arrays*, pages 107–117, February 2005.

[27] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA '90: Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, May 1990.

[28] G. Kane and J. Heinrich. *MIPS RISC architecture.* Prentice Hall, 1992.

[29] D. Kim, R. Managuli, and Y. Kim. Data cache and direct memory access in programming mediaprocessors. *IEEE MICRO Magazine*, 21(4):33–42, July-August 2001.

[30] K.H. Kuo, C.C. Ho, K.L. Huang, J. Shiu, and J.L. Wu. A low-cost mediaprocessor based real-time MPEG-4 video decoder. *IEEE Transactions on Consumer Electronics*, 49(4):1488–1497, November 2003.

[31] J. Labrousse and G.A. Slavenburg. A 50 MHz microprocessor with a very long instruction word architecture. In *ISSCC '90: Proceedings of the 37th International Solid State Circuits Conference*, pages 44–45, February 1990.

[32] J. Labrousse and G.A. Slavenburg. CREATE-LIFE: A modular design approach for high performance ASIC's. In *COMPCON '90: Proceedings of the 35th IEEE International Computer Conference*, pages 427–433, February 1990.

[33] J. Lee and A.J. Smith. Branch prediction strategies and branch-target buffer design. *IEEE Computer*, 7(1):6–22, January 1984.

[34] B. Liu and A. Zaccarin. New fast algorithms for the estimation of block motion vectors. *IEEE Transactions on Circuits and Systems for Video Technology*, 3(2):148–157, April 1993.

[35] C. Loeffler, A. Ligtenberg, and G.S. Moschytz. Pratical fast 1-D DCT algorithm with 11 multiplications. In *ICASSP '89: Proceedings of the International Conference on Acoustics Speech, and Signal Processing*, pages 988–991, May 1989.

[36] S.A. Mahlke, W.Y. Chen, W.M.W. Hwu, B.R. Rau, and M.S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *ASPLOS '92: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 238–247, October 1992.

[37] H.S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky. Low-complexity transform and quantization in H.264/AVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):598–603, July 2003.

[38] D. Marpe, T. Wiegand, and H. Schwarz. Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):620–636, July 2003.

[39] J. Montrym and H. Moreton. The GeForce 6800. *IEEE MICRO Magazine*, 25(2):41–51, March-April 2005.

[40] G.E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8):114–117, April 1965.

[41] G.E. Moore. Progress in digital integrated electronics. In *Digest of the 1975 International Electron Devices Meeting*, pages 11–13, 1975.

[42] O.A. Ojo and G. de Haan. Robust motion-compensated video up-conversion. *IEEE Transactions on Consumer Electronics*, 43(4):1045–1056, November 1997.

[43] S. Rathnam and G.A. Slavenburg. An architectural overview of the programmable multimedia processor, TM-1. In *COMPCON '96: Proceedings of the 41st IEEE International Computer Conference*, pages 319–326, February 1996.

[44] I.E.G. Richardson. *H.264 and MPEG-4 video compression.* Wiley, 2004.

[45] J.A. Rivers, G.S. Tyson, E.S. Davidson, and T.M. Austin. On high-bandwidth data cache design for multi-issue processors. In *MICRO '97: Proceedings of the 30th ACM/IEEE International Symposium on Microarchitecture*, pages 46–56, December 1997.

[46] Philips Semiconductors. *PNX1500 data sheet.* http://www.semiconductors.philips.com/acrobat_download/literature/.

[47] N. Seshan. High velociTI processing. *IEEE Signal Processing Magazine*, 15(2):86–101, March 1998.

[48] A. Shahbahrami, B. Juurlink, and S. Vassiliadis. A comparison between processor architectures for multimedia applications. In *ProRISC '04: Proceedings of the 15th Annual Workshop on Circuits, Systems and Signal Processing*, pages 138–152, November 2004.

[49] A. Shoham and J. Bier. TI aims for floating point DSP lead. *Microprocessor Report*, 12(12), September 1998.

[50] F. Sijstermans. The TriMedia processor: the price-performance challenge for media processing. In *ICME '01: Proceedings of the IEEE International Conference on Multimedia and Expo*, pages 222–225, August 2001.

[51] M. Sima, S. Cotofana, S. Vassiliadis, J.T.J. van Eijndhoven, and K. Vissers. MPEG macroblock parsing and pel reconstruction on an FPGA-augmented TriMedia processor. In *ICCD '01: Proceedings of the IEEE International Conference on Computer Design*, pages 425–430, October 2001.

[52] N.T. Slingerland and A.J. Smith. Measuring the performance of multimedia instruction sets. Technical Report UCB/CSD-00-1125, University of California, Berkeley, December 2000.

[53] A.J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[54] D. Sweetman. *See MIPS run.* Morgan Kaufmann, 1999.

[55] Synopsys. *Synopsys PowerCompiler.* http://www.synopsys.com/.

[56] D. Talla, L.K. John, and D. Burger. Bottlenecks in multimedia processing with SIMD style extensions and architectural enhancements. *IEEE Transactions on Computers*, 52(8):1015–1031, August 2003.

[57] J.W. van de Waerdt. *Pending patent application: Memory region based data pre-fetching*. Filing date: May 2002.

[58] J.W. van de Waerdt. *Pending patent application: Cache with administration for pre-fetched cache blocks*. Filing date: 2003.

[59] J.W. van de Waerdt. *Pending patent application: Combined load and computational execution unit*. Filing date: 2003.

[60] J.W. van de Waerdt and C. Basto. *Pending patent application: Cache with high access store bandwidth, at low cost*. Filing date: October 2004.

[61] J.W. van de Waerdt and J. Hoogerbrugge. *EP-1586039 A1 (patent): Using a cache miss pattern to address a stride prediction table*. October 2005.

[62] J.W. van de Waerdt, S. Mirolo, and H. van Antwerpen. *Pending patent application: Stateless Context Adaptive Binary Arithmetic Coding (CABAC) decoding operations*. Filing date: 2005.

[63] J.W. van de Waerdt, G.A. Slavenburg, J.P. van Itegem, and S. Vassiliadis. Motion estimation performance of the TM3270 processor. In *SAC '05: Proceedings of the 2005 ACM Symposium on Applied computing*, pages 850–856, March 2005.

[64] J.W. van de Waerdt and S. Vassiliadis. Instruction set architecture enhancements for video processing. In *ASAP '05: Proceedings of the 16th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 146–153, July 2005.

[65] J.W. van de Waerdt, S. Vassiliadis, and E.W. Bellers. Temporal video up-conversion on a next-generation media-processor. In *SIP '05: Proceedings of the 7th IASTED International Conference on Signal and Image Processing*, pages 434–441, August 2005.

[66] J.W. van de Waerdt, S. Vassiliadis, and et.al. The TM3270 media-processor. In *MICRO '05: Proceedings of the 38th International Symposium on Microarchitecture*, pages 331–342, November 2005.

[67] J.W. van de Waerdt, S. Vassiliadis, J.P. van Itegem, and H. van Antwerpen. The TM3270 media-processor data cache. In *ICCD '05: Proceedings of the IEEE International Conference on Computer Design*, pages 334–341, October 2005.

[68] J.T.J. van Eijndhoven, K.A. Vissers, E.J.D. Pol, P. Struik, R.H.J. Bloks, P. van der Wolf, H.P.E. Vranken, F. Sijstermans, M.J.A. Tromp, and A.D. Pimentel. TriMedia CPU64 architecture. In *ICCD '99: Proceedings of the IEEE International Conference on Computer Design*, pages 586–592, October 1999.

[69] S. Vassiliadis, J. Phillips, and B. Blaner. Interlock collapsing ALU's. *IEEE Transactions on Computers*, 42(7):825–839, July 1993.

[70] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E.M. Panainte. The MOLEN polymorphic processor. *IEEE Transactions on Computers*, 53(11):1363–1375, November 2004.

[71] R. Wester, J.W. van de Waerdt, and G.A. Slavenburg. *EP-1599803 (patent): Reducing cache effects of certain code pieces*. November 2005.

[72] T. Wiegand, G.J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):560–576, July 2003.

[73] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Improved spill code generation for software pipelined loops. In *PLDI '00: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–144, June 2000.

Appendix A

New operations

This appendix defines the new TM3270 operations that are presented in the thesis. Each operation is executed by a specific functional unit; Table A.1 gives an overview of the functional units, with their latency and issue slot location(s).

Functional unit	*Latency*	*Issue slots*				
CONST	1	1	2	3	4	5
ALU	1	1	2	3	4	5
SHIFTER	1	1	2	3	4	5
JUMP	5		2		4	
DSPALU	2	1		3	4	
IMUL	4		2	3		
FALU	4	1			4	
FMUL	4		2	3		
FCMP	2		2	3		
FTOUGH	17		2			
LS_ST	-				4	5
LS_LD	3					5
LS_SPECIAL	-					5
LS_FRAC	6					5
SUPER_ALU	1	1 + 2		3 + 4		
SUPER_DSPALU	2	1 + 2		3 + 4		
SUPER_IMUL	4		2 + 3			
SUPER_CABAC	4		2 + 3			
SUPER_LS_LD	4				4 + 5	

Table A.1: *TM3270 functional units. All functional units, except for the FTOUGH unit, are fully pipelined.*

A.1 Single slot operations

Syntax	
[IF guard] **ALLOC_SET** src1 src2	LS_SPECIAL
Description	
Allocates a data cache line, and the data of the allocated line is set to a 32-bit repeated pattern as contained within operand src1.	
Function (big endian mode)	

```
if guard[0] {
   int i;
   int address = src2 & 0xffff:ff80;  // start of 128 byte line

   for (i = 0; i < 32; i++) {
      Mem[address++] = src1[31:24];
      Mem[address++] = src1[23:16];
      Mem[address++] = src1[15:8];
      Mem[address++] = src1[7:0];
   }
}
```

Syntax	
[IF guard] **DUALASL** src1 src2 → dst1	SHIFTER
Description	
Two-way 16-bit SIMD operation that performs a left shift.	
Function	

```
if guard[0] {
   dst1[31:16] = src1[31:16] << src2;
   dst1[15:0]  = src1[15:0]  << src2;
}
```

Syntax	
[IF guard] **CLSAME** src1 src2 → dst1	DSPALU
Description	
Computes the amount of leading bits that are the same within the sources.	
Function	

```
if guard[0] {
   int temp = src1 ^ src2;
   int clz  = 0;

   while (   (clz < 32)
          && (temp & (1 << (31-clz)) == 0))
      clz++;

   dst1 = clz;
}
```

Syntax	
[IF guard] **DUALISCALEUI_RNINT** src1 src2 → dst1	DSPMUL
Description	
Two-way 16-bit SIMD operation that computes a rounded, scaled and clipped product of 16-bit input values.	
Function	

```
if guard[0] {
   int temp;
   int rounding;

   temp        = (U16) src1[31:16] * (I16) src2[31:16];
   rounding    = ((I16) src2[31:16] < 0) ? 0x1fff : 0x2000;
   temp        = (temp + rounding) >> 14;             // scaling
   dst1[31:16] = IMIN (IMAX (I16_MIN, temp), I16_MAX); // clipping

   temp        = (U16) src1[15:0] * (I16) src2[15:0];
   rounding    = ((I16) src2[15:0] < 0) ? 0x1fff : 0x2000;
   temp        = (temp + rounding) >> 14;
   dst1[15:0]  = IMIN (IMAX (I16_MIN, temp), I16_MAX);
}
```

Syntax	
[IF guard] **LD_FRAC8** src1 src2 → dst1	LS_FRAC
Description	
Retrieves five unsigned 8-bit integers from memory and performs a weighted average on neighboring elements.	
Function (big endian mode)	

```
if guard[0] {
   U8  data0    = Mem[src1];
   U8  data1    = Mem[src1+1];
   U8  data2    = Mem[src1+2];
   U8  data3    = Mem[src1+3];
   U8  data4    = Mem[src1+4];
   U4  weight   = src2[3:0];
   int rounding = 8;

   dst1[31:24] = (data0 * (16-weight) + data1 * weight + rounding) >> 4;
   dst1[23:16] = (data1 * (16-weight) + data2 * weight + rounding) >> 4;
   dst1[15:8]  = (data2 * (16-weight) + data3 * weight + rounding) >> 4;
   dst1[7:0]   = (data3 * (16-weight) + data4 * weight + rounding) >> 4;
}
```

Syntax	
[IF guard] **LD_PACKFRAC8** src1 src2 → dst1	LS_FRAC
Description	
Retrieves eight unsigned 8-bit integers from memory and performs a weighted average on neighboring elements.	
Function (big endian mode)	

```
if guard[0] {
   U8  data0    = Mem[src1];
   U8  data1    = Mem[src1+1];
   U8  data2    = Mem[src1+2];
   U8  data3    = Mem[src1+3];
   U8  data4    = Mem[src1+4];
   U8  data5    = Mem[src1+5];
   U8  data6    = Mem[src1+6];
   U8  data7    = Mem[src1+7];
   U4  weight   = src2[3:0];
   int rounding = 8;

   dst1[31:24] = (data0 * (16-weight) + data1 * weight + rounding) >> 4;
   dst1[23:16] = (data2 * (16-weight) + data3 * weight + rounding) >> 4;
   dst1[15:8]  = (data4 * (16-weight) + data5 * weight + rounding) >> 4;
   dst1[7:0]   = (data6 * (16-weight) + data7 * weight + rounding) >> 4;
}
```

Syntax	
[IF guard] **LD_FRAC16** src1 src2 → dst1	LS_FRAC
Description	
Retrieves three unsigned 16-bit integers from memory and performs a weighted average on neighboring elements.	
Function (big endian mode)	

```
if guard[0] {
   U16 data0    = Mem[src1];
   U16 data1    = Mem[src1+2];
   U16 data2    = Mem[src1+4];
   U4  weight   = src2[3:0];
   int rounding = 8;

   dst1[31:16] = (data0 * (16-weight) + data1 * weight + rounding) >> 4;
   dst1[15:0]  = (data1 * (16-weight) + data2 * weight + rounding) >> 4;
}
```

Syntax	
[IF guard] **LD_PACKFRAC16** src1 src2 → dst1	LS_FRAC
Description	
Retrieves four unsigned 16-bit integers from memory and performs a weighted average on neighboring elements.	
Function (big endian mode)	

```
if guard[0] {
   U16 data0    = Mem[src1];
   U16 data1    = Mem[src1+2];
   U16 data2    = Mem[src1+4];
   U16 data3    = Mem[src1+6];
   U4  weight   = src2[3:0];
   int rounding = 8;

   dst1[31:16] = (data0 * (16-weight) + data1 * weight + rounding) >> 4;
   dst1[15:0]  = (data2 * (16-weight) + data3 * weight + rounding) >> 4;
}
```

Syntax	
[IF guard] **ISCALEFIR8UI** src1 src2 → dst1	DSPMUL
Description	
Four-way 8-bit SIMD operation that computes a rounded, scaled and clipped sum of four products.	
Function	

```
if guard[0] {
   int temp;
   int rounding = 0x20;

   temp =   (U8) src1[31:24] * (I8) src2[31:24]
          + (U8) src1[23:16] * (I8) src2[23:16]
          + (U8) src1[15:8]  * (I8) src2[15:8]
          + (U8) src1[7:0]   * (I8) src2[7:0];

   temp = (temp + rounding) >> 6;
   dst1 = IMIN (IMAX (I8_MIN, temp), I8_MAX);
}
```

Syntax	
[IF guard] **USCALEFIR8UI** src1 src2 → dst1	DSPMUL
Description	
Four-way 8-bit SIMD operation that computes an unsigned rounded, scaled and clipped sum of four products.	
Function	

```
if guard[0] {
   int temp;
   int rounding = 0x20;

   temp =   (U8) src1[31:24] * (I8) src2[31:24]
          + (U8) src1[23:16] * (I8) src2[23:16]
          + (U8) src1[15:8]  * (I8) src2[15:8]
          + (U8) src1[7:0]   * (I8) src2[7:0];

   temp = (temp + rounding) >> 6;
   dst1 = IMIN (IMAX (0, temp), U8_MAX);
}
```

Syntax	
[IF guard] **ISCALEFIR16** src1 src2 → dst1	DSPMUL
Description	
Two-way 16-bit SIMD operation that computes a rounded, scaled and clipped sum of two products.	
Function	

```
if guard[0] {
   int temp;
   int rounding = 0x2000;

   temp =   (I16) src1[31:16] * (I16) src2[31:16]
          + (I16) src1[15:0]  * (I16) src2[15:0];

   temp = (temp + rounding) >> 14;
   dst1 = IMIN (IMAX (I16_MIN, temp), I16_MAX);
}
```

A.2 Two-slot operations

Syntax	
[IF guard] **SUPER_DUALIMEDIAN** src1 src2 src3 → dst1	SUPER_DSPALU
Description	
Two-way 16-bit SIMD operation that computes the three-input median of signed 16-bit input values.	
Function	

```
if guard[0] {
   dst1[31:16] = IMIN (IMAX (IMIN (src1[31:16], src2[31:16]), src3[31:16]),
                       IMAX (src1[31:16], src2[31:16]));
   dst1[15:0]  = IMIN (IMAX (IMIN (src1[15:0],  src2[15:0]),  src3[15:0]),
                       IMAX (src1[15:0],  src2[15:0]));
}
```

Syntax	
[IF guard] **SUPER_QUADUMEDIAN** src1 src2 src3 → dst1	SUPER_DSPALU
Description	
Four-way 8-bit SIMD operation that computes the three-input median of unsigned 8-bit input values.	
Function	

```
if guard[0] {
   dst1[31:24] = UMIN (UMAX (UMIN (src1[31:24], src2[31:24]), src3[31:24]),
                       UMAX (src1[31:24], src2[31:24]));
   dst1[23:16] = UMIN (UMAX (UMIN (src1[23:16], src2[23:16]), src3[23:16]),
                       UMAX (src1[23:16], src2[23:16]));
   dst1[15:8]  = UMIN (UMAX (UMIN (src1[15:8],  src2[15:8]),  src3[15:8]),
                       UMAX (src1[15:8],  src2[15:8]));
   dst1[7:0]   = UMIN (UMAX (UMIN (src1[7:0],   src2[7:0]),   src3[7:0]),
                       UMAX (src1[7:0],   src2[7:0]));
}
```

Syntax	
[IF guard] **SUPER_LD32R** src3 src4 → dst1 dst2	SUPER_LS_LD
Description	
Retrieves two 32-bit integers from consecutive addresses in memory.	
Function	

```
if guard[0] {
   dst1 = Mem[src3+src4];
   dst2 = Mem[src3+src4+4];
}
```

Syntax	
[IF guard] **SUPER_IFIR8UI** src1 src2 src3 src4 → dst1	SUPER_DSPMUL
Description	
Four-way 8-bit SIMD operation that computes the sum of eight products.	
Function	

```
if guard[0] {
   dst1 =   (U8) src1[31:24] * (I8) src2[31:24]
          + (U8) src1[23:16] * (I8) src2[23:16]
          + (U8) src1[15:8]  * (I8) src2[15:8]
          + (U8) src1[7:0]   * (I8) src2[7:0]
          + (U8) src3[31:24] * (I8) src4[31:24]
          + (U8) src3[23:16] * (I8) src4[23:16]
          + (U8) src3[15:8]  * (I8) src4[15:8]
          + (U8) src3[7:0]   * (I8) src4[7:0];
}
```

Syntax	
[IF guard] **SUPER_ISCALEFIR8UI** src1 src2 src3 src4 → dst1	SUPER_DSPMUL
Description	
Four-way 8-bit SIMD operation that computes a rounded, scaled and clipped sum of eight products.	
Function	

```
if guard[0] {
   int temp;
   int rounding = 0x20;

   temp =   (U8) src1[31:24] * (I8) src2[31:24]
          + (U8) src1[23:16] * (I8) src2[23:16]
          + (U8) src1[15:8]  * (I8) src2[15:8]
          + (U8) src1[7:0]   * (I8) src2[7:0]
          + (U8) src3[31:24] * (I8) src4[31:24]
          + (U8) src3[23:16] * (I8) src4[23:16]
          + (U8) src3[15:8]  * (I8) src4[15:8]
          + (U8) src3[7:0]   * (I8) src4[7:0];

   temp = (temp + rounding) >> 6;
   dst1 = IMIN (IMAX (I8_MIN, temp), I8_MAX);
}
```

Syntax
[IF guard] **SUPER_USCALEFIR8UI** src1 src2 src3 src4 → dst1 SUPER_DSPMUL
Description
Four-way 8-bit SIMD operation that computes an unsigned rounded, scaled and clipped sum of eight products.
Function

```
if guard[0] {
   int temp;
   int rounding = 0x20;

   temp =   (U8) src1[31:24] * (I8) src2[31:24]
          + (U8) src1[23:16] * (I8) src2[23:16]
          + (U8) src1[15:8]  * (I8) src2[15:8]
          + (U8) src1[7:0]   * (I8) src2[7:0]
          + (U8) src3[31:24] * (I8) src4[31:24]
          + (U8) src3[23:16] * (I8) src4[23:16]
          + (U8) src3[15:8]  * (I8) src4[15:8]
          + (U8) src3[7:0]   * (I8) src4[7:0];

   temp = (temp + rounding) >> 6;
   dst1 = IMIN (IMAX (0, temp), U8_MAX);
}
```

Syntax
[IF guard] **SUPER_IFIR16** src1 src2 src3 src4 → dst1 SUPER_DSPMUL
Description
Two-way 16-bit SIMD operation that computes the sum of four products.
Function

```
if guard[0] {
   dst1 =   (I16) src1[31:16] * (I16) src2[31:16]
          + (I16) src1[15:0]  * (I16) src2[15:0]
          + (I16) src3[31:16] * (I16) src4[31:16]
          + (I16) src3[15:0]  * (I16) src4[15:0];
}
```

Syntax	
[IF guard] **SUPER_ISCALEFIR16** src1 src2 src3 src4 → dst1	SUPER_DSPMUL
Description	
Two-way 16-bit SIMD operation that computes a rounded, scaled and clipped sum of four products.	
Function	

```
if guard[0] {
   int temp;
   int rounding = 0x2000;

   temp =   (I16) src1[31:16] * (I16) src2[31:16]
          + (I16) src1[15:0]  * (I16) src2[15:0]
          + (I16) src3[31:16] * (I16) src4[31:16]
          + (I16) src3[15:0]  * (I16) src4[15:0];

   temp = (temp + rounding) >> 14;
   dst1 = IMIN (IMAX (I16_MIN, temp), I16_MAX);
}
```

Syntax	
[IF guard] **SUPER_QUADIMIXUI** src1 src2 src3 src4 → dst1 dst2	SUPER_DSPMUL
Description	
Four-way 8-bit SIMD operation that computes four sums of two products.	
Function	

```
if guard[0] {
   int temp;

   temp       = (U8) src1[31:24] * (I8) src2[31:24] + (U8) src3[31:24] * (I8) src4[31:24];
   dst1[31:16] = IMIN (IMAX (I16_MIN, temp), I16_MAX);
   temp       = (U8) src1[15:0]  * (I8) src2[15:0]  + (U8) src3[15:0]  * (I8) src4[15:0];
   dst1[15:0]  = IMIN (IMAX (I16_MIN, temp), I16_MAX);
   temp       = (U8) src1[15:8]  * (I8) src2[15:8]  + (U8) src3[15:8]  * (I8) src4[15:8];
   dst2[31:16] = IMIN (IMAX (I16_MIN, temp), I16_MAX);
   temp       = (U8) src1[7:0]   * (I8) src2[7:0]   + (U8) src3[7:0]   * (I8) src4[7:0];
   dst2[15:0]  = IMIN (IMAX (I16_MIN, temp), I16_MAX);
}
```

Syntax
[IF guard] **SUPER_QUADISCALEMIXUI** src1 src2 src3 src4 → dst1 SUPER_DSPMUL
Description
Four-way 8-bit SIMD operation that computes four signed rounded, scaled and clipped sums of two products.
Function

```
if guard[0] {
   int temp;
   int rounding = 0x20;

   temp        = (U8) src1[31:24] * (I8) src2[31:24] + (U8) src3[31:24] * (I8) src4[31:24];
   temp        = (temp + rounding) >> 6;
   dst1[31:24] = IMIN (IMAX (I8_MIN, temp), I8_MAX);
   temp        = (U8) src1[15:0]  * (I8) src2[15:0]  + (U8) src3[15:0]  * (I8) src4[15:0];
   temp        = (temp + rounding) >> 6;
   dst1[23:16] = IMIN (IMAX (I8_MIN, temp), I8_MAX);
   temp        = (U8) src1[15:8]  * (I8) src2[15:8]  + (U8) src3[15:8]  * (I8) src4[15:8];
   temp        = (temp + rounding) >> 6;
   dst1[15:8]  = IMIN (IMAX (I8_MIN, temp), I8_MAX);
   temp        = (U8) src1[7:0]   * (I8) src2[7:0]   + (U8) src3[7:0]   * (I8) src4[7:0];
   temp        = (temp + rounding) >> 6;
   dst1[7:0]   = IMIN (IMAX (I8_MIN, temp), I8_MAX);
}
```

Syntax
[IF guard] **SUPER_QUADUSCALEMIXUI** src1 src2 src3 src4 → dst1 SUPER_DSPMUL
Description
Four-way 8-bit SIMD operation that computes four unsigned rounded, scaled and clipped sums of two products.
Function

```
if guard[0] {
   int temp;
   int rounding = 0x20;

   temp       = (U8) src1[31:24] * (I8) src2[31:24] + (U8) src3[31:24] * (I8) src4[31:24];
   temp       = (temp + rounding) >> 6;
   dst1[31:24] = IMIN (IMAX (0, temp), U8_MAX);
   temp       = (U8) src1[15:0]  * (I8) src2[15:0]  + (U8) src3[15:0]  * (I8) src4[15:0];
   temp       = (temp + rounding) >> 6;
   dst1[23:16] = IMIN (IMAX (0, temp), U8_MAX);
   temp       = (U8) src1[15:8]  * (I8) src2[15:8]  + (U8) src3[15:8]  * (I8) src4[15:8];
   temp       = (temp + rounding) >> 6;
   dst1[15:8]  = IMIN (IMAX (0, temp), U8_MAX);
   temp       = (U8) src1[7:0]   * (I8) src2[7:0]   + (U8) src3[7:0]   * (I8) src4[7:0];
   temp       = (temp + rounding) >> 6;
   dst1[7:0]   = IMIN (IMAX (0, temp), U8_MAX);
}
```

Syntax
[IF guard] **SUPER_DUALISCALEMIX** src1 src2 src3 src4 → dst1 SUPER_DSPMUL
Description
Two-way 16-bit SIMD operation that computes two signed rounded, scaled and clipped sums of two products.
Function

```
if guard[0] {
   int temp;
   int rounding = 0x2000;

   temp        = (I16) src1[31:16] * (I16) src2[31:16] + (I16) src3[31:16] * (I16) src4[31:16];
   temp        = (temp + rounding) >> 14;
   dst1[31:16] = IMIN (IMAX (I16_MIN, temp), I16_MAX);
   temp        = (I16) src1[15:0]  * (I16) src2[15:0]  + (I16) src3[15:0]  * (I16) src4[15:0];
   temp        = (temp + rounding) >> 14;
   dst1[15:0]  = IMIN (IMAX (I16_MIN, temp), I16_MAX);
}
```

A.3 CABAC operations

This section describes the TM3270 CABAC operations (Section 2.3.5). These operations were added to meet the processor's H.264 decoding requirement: main profile H.264 decoding at main level at a sustained bitrate of at least 2.5 Mbits/s with a maximum dynamic performance complexity of 300 MHz. The CABAC operations are used to efficiently implement the *biari_decode_symbol* function, which decodes a single binary value *bit* from a CABAC coded bitstream [38]:

```
LpsRangeTable[64][4]    // range table for least probable symbol (LPS)
MpsNextStateTable[64]   // MPS state transition table
LpsNextStateTable[64]   // LPS state transition table

biari_decode_symbol (   // decodes a single binary value "bit" from the
                        // CABAC coded bitstream
   inout value,                // coding value, 10-bit value
   inout range,                // coding range, 9-bit value
   inout state,                // modeling context state, 6-bit
   inout mps,                  // modeling context MPS, 1-bit
   in    stream_data,          // bitstream data
   inout stream_bit_position,  // bit position in "stream_data"
   out   bit)                  // decoded binary value
{
   stream_data_aligned = stream_data << stream_bit_position;
   range_lps           = LpsRangeTable[state][(range >> 6) & 3)];
   temp_range          = range - range_lps

   if (value < temp_range) {      // MPS: most probable symbol
      value = value;
      range = temp_range;
      bit   = mps;
      mps   = mps;
      state = MpsNextStateTable[state];
   } else {                       // LPS: least probable symbol
      value = value - temp_range;
      range = range_lps;
      bit   = !mps;
      mps   = mps ^ (state != 0);
      state = LpsNextStateTable[state];
   }

   while (range < 256) { // renormalization, at most 8 bits can be consumed
      value               =   (value << 1)
                            | ((stream_data_aligned >> 31) & 1);
      range               <<= 1;
      stream_data_aligned <<= 1;
      stream_bit_position += 1;
   }
}
```

The two 64 entry state transition tables $MpsNextState$ and $LpsNextState$ are implemented as a hard-coded lookup tables in the SUPER_CABAC functional unit, and are defined by Tables A.2 and A.3. The 64 by 4 entry least probable

symbol range table $LpsRangeTable$ is implemented as a hard-coded lookup table in the SUPER_CABAC functional unit, and is defined by Table A.4.

State	0	1	2	3	4	5	6	7	8	9
NextState	1	2	3	4	5	6	7	8	9	10
State	10	11	12	13	14	15	16	17	18	19
NextState	11	12	13	14	15	16	17	18	19	20
State	20	21	22	23	24	25	26	27	28	29
NextState	21	22	23	24	25	26	27	28	29	30
State	30	31	32	33	34	35	36	37	38	39
NextState	31	32	33	34	35	36	37	38	39	40
State	40	41	42	43	44	45	46	47	48	49
NextState	41	42	43	44	45	46	47	48	49	50
State	50	51	52	53	54	55	56	57	58	59
NextState	51	52	53	54	55	56	57	58	59	60
State	60	61	62	63						
NextState	61	62	62	63						

Table A.2: *Most probable symbol state transition table:* $MpsStateNext$.

State	0	1	2	3	4	5	6	7	8	9
NextState	0	0	1	2	2	4	4	5	6	7
State	10	11	12	13	14	15	16	17	18	19
NextState	8	9	9	11	11	12	13	13	15	15
State	20	21	22	23	24	25	26	27	28	29
NextState	16	16	18	18	19	19	21	21	22	22
State	30	31	32	33	34	35	36	37	38	39
NextState	23	24	24	25	26	26	27	27	28	29
State	40	41	42	43	44	45	46	47	48	49
NextState	29	30	30	30	31	32	32	33	33	33
State	50	51	52	53	54	55	56	57	58	59
NextState	34	34	35	35	35	36	36	36	37	37
State	60	61	62	63						
NextState	37	38	38	63						

Table A.3: *Least probable symbol state transition table:* $LpsStateNext$.

	(range >> 6) & 3					(range >> 6) & 3			
State	0	1	2	3	State	0	1	2	3
0	128	176	208	240	32	27	33	39	45
1	128	167	197	227	33	26	31	37	43
2	128	158	187	216	34	24	30	35	41
3	123	150	178	205	35	23	28	33	39
4	116	142	169	195	36	22	27	32	37
5	111	135	160	185	37	21	26	30	35
6	105	128	152	175	38	20	24	29	33
7	100	122	144	166	39	19	23	27	31
8	95	116	137	158	40	18	22	26	30
9	90	110	130	150	41	17	21	25	28
10	85	104	123	142	42	16	20	23	27
11	81	99	117	135	43	15	19	22	25
12	77	94	111	128	44	14	18	21	24
13	73	89	105	122	45	14	17	20	23
14	69	85	100	116	46	13	16	19	22
15	66	80	95	110	47	12	15	18	21
16	62	76	90	104	48	12	14	17	20
17	59	72	86	99	49	11	14	16	19
18	56	69	81	94	50	11	13	15	18
19	53	65	77	89	51	10	12	15	17
20	51	62	73	85	52	10	12	14	16
21	48	59	69	80	53	9	11	13	15
22	46	56	66	76	54	9	11	12	14
23	43	53	63	72	55	8	10	12	14
24	41	50	59	69	56	8	9	11	13
25	39	48	56	65	57	7	9	11	12
26	37	45	54	62	58	7	9	10	12
27	35	43	51	59	59	7	8	10	11
28	33	41	48	56	60	6	8	9	11
29	32	39	46	53	61	6	7	9	10
30	30	37	43	50	62	6	7	8	9
31	29	35	41	48	63	2	2	2	2

Table A.4: *Least probable symbol range table: LpsRangeTable.*

Syntax
[IF guard] **SUPER_CABAC_CTX** src1 src2 src3 src4 → dst1 dst2
SUPER_CABAC
Description
H.264 context modeling.
Function

```
if guard[0] {
   int value              = (src1 >> 16) & 0x3ff;
   int range              = src1  & 0x1ff;
   int stream_bit_position = src2 & 0x1f;
   int stream_data        = src3;
   int state              = (src4 >> 1) & 0x3f;
   int mps                = src4 & 1;

   int stream_data_aligned = stream_data << stream_bit_position;
   int range_lps          = LpsRangeTable[state][(range >> 6) & 3)];
   int temp_range         = range - range_lps

   if (value < temp_range) {      // MPS: most probable symbol
      value = value;
      range = temp_range;
      mps   = mps;
      state = MpsNextStateTable[state];
   } else {                       // LPS: least probable symbol
      value = value - temp_range;
      range = range_lps;
      mps   = mps ^ (state != 0);
      state = LpsNextStateTable[state];
   }

   while (range < 256) { // renormalization, at most 8 bits can be consumed
      value              =   (value << 1)
                           | ((stream_data_aligned >> 31) & 1);
      range              <<= 1;
      stream_data_aligned <<= 1;
      stream_bit_position += 1;
   }

   dst1 = (value << 16) | range;
   dst2 = (state << 16) | mps;
}
```

Syntax
[IF guard] **SUPER_CABAC_STR** src1 src2 src4 → dst1 dst2
SUPER_CABAC
Description
H.264 bitstream processing.
Function

```
if guard[0] {
   int value               = (src1 >> 16) & 0x3ff;
   int range               = src1  & 0x1ff;
   int stream_bit_position = src2 & 0x1f;
   int state               = (src4 >> 1) & 0x3f;
   int mps                 = src4 & 1;
   int bit;

   int range_lps           = LpsRangeTable[state][(range >> 6) & 3)];
   int temp_range          = range - range_lps

   if (value < temp_range) {      // MPS: most probable symbol
      range = temp_range;
      bit   = mps;
   } else {                       // LPS: least probable symbol
      range = range_lps;
      bit   = !mps;
   }

   while (range < 256) { // renormalization, at most 8 bits can be consumed
      range              <<= 1;
      stream_bit_position += 1;
   }

   dst1 = stream_bit_position & 0x3f;
   dst2 = bit;
}
```

De TM3270 Media-processor

Jan-Willem van de Waerdt

Samenvatting

In dit proefschrift presenteren we de TM3270 VLIW media-processor, de laatste processor in de rij van TriMedia processoren. We beschrijven de innovatieve aspecten van deze processor in relatie tot zijn voorganger: de TM3260. We beschrijven innovaties op het gebied van het processor data cache geheugen, zoals het automatisch *pre-fetchen* van data in het cache geheugen. Daarnaast worden architectuuruitbreidingen beschreven, zoals uitbreidingen aan de Instructie Set Architectuur (ISA) van de processor, Voorbeelden van deze uitbreidingen zijn *collapsed load* operaties, *two-slot* operaties en specifieke H.264 CABAC operaties. De TM3270 innovaties hebben een gemeenschappelijk doel: een gebalanceerd processor ontwerp in termen van silicium oppervlakte en stroomverbruik, dat in staat is om geluid en standaard resolutie video materiaal te bewerken. De prestatie van de processor wordt geevalueerd aan de hand van een serie van video applicaties: een bewegings-schatter. Het decoderen van een MPEG2 video stroom en het opschalen van video materiaal in de tijd. Elk van deze applicaties is geoptimaliseerd door gebruik te maken van de innovaties van de TM3270 processor. Dit stelt ons in staat om de prestatieverbetering van de individuele innovaties te meten, zoals verbeteringen op het gebied van processor data-cache geheugen en nieuwe operaties. We tonen aan dat verbeteringen in het processor data cache geheugen, zoals het automatisch *pre-fetchen* van data in het cache geheugen, de prestatie van de processor kan verdubbelen, wanneer het achtergrond geheugen traag is. De nieuwe operaties reduceren het aantal VLIW instructies dat door de processor wordt uitgevoerd (de zogenaamde statische complexiteit) en verbeteren de prestatie van de processor (de zogenaamde dynamische complexiteit). Gecombineerd resulteren de innovaties tot een verdubbeling van de TM3270 prestatie in relatie tot zijn voorganger, de TM3260, op de geevalueerde video applicaties.

Biography

Jan-Willem van de Waerdt was born in Maarsbergen, the Netherlands, on August 21, 1970. In 1988, he received his Athenaeum diploma, from the *Thorbecke* college in Amersfoort, the Netherlands. In 1994, he received a M.Sc. degree in Computer Science (cum laude), from the *Technical University Twente* in Enschede, the Netherlands. Furthermore, from 1993 till 1994 he studied Psychology at *Utrecht University*, which resulted in a Propedeuse certificate. After his compulsory military service from 1994 till 1995, he started his working career with Philips Research.

From 1995 till 1998, he was employed as a research scientist at Philips Research in Eindhoven, the Netherlands. His work focused on real-time operating systems, multimedia codecs and computer architecture.

Since 1998, he is employed by Philips Semiconductors in San Jose, USA. He worked as logic designer, architect and chief architect in the design of MIPS processors, TriMedia processors and on-chip memory infrastructure components. He is currently the chief processor architect for the TriMedia design activities.

He holds 11 patents and published six papers. In 2003, the TM5250 TriMedia media-processor received the *Best Media Processor of the Year* award from Microprocessor Report, for which he did the architecture and part of the logic design.

The work that forms the basis of this thesis, the TM3270 TriMedia media-processor design, was done at Philips Semiconductors in San Jose, USA.

Patents

US-6643739 B2, EP-1370946 A2, J.W. van de Waerdt and P. Stravers, *Cache way prediction based on instruction base register*, November 2003.

US-6678792 B2, EP-1402373 A2, J.W. van de Waerdt, *Fast and accurate cache way selection method to enable non-speculative data forwarding*, January 2004.

US-6760818 B2, EP-1504341 A2, J.W. van de Waerdt, *Memory region based data pre-fetching*, August 2005.

EP-1530760 A2, J.W. van de Waerdt, *Instruction cache way prediction for jump targets*, May 2005.

EP-1535163 A1, J.W. van de Waerdt, *Processor prefetch to match memory bus protocol characteristics*, June 2005.

EP-1546868 A1, G. Slavenburg and J.W. van de Waerdt, *System and method for a fully synthesizable superpipelined VLIW processor*, June 2005.

EP-1552397 A1, P. Stravers and J.W. van de Waerdt, *Iterative translation lookaside buffer*, July 2005.

EP-1568036 A1, J.W. van de Waerdt, *SDRAM address mapping optimized for two-dimensional accesses*, August 2005.

EP-1576465 A1, J. Hoogerbrugge and J.W. van de Waerdt, *Counter based stride prediction table data prefetching*, September 2005.

EP-1586039, J.W. van de Waerdt and J. Hoogerbrugge, *Using a cache miss pattern to address a stride prediction table*, October 2005.

EP-1599803, R. Wester, G. Slavenburg and J.W. van de Waerdt, *Reducing cache effects of certain code pieces*, November 2005.

Publications

J.W. van de Waerdt, G.A. Slavenburg, J.P. van Itegem and S. Vassiliadis, Motion estimation performance of the TM3270 processor, In *SAC '05: Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 850-856, March 2005.

J.W. van de Waerdt and S. Vassiliadis, Instruction set architecture enhancements for video processing, In *ASAP '05: Proceedings of the 16th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 146-153, July 2005.

J.W. van de Waerdt, S. Vassiliadis and E.W. Bellers, Temporal video up-conversion on a next-generation media-processor, In *SIP '05: Proceedings of the 7th IASTED International Conference on Signal and Image Processing*, pages 434-441, August 2005.

J.W. van de Waerdt, S. Vassiliadis, J.P. van Itegem and H. van Antwerpen, The TM3270 media-processor data cache, In *ICCD '05: Proceedings of the IEEE International Conference on Computer Design*, pages 334-341, October 2005.

J.W. van de Waerdt, S. Vassiliadis et.al., The TM3270 media-processor, In *MICRO '05: Proceedings of the 38th International Symposium on Microarchitecture*, pages 331-342, November 2005.

J.W. van de Waerdt, S. Vassiliadis, E.W. Bellers and J.G.W.M. Janssen, Motion estimation and temporal up-conversion on the TM3270 media-processor, In *ICCE '06: Proceedings of the IEEE International Conference on Consumer Electronics*, pages 315-316, January 2006.

Award

T. Halfhill, Best media-processor: TriMedia TM5250, *Microprocessor Report*, November 2003. *Award for the best media-processor of 2003.*