Hardware / Software co-design for JPEG2000

Examensarbete utfört I Datorteknik
Vid Tekniska Högskolan I Linköping
Av

Per Nilsson

Reg nr: LiTH-ISY-EX--06/3605--SE
Linköping 2006
Hardware / Software co-design for JPEG2000

Examensarbete utfört I Datorteknik
Vid Tekniska Högskolan I Linköping
Av

Per Nilsson

Reg nr: LiTH-ISY-EX--06/3605--SE

Supervisor: Dake Liu
Examiner: Dake Liu
Linköping 8th February 2006
Abstract

For demanding applications, for example image or video processing, there may be computations that aren’t very suitable for digital signal processors. While a DSP processor is appropriate for some tasks, the instruction set could be extended in order to achieve higher performance for the tasks that such a processor normally isn’t actually design for. The platform used in this project is flexible in the sense that new hardware can be designed to speed up certain computations.

This thesis analyzes the computational complex parts of JPEG2000. In order to achieve sufficient performance for JPEG2000, there may be a need for hardware acceleration.

First, a JPEG2000 decoder was implemented for a DSP processor in assembler. When the firmware had been written, the cycle consumption of the parts was measured and estimated. From this analysis, the bottlenecks of the system were identified. Furthermore, new processor instructions are proposed that could be implemented for this system. Finally the performance improvements are estimated.

Keywords: JPEG2000, Discrete Wavelet Transform, arithmetic coding, DSP processors, HW/SW partitioning,
Acknowledgment

I would like to thank my supervisor and examiner Professor Dake Liu for letting me write a thesis on this topic.

I would also like to thank Di Wu and Mikael Andersson for valuable help and discussions regarding this project, and Oskar Flordal for being my opponent on the presentation.
### Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Coding</td>
<td>Coding method to reduce redundancy</td>
</tr>
<tr>
<td>BPP</td>
<td>Bits Per Pixel – an measure on the amount of data used to encode an image compared to its size.</td>
</tr>
<tr>
<td>CIF</td>
<td>Common Intermediate Format – a resolution for videoconferencing</td>
</tr>
<tr>
<td>Code-block</td>
<td>A small part of an image processed by arithmetic coding</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete Cosine Transform</td>
</tr>
<tr>
<td>DWT</td>
<td>Discrete Wavelet Transform</td>
</tr>
<tr>
<td>EBCOT</td>
<td>Embedded Block Coding with Optimized Truncation</td>
</tr>
<tr>
<td>FPS</td>
<td>Frames Per Second – a measurement on the update speed for video (normally 25 on PAL systems and ~30 for NTSC).</td>
</tr>
<tr>
<td>LPS</td>
<td>Least Probable Symbol</td>
</tr>
<tr>
<td>MPS</td>
<td>Most Probable Symbol</td>
</tr>
<tr>
<td>MQ-coding</td>
<td>A specific implementation of a binary arithmetic coder</td>
</tr>
<tr>
<td>QCIF</td>
<td>Quarter CIF – a scaled down version of CIF</td>
</tr>
<tr>
<td>RGB</td>
<td>A color space (Red, Green, Blue)</td>
</tr>
<tr>
<td>Tile</td>
<td>An independent part of a JPEG2000 image</td>
</tr>
<tr>
<td>YCbCr</td>
<td>Luminance, blue chrominance and red chrominance</td>
</tr>
</tbody>
</table>
Contents

ABSTRACT ....................................................................................................................................................... 1
ACKNOWLEDGMENT ....................................................................................................................................... 1
GLOSSARY .......................................................................................................................................................... 1
CONTENTS ......................................................................................................................................................... 1
INTRODUCTION ............................................................................................................................................ 1
1.1 BACKGROUND ........................................................................................................................................... 1
1.1 PURPOSE AND GOAL ............................................................................................................................... 1
1.2 READING INSTRUCTIONS ...................................................................................................................... 2
1.3 WHO SHOULD READ THIS THESIS? ..................................................................................................... 2
2 GENERAL IMAGE CODING THEORY .................................................................................................... 3
2.1 RGB AND YUV COLOR SPACES ............................................................................................................. 3
2.2 ARITHMETIC CODING ......................................................................................................................... 3
2.3 DISCRETE WAVELET TRANSFORMS ....................................................................................................... 4
3 JPEG2000 THEORY .................................................................................................................................. 6
3.1 OVERVIEW ............................................................................................................................................... 6
3.2 DATA ORDERING .................................................................................................................................... 7
3.3 WAVELET TRANSFORM ......................................................................................................................... 7
3.4 BLOCK DECODING .................................................................................................................................. 8
3.5 MQ-DECODING ...................................................................................................................................... 9
3.6 CONTEXT FORMATION ........................................................................................................................... 14
3.6.1 Significance pass ................................................................................................................................. 15
3.6.2 Magnitude Refinement ..................................................................................................................... 15
3.6.3 Clean-up ........................................................................................................................................... 16
3.6.4 Notes on flag updating ..................................................................................................................... 16
3.7 FILE FORMAT AND CODE-STREAM SYNTAX ...................................................................................... 16
4 FIRMWARE DEVELOPMENT ...................................................................................................................... 18
4.1 TARGET PLATFORM ............................................................................................................................... 18
4.2 THE FOCUS OF THE DEVELOPMENT ................................................................................................... 18
4.3 PROGRAM FLOW ................................................................................................................................... 18
4.4 EBCOT MODIFICATION ......................................................................................................................... 19
4.5 LIFTING BASED INVERSED DWT .......................................................................................................... 20
4.6 MEMORY MAPPING ................................................................................................................................ 20
5 PERFORMANCE ESTIMATIONS AND BENCHMARKS ........................................................................ 22
5.1 NOTES ON TARGET PLATFORM ........................................................................................................... 22
5.2 INVERSE DISCRETE WAVELET TRANSFORM ...................................................................................... 22
5.3 MQ-decoding ......................................................................................................................................... 23
5.4 CONTEXT FORMATION .......................................................................................................................... 23
5.5 OTHER OPERATIONS ............................................................................................................................ 23
5.6 OVERALL PERFORMANCE .................................................................................................................... 23
6 INSTRUCTION-SET OPTIMIZATION ........................................................................................................... 25
6.1 Comments about the Current Architecture ........................................................................................ 25
6.2 Accelerators and HW/SW Partitioning ............................................................................................. 25
6.3 Stream Reading ........................................................................................................................................ 26
6.4 Usage of 32-bit Shift Operations ........................................................................................................ 26
6.5 Fast Table Lookups ............................................................................................................................ 27
  6.5.1 Zero Coding Context (ZCC) ........................................................................................................... 27
  6.5.2 Sign Prediction Bit (SPB) .............................................................................................................. 27
  6.5.3 Sign Coding Context (SCC) ........................................................................................................... 28
  6.5.4 Magnitude Context (MAGC) ........................................................................................................... 28
6.6 JumpFZ .................................................................................................................................................. 28
6.7 Summary ................................................................................................................................................ 29

7 FURTHER HARDWARE ACCELERATION ................................................................................................. 30
7.1 A Superscalar DWT Solution ............................................................................................................... 30
7.2 Parallelization of Context Formation ............................................................................................... 30
  7.2.1 Significance propagation pass ...................................................................................................... 31
  7.2.2 Magnitude refinement pass ........................................................................................................... 31
  7.2.3 Cleanup pass .................................................................................................................................... 31
7.3 2-Dimensional Addressing .................................................................................................................. 31
7.4 MQ-Decoder ........................................................................................................................................... 32

8 CONCLUSIONS ......................................................................................................................................... 33

9 BIBLIOGRAPHY ....................................................................................................................................... 34

10 APPENDIX .............................................................................................................................................. 35
10.1 State Register Mapping for Context Formation ............................................................................... 35
10.2 Assembler Instructions Syntax ........................................................................................................ 36
Introduction

1.1 Background

Digital images are an essential part of modern society and the technological progress raises the demand for these applications. Older digital imaging standards, e.g. JPEG or GIF, have proven to be satisfying for yesterday’s applications. But the consumers and the industry has an increasing need for efficient image processing. Aside from just being a more efficient image codec, JPEG2000 also provides a lot of new features for images that its predecessors haven’t.

JPEG2000 seems to be gaining some acceptance and this is partly because of its utility in digital cinemas. Because of the rich set of features, a JPEG2000 stream can be made scalable in many aspects. The decoder system can neglect resolutions levels, signal-to-noise refinement, etc. if it isn’t powerful enough to handle it. Thus, only one stream is required for both ultra high definition hardware and more affordable system.

JPEG2000 offers both lossy and lossless compression. The former method excludes some information from the image and as a result the quality will be decreased to some extent. Because of its flexibility, it is more likely that JPEG2000 will attract professional users and the cinema industry rather than being a standard for less complex units.

The International Standards Organization’s JPEG2000 committee originally came up with the specification, which has become ISO 15444 after officially being supported in 2001. Part 1 of JPEG2000 is royalty and licence-fee free.¹

1.1 Purpose and Goal

The purpose of this master thesis is to construct a decoder and measure the performance, so that an appropriate hardware architecture can be proposed. It is an attempt to see how JPEG2000 performs on a simple single-scalar DSP processor. This later gives rise to several questions: Would additional processor instructions give a significant increase in performance? Are there any other shortcomings of the processor? How would dedicated hardware acceleration for high-performance systems look like? Performance gains will be estimated and/or benchmarked to find out the reasonable alternatives for a JPEG2000 decoder system.

¹ For more legal information, see http://www.jpeg.org/jpeg2000/
1.2 Reading Instructions

Chapter 2 covers general image coding theory, primarily regarding the techniques used by JPEG2000.

Chapter 3 is more oriented towards JPEG2000 as a standard. It describes the algorithms more into detail, how the images structure looks like, etc.

Chapter 4 covers the implementation considerations that were taken during the development of the software decoder.

Chapter 5 presents benchmarks and performance estimations for a software decoder.

Chapter 6 analyses the proposed hardware instructions and their impact on overall system performance.

Chapter 7 briefly treats further work that could be made with more advanced hardware solutions.

1.3 Who Should Read This Thesis?

While it is not it’s main purpose, the reader could take this as an introduction to the imaging standard JPEG2000. But the main focus remains on how hardware architectures for such a system could be designed. The thesis describes how the hardware acceleration improvements are identified and evaluated. The intended reader is a person who has an equivalent of about 4 years of studies at a technical master program or more. The reader should have some basic knowledge about imaging and DSP processors.
2 General image coding theory

2.1 RGB and YUV color spaces

RGB is the most common way to represent computer graphics. The system consists of three channels (that can be thought of as grayscales) and these are mapped to the lighting colors of red, green and blue. This technique is used by graphics cards, monitors, etc, when handling images. Each of these channels are represented by one byte per pixel.

But these colors are not equally important to the human eye. Image coding techniques uses a transform to sort out the information that the human eye is most sensitive towards and place it in channel of its own [3]. The two other channels may have some information subtracted from it through down-sampling. The more important channel is called luminance and the two others are called chrominance. There are two matrixes for transforming between the RGB and YUV color spaces. If the two chrominance channels are reduced with 50% in both horizontal and vertical direction, then there image has reduced its need of storage with 50% in total.

2.2 Arithmetic Coding

Arithmetic coding is processed by dividing a range between 0 and 1. This is done iteratively and depending on the probability distribution of different symbols. I will illustrate this principle with the following example:

Consider a set of four symbols \{A, B, C, D\} that has the probabilities \{1/2, 1/6, 1/6, 1/6\} for a given amount of data. When a symbol is encoded, the current range will be divided according to the magnitude of the probability. The new range will then be used iteratively for the next symbol is encoded. The encoding of the message “ABAD” will take place in this manner:

Divide the original range \([0,1]\) to \([0,0.5]\). If the next symbol also is an A we would yet again have chosen the lower 50% of \([0,0.5]\), but since it is a B we will choose the next 1/6 of it, i.e. \([0.25, 0.333]\). In the next iteration we will choose the lower 50%, which gives us \([0.25, 0.29166]\). The final symbol D will be encoded by choosing the last 1/6 of this interval, which is \([284722, 0.29166]\).

We will now select a number in that interval that requires the least possible amount of bits. That number is 0.2890625 which requires 7 bits to be stored (compared to the 8 bits that an un-encoded message would require).
Under very fortunate conditions, arithmetic coding could actually give us a compression that results in *less than* 1 bit per symbol. Traditional Huffman coding (that is used in standard JPEG) would require us to form larger symbols out of the regular ones in order to achieve such powerful compression.

### 2.3 Discrete Wavelet Transforms

The Discrete Wavelet Transform DWT is what we call a dyadic tree-structured subband transform with a multi-resolution structure. Compression schemes that are based upon subband transforms that is arranged in tree-structures are called wavelet-based schemes [4]. If a 2-dimensional transform is applied once to an image the result will be four bands (that are the high and low pass bands in vertical and horizontal direction respectively). The low/low band can be seen as a miniature of the original image. The transform is applied iteratively on this new subband, while the other three bands are left unchanged. This is repeated for a few level until there is an actual “tree structure”, as the text above referred to.
The utility of this transform for image compression is evident. The high-pass bands ends up having far less information and are thus relatively easy to compress with satisfying results. Furthermore, it will be possible to extract a thumbnail of an image without having to decompress the whole image into detail (for example, LL3 in this example will look like that). A higher number of levels will improve the compression ratio, but it has an upper limit that will be approached around 5-9 levels.

<table>
<thead>
<tr>
<th>LL3</th>
<th>HL3</th>
<th>HL2</th>
<th>HL1</th>
</tr>
</thead>
<tbody>
<tr>
<td>LH3</td>
<td>HH3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LH2</td>
<td>HH2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LH1</td>
<td></td>
<td></td>
<td>HH1</td>
</tr>
</tbody>
</table>

The original JPEG standard uses the DCT transform instead, which gives the image some annoying blocking artifacts. Note the orthogonal edges at the picture to the left. This is because it transforms 8x8 blocks individually, while the wavelet transform can be applied to arbitrary large areas. The picture above illustrates this improvement (JPEG2000 to the right). The images are enlarged cutouts from an image with a compression ratio of 1:94. I should point out that the DCT is less computational complex, which is one of the reasons why video codecs like various MPEG standards use it.
3 JPEG2000 theory

This chapter explains the principles behind the JPEG2000 codec. First there is an overview of how the different processes stick together and later they are further treated in detail.

3.1 Overview

Basically, everything that is done in the encoder has a corresponding operation in the decoder, although the steps are done in reversed order of course. The first step in the encoding (and equally the last step of the decoder) is the color space conversion. While it is possible to exclude this operation (e.g. for gray-scale images) it is highly recommended for RGB images. In most cases, the image has had its color space transformed in order to obtain one channel that the eye is more sensitive towards (luminance) and two channels that the eye is less sensitive to (chrominance). The pixels on a computer are normally represented in the RGB space, while lossy image codecs might convert this to YCrCb and downsample the chrominance channels (still without losing much of the quality). During the decoding phase, this leads to a linear matrix operation. In various extensions of JPEG2000, the encoder has the option to define a custom-made color transformation.

The encoder then performs a 2-dimensional wavelet transform on each color channel. This operation gives four frequency bands, and then the encoder performs the transform once again on one of those bands. This is repeated a certain number of times. The decoder’s task is to perform an inversed transform correspondingly.

After truncation of the coefficients, the encoder performs so-called “block coding”. This is the most computational complex part of the system. In the decoder part, the coefficients are scanned in a certain order so that different coding passes can be applied. The coding passes update a fairly large table of states for the coefficients. These passes consume encoded data that is decoded using a subsystem with MQ-coding. Depending on this information and its internal state, the coefficient bits are built up and the states are further updated. In that way, it is a context-adaptive binary arithmetic coding.
3.2 Data ordering

The partitioning of data in JPEG2000 is non-trivial and clearly deserves an explanation. The division on the highest level occurs when an image is divided into tiles. Normally, a tile is 256x256 or 128x128 pixels. These tiles are divided into channels (though there may be tiles that only possess one channel). A standard image consists of three channels per tile. The wavelet transform operates on these channels and the result is several bands from each channel. The bands are further divided into rectangular precincts, which are divided into code-blocks. The arithmetic coding operates on such code-blocks. The encoder selects bitplanes from the code-blocks and forms them into indivisible packets that then occur in the bitstream. The reason why this “packet” unit is introduced is because it makes it possible to enable scalability (low-quality decoding doesn’t always need the least significant bits). A decoder on a slow system might opt to just skip certain packets.

3.3 Wavelet transform

When it comes to the irreversible discrete wavelet transform, part 2 of JPEG2000 supports a large class of wavelet kernel, but the basic version only support the CDF 9/7. The low- and high-pass analysis filters have lengths 9 and 7 respectively (for both the DWT and the inverse DWT). Note that the filter is separable, which means that the two-dimensional filter can be applied by performing two one-dimensional filters in each direction.

\[ h_0(z) = 0.6029 + 0.2669 (z^1 + z^{-1}) - 0.0782 (z^2 + z^{-2}) - 0.0169 (z^3 + z^{-3}) + 0.0267 (z^4 + z^{-4}) \]

\[ h_1(z) = 0.5575 - 0.2956 (z^1 + z^{-1}) - 0.288 (z^2 + z^{-2}) + 0.0456 (z^3 + z^{-3}) \]
Note that the edges of a tile require this non-causal filter to extend the y signal in the corners by mirroring it. This is done on both edge according to the illustrated example.

There is also a reversible DWT that is used exclusively for lossless encoding. This transform doesn’t require any multiplication with real numbers, since shift operations replaces the need for this. The basic part of JPEG2000 specifies only one transform, which is derived from the spline 5/3 transform. It uses right shift operations instead of multiplications with real numbers, which gives no truncation or other effects that can modify the final look of the image.

### 3.4 Block decoding

The EBCOT (Embedded Block Coding with Optimized Truncation) is the “heart” of JPEG2000. On the decoder side, it operates directly on the code-blocks in the bit-stream and the output is actual coefficients in that block. The main loop in this process applies three different decoding passes on these coefficients; Significance propagation, Magnitude refinement and Cleanup. The decoding passes will use the MQ coder as a subsystem for making decisions – or rather the MQ decoder extracts bit by using the context provided from the passes. In addition, the MQ coder must access the context states that are updated during the processing.
The main loop itself iterates through the coefficients that it builds up in a specific manner: First of all, the most significant bits are built up first. All coefficients are zero until the significance pass sets the first value (depending on which bitplane it activates on). Coefficients that have been made active gets additions from the magnitude refinement pass (the size of the added number also depending on the bitplane). The last pass performed is finally the clean-up. The code-blocks are normally 32x32 or in some cases 64x64 coefficients. They are scanned 4 rows at a time, and these 4 rows have scanned from left to right column-wise.

### 3.5 MQ-decoding

As previously mentioned arithmetic coding has one major drawback if one considers computation speed; multiplications. The MQ coder has managed to avoid this problem and thus making the encoding and decoding processes far less time consuming. Instead, the range is set as an integer (that is mapped to a presumed floating point value). The range is kept close to 1.0 (between 0.75 to 1.5) so that a calculation can be approximated. A shifting operation is used for every time the value ends up below 0.75, which is called renorming.

The Decoder initializes the decoder through the process *initdec*. Contexts (CX) and bytes of compressed data (CD) are read and the output is a binary return value (D). The probability estimation procedures are located in a process called *decode*. The JPEG2000 final draft [2] proposed the following routine:
The decoder has 3 16-bit registers for state information; Chigh, Clow and A. The former two registers are sometimes thought of as one single 32-bit register during the renormalization process. The decoding comparisons use Chigh alone though. New data is read to the 8 most significant bits of Clow.
Depending on the C and A values, there may be a need for renormalization. If the MPS sub-interval size A is not logically less than the LPS probability estimate Qe(I(CX)), an MPS did occur and the decision can be set from MPS(CX). The index I(CX) is updated from the next MPS index (NMPS), which is stored in a table. If the LPS sub-interval is larger, the conditional exchange and LPS have occurred. The probability update switches the MPS sense if the specific state has the switch bit associated with it and updates the next LPS index (also stored in the table).

The aforementioned table of states consists of 46 rows. Each row has an integer value Qe, two pointers to other state rows (next MPS and LPS) and a switch flag.
The MPS and LPS exchanges are similar.
The renormd step may read a new byte if the buffer is empty. Then it left shifts both A and C and decreases the buffer counter. If A is less than 0x8000, we may have to repeat that step. The bytein procedure handles bit-stuffing techniques in the code stream. It also adds the read byte to C. The variable CT in the flow chart is a pointer to the current bit position in the byte that is buffered.
3.6 Context formation

The coding passes needs to store context information about the coefficient, which is as much as 15 bits / or flags per coefficient. The flags are updated depending on the status of all eight neighbors! Needless to say, the “worst case” may require a coding pass to check all neighbors, although this occurs fairly seldom. These 15 flags then determine which context state that is sent to the MQ coders. This could be calculated by a series of arithmetic operations, but most mature implementations use a lookup table instead (approximately 5.5 KWords). Software implementations may build up the tables during startup and then refrain from recalculate the flag/context mapping.

A bit-plane’s stripe (a unit with of 4 pixels height) is first scanned with the significance pass. Then the next stripe is done. When all stripes have been done, continue with the magnitude refinement pass and the clean-up pass. After this has
been completed, move on to the next bitplane and repeat the process until all bitplanes have been done. The scanning order within a stripe is made clear in the illustration.

![Diagram of scanning order within a stripe](image)

The width of a stripe is different depending on the size of the block, but there is usually an upper limit at 64 pixels. Every codec pass starts by checking certain flags; if it shouldn’t be invoked then it will exit immediately. Here is a summary for how the steps are performed for the different passes. *Flag* denotes the flag register file for a coefficient and the constants \(k1\), \(k2\) and \(k3\) are certain flag combinations that are pre-determined.

### 3.6.1 Significance pass

- If this condition is not fulfilled, then exit: (flags AND \(k1\)) and (flags AND \(k2\)).
- Look up the context for this flags and use it as an in parameter for the MQ-decoder that is called.
  - If the return value is true, lookup a new context number depending on the flags and call the MQ-decoder.
  - Look up another table value for the flags and perform an AND-operation on the flags with this.
  - Update flags for this sample by using this data.
  - Set the *significant* and *visit* flags.
  - Depending on the last lookup value, set the coefficient and exit
- If the original value from the MQ-decoder was false, set the visit flag.

### 3.6.2 Magnitude Refinement

- If this condition is not fulfilled, then exit: (flags AND \(k2\)) and not (flags AND \(k3\)).
- Look up the context for this flags and use it as an in-parameter for the MQ-decoder that is called.
  - If the return value is true, lookup a new context number depending on the flags and call the MQ-decoder. Otherwise exit.
  - Depending on this value and if (coefficient < 0), add or subtract a pre-calculated constant from the coefficient.
  - Set the *refine* flag.
3.6.3 Clean-up

- If this condition is not fulfilled, then unset the visit flag and exit: (flags AND k2) and (flags AND k3).
- Look up the context for this flag and use it as an in-parameter for MQ-coder that is called.
  - If the return value is true, lookup a new context number depending on the flags and call the MQ-decoder.
  - Look up another table value for the flags and perform an AND-operation on the value from the MQ-coder with this.
  - Depending on the last lookup value, set the coefficient.
  - Update flags for this sample by using this data from the lookup.
  - Set the significance flag.

3.6.4 Notes on flag updating

The table-lookups are different depending on the coding pass, but the flag updating (i.e. the non-predetermined ones) is the same algorithm. There are however two different ways of doing this depending on if a property of the code stream called causal mode is switched on or off. Flags are always set and never unset, which basically means logical OR operations. If causal mode is on, this means that on the flags on the current and successive rows are updated, while non-causal mode also requires that the flags on the preceding row are set. This gives 5 and 8 flag updates for the neighboring coefficients respectively.

Why is the flag-updating necessary? Since the probability distribution of a coefficient’s amplitude is correlated with the status of its neighbors. A coefficient that has many neighbors that are significant at a given bitplane is more likely to become significant during the next bitplane. There are often big areas in a high-frequency band that are truncated to zero or at least have very low amplitude (this is related to the absence of sharp edges in the image). Apart from this, the context formation is of course dependent on the band orientation since high-frequency bands usually have a less random distribution of data.

3.7 File format and code-stream syntax

JPEG2000 offers a rich file format and many ways to include information about the image’s structure.

The file format is structured as boxes that in turn have sub boxes. For example, the codestream which contains the compressed data for a tile is stored in one box, information about color transformation is stored in another box, information about intellectual property rights and so on. There are even boxes dedicated to include XML information in a image, which then would be very application-specific for custom solutions as the standard doesn’t declare how they should be used. A box typically begins with a marker that declares the type of
information, followed by an integer with the length of the information length (always present) and finally the data/variables in a pre-determined order. This also means that different parts of an image can be encoded in different styles. A sophisticated encoder might try to use different styles if an image’s characteristics differ much between a couple of areas.
4 Firmware Development

This chapter discusses some implementation considerations and the development environment.

4.1 Target Platform

The DSP processor is a 16-bit fixed point DSP with one MAC unit with 36-bits accumulators. It features parallel accesses to data memories (there are two of them, with two address pointers for each). In addition to its present instructions, the development tools allow custom-built hardware instructions to be simulated easily. The purpose is to allow application-specific features, rather than having a very rich instruction set for a processor. This can provide higher performance, smaller silicon area and less power consumption. Accelerators may be designed in a way that they can benefit the most computational intensive tasks of a targeted application.

4.2 The focus of the development

One can divide the block coding into two parts; tier-1 and tier-2. Tier-1 handles all calculations from the codeblock-level and down to the MQ-decoder. Tier-2 on the other hand manages packets, tagtree-buildups, etc. Because tier-1 is the computational complex part, the focus on the development has been on the tier-1 algorithms. Those are the ones necessary to give good performance estimations, which is the goal of this thesis. While some work on tier-2 has been done, many of the features were dropped or simplified due to time constraints and the scope of the thesis.

4.3 Program flow

Here follows the main loop of the software implementation.
The parsing step is responsible for extracting the necessary information about the file and the bit-stream. It will ultimately localize the encoded information for the code-blocks.

Once a code-block has been identified, the EBCOT processing is started. The EBCOT interacts with the coding passes and the MQ-decoder as previously described. The dequantization step is more or less merged with this step, but shown individually in this diagram to emphasize the nature of the processing order.

The inverse DWT and color transformation are the last two steps. They work on the same

### 4.4 EBCOT modification

The MQ decoder that was modeled previously was modified slightly in order to improve performance. The probability index table is almost similar to the standard, but the number of rows has been doubled. There is now one state for mps=1 and one state for mps=0. Less arithmetic operations are needed in that way. This “trick” is done by other JPEG2000 libraries on the market, e.g. JasPer. The pointers in the table are of course revised accordingly. The minor drawback is that the table will give up more memory, but 376 words out of the 64K in a tap-memory is well worth the computational time that it can save.

The case is similar with the lookup-tables for the context formation, but 5.5K clearly has to be motivated by significant speedups. A “direct” calculation would require tens of extra clock cycles every time.
4.5 Lifting based inversed DWT

The straightforward DWT filter mentioned in the theory chapter can be speeded up significantly by using a lifting based scheme. In many cases, it may be preferable to employ this method according to the experts [1]. It requires fewer arithmetic computations than direct implementations of the equations.

\[
X(2n) = K x Y_{\text{ext}}(2n)
\]

\[
X(2n+1) = \frac{1}{K} x Y_{\text{ext}}(2n+1)
\]

\[
X(2n) = X(2n) - (\lambda_1 x [X(2n-1) + X(2n+1)])
\]

\[
X(2n+1) = X(2n+1) - (\lambda_2 x (X(2n) + X(2n+2)])
\]

\[
X(2n) = X(2n) - (\lambda L-1 x [X(2n-1) + X(2n+1)])
\]

\[
X(2n+1) = X(2n + 1) - (\lambda_L x (X(2n) + X(2n + 2)])
\]

\[
\alpha = -1.586, \beta = -0.052, \gamma = 0.882, \delta = 0.443, K = 1.230
\]

4.6 Memory mapping

For the wavelet transform, the firmware occupies two memory spaces in two different tap memories. There are two reasons for this. The memories are small so that we want to allow bigger tile sizing in that way. Secondly, there might be some benefits as parallel memory transfers can be made in that way. The horizontal transform is represented by a move from memory A to B, while the vertical transform represents a move back from B to A.

For the context formation process, the coefficients and the flags files are located in different memories. There are rather big tables located in the same memory area as the coefficients. There are most accesses to the flags and tables, which makes it a good choice to place them in the separate memories. The lookup
Tables are static, so they will always be located in that memory area. The coefficients of each code-block will be moved after all have been calculated before the inverse DWT is applied. The address calculations will be simpler during the block-encoding phase in that way. Since they are done once per bitplane, it makes more sense to address them as an independent block. Plus, it is easier to implement and it corresponds well with the reference code. The rearrangement is needed because the wavelet transform loop then may be able to read the samples sequentially. During the inversed wavelet transform, the input data comes from two distinguished memory areas within the same memory, but is merged together when written to a lower-level subband in the output memory.

<table>
<thead>
<tr>
<th>DM0</th>
<th>TM0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usage</strong></td>
<td><strong>Usage</strong></td>
</tr>
<tr>
<td>1. Various image and code stream properties and constants, temporary variables</td>
<td>&lt;1K</td>
</tr>
<tr>
<td>2. Look-up tables</td>
<td>&lt;6K</td>
</tr>
<tr>
<td>3a. Block decoding coefficients</td>
<td>4K</td>
</tr>
<tr>
<td>3b. Wavelet coefficients</td>
<td>57K</td>
</tr>
<tr>
<td>3c. Downsampled color components</td>
<td>57K</td>
</tr>
<tr>
<td></td>
<td>4a. Block decoding flag matrix</td>
</tr>
<tr>
<td></td>
<td>4K</td>
</tr>
<tr>
<td></td>
<td>4b. Temporary Wavelet result</td>
</tr>
<tr>
<td></td>
<td>57K</td>
</tr>
<tr>
<td></td>
<td>4c. RGB samples</td>
</tr>
<tr>
<td></td>
<td>64K</td>
</tr>
</tbody>
</table>

Areas 3 and 4 are shared between different operations. Area 2 is read-only during the actual image processing. Since TM0 hasn’t got any static content, it can be utilized fully for calculating the final image. The final image can also be written to an off-chip memory if preferred. For bigger tile sizing (e.g. 256x256) an off-chip memory must be used temporarily during the transforms, but not for smaller tile sizing (like the common 128x128 partition, or a 176x128 QCIF image).

Communication between different functions mostly uses registers as in- and out-parameters (sometimes with addresses to a specific memory area). In many cases, parameters are stored as global variables (in area 1).
5 Performance Estimations and Benchmarks

This chapter intends to predict performance for a pure software implementation by using benchmarks and estimations.

5.1 Notes on target platform

The processor architecture has a single issue datapath, one MAC and one ALU. If is a basic DSP processor that probably is a good example of a “standard” single-scalar processor. I will not discuss how other platforms would perform, but the results are probably on pair with many other processors out on the market. Also, I should point out that this thesis will not provide a deep discussion on lossless compression (since it isn’t as common as the low-bitrate lossy solutions). Furthermore, I will compare the performance by discussing how it would handle QCIF video (which could be an interesting application) and predict MIPS costs in that situation.

5.2 Inverse Discrete Wavelet Transform

Since the DSP processor has fast MAC operations, the results are somewhat satisfying for a pure software solution. Since the irreversible transform is probably more interesting for real-time applications, this thesis will primarily discuss the results that are related to it. One level requires 5 MAC operations in average for both the horizontal and vertical direction. On average, one sample will be included in less than \[\sum(1/4^i) = 1.33\] levels. This leads to about 13 MAC operations per pixel for a given tile. The current overhead for memory accesses that cannot be parallelized, pipeline-stalling, etc. in the main transform loop is roughly 1.5 clock cycles per transform (and thus almost insignificant). For NTSC QCIF (176x120 pixels) and a typical YCbCr color space (1.5 color component per pixel because of down-sampling), this would mean less than 12 million MAC operations per second if the solution aims for a frame rate at 30 fps. On the other hand, if VGA resolution (640x480) is desired, there will be a need for almost 180 million MAC operations per second. The MIPS budget on the current DSP processor does not allow that. A MAC operation will actually consume 2 clock cycles if done alone,
but the second execution cycle is done in parallel with the next instruction and the result is available thereafter.

5.3 MQ-decoding

The MQ-decoding requires approximately 40 clock cycles per coded bit. This may be depending on how efficient the coding is though. Efficient coding requires significantly less new bytes read and renormalizations, while more random data may have a longer path to the output. The ideal situation would involve a table lookup, 2 subtractions, AND-operation and 2 conditional jumps. Assuming that the image deals with approximately 0.2 bits per pixel, this will mean about 8 clock cycles per pixel. QCIF would then require about 5 MIPS and VGA would require about 70 MIPS (which would consume a great deal of the execution time for lower power general purpose processors).

5.4 Context formation

Since the flag updating algorithm is about equally complicated for each case, its clock cycle consumption is easily predicted. The software implementation consumes 39 clock cycles for causal mode and 50-55 cycles for non-causal mode. Because of this operation and the fact that it two MQ-calls might be needed, most time consuming coding passes are those that handle the significance propagation and cleanup. They are about equal to the MQ-decoding when it comes to operations per pixel, while the magnitude refinement pass equals about half of it. About 100 clock cycles per coded bit should be expected in total.

5.5 Other Operations

The file parsing, color transformation, etc. is not of big interest while benchmarking JPEG2000 solutions. The color transformation is simply a matrix multiplication with (most often) a 3x3 matrix and almost any DSP processor will perform this without too much trouble. Even though JPEG2000 has a rather advanced system for arranging data and setting up files with boxes, it is not a major issue (at least performance-wise) It should be noted that implementing this management in assembler language is time consuming and the development would probably have been better off by using C in retrospect. This decision was probably unwise to some extent.

5.6 Overall performance

A pure software solution will clearly not be able to handle higher resolutions. It will primarily be able to achieve performance that is suitable for smaller
handheld devices, such as PDAs or cell phones. Here is a performance estimation on a 0.2 bpp video at 30 fps (for NTSC QCIF).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of clock cycles</th>
<th>Share of total MIPS budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>MQ-decoder</td>
<td>5M</td>
<td>14%</td>
</tr>
<tr>
<td>Significance pass</td>
<td>5M</td>
<td>14%</td>
</tr>
<tr>
<td>Magnitude refinement pass</td>
<td>3M</td>
<td>9%</td>
</tr>
<tr>
<td>Cleanup pass</td>
<td>5M</td>
<td>14%</td>
</tr>
<tr>
<td>Inverse wavelet transform</td>
<td>17M</td>
<td>48%</td>
</tr>
<tr>
<td>TOTAL</td>
<td>35M</td>
<td></td>
</tr>
</tbody>
</table>

Adding calculations for loop preparations, parsing operations, color transformation, scanning loop control, etc. the overall requirement for QCIF video will be at around 45 MIPS. It should be noted that the approximation here probably has an error margin at about 20%. It is still highly relevant, since it illustrates that low-resolution video is possible without hardware acceleration and offers an approximate distribution of computational complexity.

A resolution like CIF (352x240) would not be very far away if we had access to a processor that could be clocked around 150-200 MHz. However, it is definitely a case where just a modest improvement in performance could be enough to satisfy the requirements for this resolution. Resolutions around VGA and higher is not possible for simple low-power DSP processors without adding any dedicated hardware.

These numbers apply to low-bitrate images. This is why the Inverse Wavelet transform has a seemingly high proportion of the total MIPS cost. The number of computations for a high-bitrate image will be the same, since it depends on the number of pixels (and to some extent the number of resolution-levels that are used). When decoding higher bitrates the context formation and bitplanes will have to perform much more decoding. It will not grow linearly, but not very far from that in the end. When processing “near-perfect” lossy images, the whole EBCOT system will come out has the most computational intense part of the decoding.
6 Instruction-set Optimization

For DSP processors that allow custom-built accelerators to enhance performance, it is possible to design instructions that improve performance for the target application. This chapter proposes a couple of new instructions that will speed up JPEG2000 decoding compared to a pure software implementation without any application-specific instructions.

6.1 Comments about the Current Architecture

The DSP processor has only 16 general purpose registers, which gives a JPEG2000 decoder a severe performance penalty. For instance, the MQ decoder is called very frequently within the coding passes. A bigger processor could store the state information in the registers (benefits up to 8 registers in the software implementation including the data sent). This would only leave another 8 registers for the context formation, which is too little to really make it efficient. A processor with 32 registers would not run out of registers in a serious way.

Also, the instruction set doesn’t support AND and OR operations with immediate operands, which would be useful on many occasions (this doesn’t imply that the presence of such instructions necessarily would improve the architecture, since it is a trade-off of many things).

6.2 Accelerators and HW/SW Partitioning

The DSP processor architecture is prepared for adding new hardware instructions that can be implemented. From the programmer point of view the instruction set is extended with these new additions. The idea is that it should be possible to view the accelerator as an intrinsic part of the processor on the assembler level. The accelerated instructions have access may use the same registers as operands for reading and writing. It also means that the same address bus and memories is available for it. An instruction could operate within a fixed number of clock cycles, e.g. only one. Instructions that run in only one clock cycle are easier to implement, since it is a pure slave machine.

The new instructions that are proposed are for the arithmetic coding and context formation tasks of JPEG2000. The inverse DWT transform will not be covered, since the MAC operations that it is dependent on isn’t really replaceable with anything that could be done by simple modifications of the hardware. DSP
systems are composed of blocks representing well-known “standard” functions, such as adaptive filters, correlation, spectral estimation, discrete cosine transforms, etc. [5]. They are in general not 100% suitable for all operations in an image codec though, which makes the design of custom-made instructions appealing. Adding simple instructions will not increase the chip area by much, so it may be a wise trade-off between performance and complexity. The Hardware/Software partitioning problem is an optimization problem with constraints such as silicon area, power consumption, money expenses, time-to-market and execution time.

6.3 Stream Reading

Every time a bit is consumed, there is a number of operations involved to extract this bit from a word. Since this occurs roughly once for every fifth pixel or so, it isn’t extremely frequent, but is does have some impact on performance since it takes a dozen clock cycles once it happens. Implementing such an instruction would also benefit most other codecs that consume bit-values rather than byte-values. Implementing an instruction of this kind could save 1-2 MIPS for a QCIF video depending on the bitrate.

6.4 Usage of 32-bit shift operations

The MQ decoder often performs shift operations on the 32-bit C register. This happens during the renorme operation. The architecture for this DSP processor has only 16-bit general registers aside from the MAC registers, which makes the decoder a little bit slower. A simple, but yet powerful, new instruction would be a double-precision left-shift operation. Consider the code of the pure software implementation²:

```
move gr8, constant
move ACR1.h , gr1
move ACR1.l , gr0
imul ACR1, gr8
move gr1, ACR1.h
move gr0, ACR1.l
```

This could be replaced with this single instruction:

```
DLSHIFT gr1, gr2
```

This will only require 1 clock cycle compared to the 6 that is required for the previous operation. Renormalization happens on roughly 2/3 of all calls [6], so about 4 clock cycles per decoded bit will be saved. This is a performance boost for

---
² See the appendix for an explanation of the instructions
the MQ-decoder of about 10%, and will in total save around 0.5 MIPS in the QCIF video example.

6.5 Fast Table Lookups

There are four different table lookups during the context formation. The address offset in the table lookup is calculated in a rather time-consuming way. It is worth the effort to design instructions that calculates these more efficiently. It should be pointed out that all these table-lookup instructions are very application-specific for JPEG2000. On the other hand, they would also be useful to an encoder in a similar way. The original offset calculation in a pure software implementation will now be compared to the new instructions. These operations are made in accordance with the mapping presented in the appendix.

6.5.1 Zero Coding Context (ZCC)

Left shift value A by eight, AND operation on value B with a constant K1 and finally an OR operation on the results.

```
move gr2, gr1
lshift gr2, 8
move gr3, gr0
and gr3, 0x00FF
or gr2, gr3
```

A new instruction would do this calculation in these steps:

```
ZCC1 gr2, gr1
ZCC2 gr2, gr0
```

A processor architecture that allows three operands would of course be able to do this in one cycle.

6.5.2 Sign Prediction Bit (SPB)

AND operation on the lookup value with a constant K2 and then right shift by 4.

```
move gr2, 0x0FF0
and gr2, gr0
lshift gr2, 4
```

The new instruction would do this in one step:

```
SPB gr2, gr0
```

---

3 Other flag mappings are of course possible, but these seem to be efficient. The instructions would be different if the flags bits were mapped in another way.
This saves only two clock cycles though.

6.5.3 Sign Coding Context (SCC)

Same as for SPB, except that is applied on another table.

6.5.4 Magnitude Context (MAGC)

AND operation on the lookup value with a constant K1, then set the 11\textsuperscript{th} bit to the inverted value to the inverted 14\textsuperscript{th} bit of the lookup value.

```
move gr2, 0x00FF
move gr4, 0x0800
move gr5, 0
and gr2, gr0
move gr3, 0x2000
and gr3, gr0
if ane move gr5, gr4
or gr3, gr5
```

The new instruction would do this in one step:

```
MAGC gr2, gr0
```

This saves 7 clock cycles, which makes it the most efficient improvement for the lookup offset calculation. This speed up the refinement pass steps with 20-25\% fairly easily (counting those passes that actually are computed). The implementation of this instruction is not very complicated, since it is possible to perform mainly with an inverter for one bit and rearrange some bits or set them to zero.

The benefits for the significance propagation pass is considerably smaller as only 8 clock cycles can be saved in total, which gives only about a 10\% reduction since the flag updating algorithm requires so many clock cycles. The same principle applies to the cleanup pass.

6.6 JUMPIFZ

While doing a lot of flag-checking in the coding passes, it is obvious that instructions that could make this run faster also would have a significant impact on performance. Consider the following instruction:

```
JUMPIFZ <address-offset> grX
```

Where grX is an arbitrary general purpose register and the address offset is an immediate operand, representing a 6-bit signed integer with the relative jump address. A pointer to the specified bit could be held in a pre-mapped general
purpose register to save bits for the instruction length. Originally, we may have a code that looks like this:

```
move gr0, constant ; constant contains the flag
move gr1, 0
and gr2, gr0
comp gr2, gr1
if aeq jump <address>
```

This now could be replaced with:

```
move gr0, bit-pointer ; gr0 is pre-mapped to "jumpifz"
jumpifz <address> gr2
```

Such an instruction is fairly simple to implement; a 4-bit multiplexer that selects a specific bit from the grX register with (in this case) gr0 as an input, and logic that assigns the immediate operand to the program counter.

The benefits are also fairly large, since it may save 2-3 clock cycles per coding pass for a bitplane. It is also of interest not only to JPEG2000, but probably also to other codecs that need to check states frequently. For the previously mentioned QCIF video, this saves roughly 2 MIPS per bitplane that is used on average. If the average number of planes is approximately 3, it could save around 6 MIPS. (Note that no deep analysis have been made to estimate the number of bitplanes per coefficient on a typical lossy image. Lossless compression would have 8 though.)

### 6.7 Summary

The benefits of instruction-level optimizations are not insignificant, but it does not boost performance to new levels actually. The numbers apply to the previously mentioned QCIF video example.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Impacts</th>
<th>MIPS benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZCC</td>
<td>3 cs on significance and cleanup passes</td>
<td>0.2</td>
</tr>
<tr>
<td>SPB</td>
<td>4 cs on significance and cleanup passes</td>
<td>0.3</td>
</tr>
<tr>
<td>MAGC</td>
<td>7 cs on refinement pass</td>
<td>0.5</td>
</tr>
<tr>
<td>JUMPIFZ</td>
<td>3 cs when interrupting any pass</td>
<td>Up to 6</td>
</tr>
<tr>
<td>DLSHIFT</td>
<td>5 cs during MQ-decoder renormalization</td>
<td>0.5</td>
</tr>
<tr>
<td>Stream-reading</td>
<td>2 cs when feeding decoder with the bitstream</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Note that the most powerful hardware instruction is also the one with the most imprecise MIPS benefit. While being very useful often, it is also very dependent on the average number of bitplanes for the codeblocks.
7 Further Hardware Acceleration

For demanding applications, a dedicated hardware solution or more advanced general purpose digital signal processors will be needed in order to achieve sufficient performance. This chapter intends to propose improvements in this area.

7.1 A Superscalar DWT solution

It is not very hard to parallelize the discrete wave transform for DSP processors that have several MAC units. 2-way superscalar DSPs handling wavelet transforms would almost get a linear improvement over single-scalar DSP. Since the DWT accounts to roughly a half of the total computational time, it is well worth considering. Processors that are 4-way superscalar will not benefit from the extra MAC units because of the dependencies shown in the equations of the lifting implementation. It would also be beneficial to implement dedicated inversed DWT hardware, but this will of course be more expensive when it comes to chip area and design time.

7.2 Parallelization of context formation

The context formation process of the EBCOT is not very easy to perform in parallel, due to many pipeline stalls. The general idea for my proposal is to perform a stripe column in parallel. The reason is because it is related to the natural scanning order (and the MQ decoder gives its output sequentially). If any further acceleration is to be made for the context formation engine, it should probably be easier (and more resource-consuming) to use multiple units that work independently on different code-blocks.

| Dependencies between coding pass operations (excluding previous MQ-decoding) |
|---------------------------|--------------------------------------------------------------------------|
| **Significance propagation** |                                                                         |
| Immediate exit            | Cleanup pass results from previous bitplane                              |
| ZCC context lookup        | Significance-pass for previous coefficient                                |
| SCC context lookup        | Significance-pass for previous coefficient                                |
| SPB lookup                | Significance-pass for previous coefficient                                |
| **Magnitude refinement**  |                                                                         |
| Immediate exit            | Significance pass for this coefficient                                    |
| MAGC context lookup       | Significance and magnitude pass and for *this* coefficient               |
### 7.2.1 Significance propagation pass

The scanning of the southernmost pixel is delayed until the process of the former pixel has called the MQ-decoder for the second time. However, if the time-consuming flag updating operation is needed, it can be made in parallel with many table lookups.

### 7.2.2 Magnitude refinement pass

It is easier to parallelize the refinement pass because of two reasons: First of all, the table lookups aren’t depending on the flags that may be set during the context formation of the previous pixel. Secondly, the calculation of the new magnitude (after the information was received from the MQ-coder) can be made in parallel. This means that with a fast hardware-accelerated MQ-decoder, the refinement pass can be pipelined quite efficient and utilize superscalar general purpose DSPs.

### 7.2.3 Cleanup pass

The cleanup pass is quite similar to the significance propagation pass when it comes to the possible scheduling. Thus, it is mainly the flag updating that can be executed in parallel.

### 7.3 2-dimensional addressing

During the flag updating process, the calculation of addresses to surrounding pixels takes up a big part of the time. Hence, an address calculation unit that can handle 2-dimentional addressing may speed up this significantly. The address offset between rows requires an additional register $R_{row}$. In that case, I propose an addressing mode that may operate like this:

```plaintext
move grX, dm0(daY, +1, +Rrow)
moves grX, dm0(daY, -1, -Rrow)
```

Where the first instruction would perform a move instruction from the southeast neighbor and the second instruction would perform a move from the northwest neighbor. GrX denotes an arbitrary general purpose register, dm0 is the memory bank and daY is an arbitrary address register. This would speed up the

<table>
<thead>
<tr>
<th>Cleanup</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate exit</td>
<td>Significance pass result for this coefficient</td>
</tr>
<tr>
<td>ZCC context lookup</td>
<td>Cleanup-pass for previous coefficient</td>
</tr>
<tr>
<td>SCC context lookup</td>
<td>Cleanup-pass for previous coefficient</td>
</tr>
<tr>
<td>SPB lookup</td>
<td>Significance-pass for previous coefficient</td>
</tr>
</tbody>
</table>
flag-updating process from 39 clock cycles to *around* 25 (for causal mode, and similar improvements for non-causal mode). However, this is far from a simple solution and the author will refrain from speculating about how complex the implementation of such an addressing unit could be.

### 7.4 MQ-decoder

This department has already treated this in a previous master thesis [6], so no more proposals will be presented in this thesis. That parallel hardware solution made a 50-fold speedup compared to a pure software implementation. The need for a hardware MQ-decoder is not at all impossible to meet if there is time to implement this in a product.
8 Conclusions

JPEG2000 decoders are computational complex, with context formation and wavelet transforms being the most intensive tasks. My results show that a careful redesign of the instruction sets may result in significant performance improvements, although the increase of about 20% is not enough for higher bandwidths. A VLSI implementation of inverse DWT seems to be the easiest way to increase performance for high-performance systems. The proposed instructions are easy to implement in hardware, while some of them are very specific for the JPEG2000.

For a high-performance JPEG2000 decoder, the most critical path is the context formation for the Significance Propagation and Cleanup passes. These will probably become the bottleneck of any high-performance solution.

A DSP processor with better instructions for program flow control (e.g. JUMPIFZ), 2-dimensional addressing, a superscalar MAC architecture and more registers would prove to be quite optimal for a decoder of this kind.
9 Bibliography


## 10 Appendix

### 10.1 State register mapping for context formation

<table>
<thead>
<tr>
<th>Flag information</th>
<th>Meaning</th>
<th>Bit position</th>
</tr>
</thead>
<tbody>
<tr>
<td>NESIG</td>
<td>Northeast neighbor found to be significant</td>
<td>0</td>
</tr>
<tr>
<td>SESIG</td>
<td>Southeast neighbor found to be significant</td>
<td>1</td>
</tr>
<tr>
<td>SWSIG</td>
<td>Southwest neighbor found to be significant</td>
<td>2</td>
</tr>
<tr>
<td>NWSIG</td>
<td>Northwest neighbor found to be significant</td>
<td>3</td>
</tr>
<tr>
<td>NSIG</td>
<td>North neighbor found to be significant</td>
<td>4</td>
</tr>
<tr>
<td>ESIG</td>
<td>East neighbor found to be significant</td>
<td>5</td>
</tr>
<tr>
<td>SSIG</td>
<td>South neighbor found to be significant</td>
<td>6</td>
</tr>
<tr>
<td>WSIG</td>
<td>West neighbor found to be significant</td>
<td>7</td>
</tr>
<tr>
<td>NSGN</td>
<td>North neighbor is negative</td>
<td>8</td>
</tr>
<tr>
<td>ESGN</td>
<td>East neighbor is negative</td>
<td>9</td>
</tr>
<tr>
<td>SSGN</td>
<td>South neighbor is negative</td>
<td>10</td>
</tr>
<tr>
<td>WSGN</td>
<td>West neighbor is negative</td>
<td>11</td>
</tr>
<tr>
<td>SIG</td>
<td>This coefficient is found to be significant</td>
<td>12</td>
</tr>
<tr>
<td>REFINE</td>
<td>This coefficient has been refined at least once</td>
<td>13</td>
</tr>
<tr>
<td>VISIT</td>
<td>This sample has been visited by significance pass</td>
<td>14</td>
</tr>
</tbody>
</table>
10.2 Assembler instructions syntax

A brief explanation of the instruction set for the processor that has been used:
Operations could be made on 0-2 operands, depending on the instructions.
General Registers are numbered between gr0 to grF.
There are two tap memories (DM0 and TM0).
There are two address registers for each tap memory (da0, da1, tm0, tm1).
There are two accumulator registers (ACR0 and ACR1), the upper and lower byte can be accessed by regular MOVE operations.

Some instructions in the examples:
Move Moves content between registers
Imul Multiplication in the accumulator register
Lshift Left shift (may use immediate operand)
And Bitwise and operation (may not use immediate operand)
Or Bitwise and operation (may not use immediate operand)
if ane Depending on the flags set by previous instruction (result = not equal), perform an instruction
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Per Nilsson]