Wavelet based image compression using FPGAs

Dissertation

zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)

vorgelegt der

Mathematisch-Naturwissenschaftlich-Technischen Fakultät
der Martin-Luther-Universität Halle-Wittenberg

von Herrn Jörg Ritter

geb. am 12. April 1971 in Greiz

Gutachter:

1. Prof. Dr. Paul Molitor, Martin-Luther-University Halle-Wittenberg, Germany
2. Prof. Dr. Scott Hauck, University of Washington, Seattle, WA, USA
3. Prof. Dr. Thomas Rauber, University of Bayreuth, Germany

Halle (Saale), 06.12.2002

urn:nbn:de:gbv:3-000004615
[http://nbn-resolving.de/urn/resolver.pl?urn=nbn%3Ade%3Agbv%3A3-000004615]

Abstract

In this work we have studied well known state of the art image compression algorithms. These codecs are based on wavelet transforms in most cases. Their compression efficiency is widely acknowledged. The new upcoming JPEG2000 standard, e.g., will be based on wavelet transforms too. However, hardware implementations of such high performance image compressors are non trivial. In particular, the on chip memory requirements and the data transfer volume to external memory banks are tremendous.
We suggest a solution which minimizes the communication time and volume to external random access memory. With negligible internal memory requirements this bottleneck can be avoided using the partitioned approach to wavelet transform images proposed in this thesis. Based on this idea we present modifications to the well known algorithm of Said and Pearlman *Set Partitioning In Hierarchical Trees SPIHT* to restrict the necessity of random access to the whole image to a small subimage only, which can be stored on chip. The compression performance in terms of visual property (measured with peak signal to noise ratio) compared to the original codec remains still the same or nearly the same. The computational power of the proposed circuits targeting to programmable hardware are promising. We have realized a prototype of this codec in a XC4000 Xilinx FPGA running at 40MHz which compresses images 10 times faster than a 1GHz Athlon processor. An application specific integrated circuit based on our approach should be much faster over again.

Zusammenfassung

Im Rahmen dieser Disseration haben wir die besten, derzeit verfügbaren Bildkompressionsverfahren analysiert. Die meisten dieser Algorithmen basieren auf Wavelet-Transformationen. Durch diese Technik können erstaunliche Ergebnisse im Vergleich zu den bekannten, auf diskreten Kosinus-Transformationen basierenden Verfahren erreicht werden. Unterstrichen wird diese Tatsache durch die Wahl eines waveletbasierten Kodierers von der Joint Picture Expert Group als Basis für den neuen JPEG2000 Standard. Die Entwicklung von speziellen Hardware-Architekturen für diese Kompressionsalgorithmen ist sehr komplex, da meist viel interner Speicher benötigt wird und zusätzlich riesige Datenmengen zwischen Peripherie und dem Hardware-Baustein ausgetauscht werden müssen.
Wir stellen in dieser Arbeit einen Ansatz vor, welcher ausgesprochen wenig internen Speicher erfordert und gleichzeitig das zu transportierende Datenvolumen drastisch reduziert. Aufbauend auf diesem partitionierten Ansatz zur Wavelet-Transformation von Bildern stellen wir Modifikationen des bekannten Kompressionsverfahrens von Said und Pearlman *Set Partitioning In Hierarchical Trees SPIHT* vor, um diesen effektiv in Hardware realisieren zu können. Diese Veränderungen sind notwendig, da im Original extensiv von dynamischen Datenstrukuren Gebrauch gemacht wird, welche sich nicht oder nur mit erheblich Aufwand an Speicher realisieren lassen. Die visuelle Qualität im Vergleich zum Originalalgorithmus bleibt jedoch exakt gleich oder ist dieser sehr ähnlich. Jedoch sind die Geschwindigkeitsvorteile unserer Architektur gegenüber aktuellen Prozessoren von Arbeitsplatzrechnern sehr vielversprechend, was wir durch praktische Versuche auf einem programmierbaren Hardware-Baustein überzeugend nachweisen konnten. Wir haben einen Prototypen auf einem Xilinx XC4000 FPGA realisiert, welcher mit 40 MHz getaktet werden konnte. Schon dieser Prototyp des Hardware-Bildkomprimierers komprimiert Bilder 10 mal schneller als ein Athlon Prozessor getaktet mit 1GHz. Ein mit ensprechender Technologie basierend auf unserem partitioniertem Ansatz produzierter anwendungsspezifischer Schaltkreis würde diese Leistung noch bei weitem übertreffen.

Contents

Introduction

Reading a user manual of a new mobile phone you may wonder whether it is possible to make a phone call at all. We grant that this is overstated. But besides the central functionality of a mobile phone there are a lot of additional features. Many of them fall into the category of multimedia applications. You can play music, hear radio, read and write emails, and surf in the internet. However, even with new connection standards like GPRS, HSCD, or UMTS, which provide high speed data transfers, the bandwidth is limited. Data compression and error resilience in noisy environments like transfers to mobile phones are basically to provide those multimedia features. In contrast to a personal computer where you can rely on powerful processors with huge main storage capabilities one has to think about low cost hardware solutions, here.

This is also the case for digital cameras. The photograph expects that immediately after taking a picture he can inspect the result. To support this behavior the picture has to be compressed, stored on a flash memory card, decompressed, and shown at a LCD display in nearly real time. Features like high speed previews with incremental refinement have to be provided. Furthermore the digital photograph could expect, that more than say 36 pictures can be stored on the memory stick. Therefore efficient hardware image compression algorithms with excellent visual properties are necessary.

Even surfing the world wide web using a powerful personal computer and high speed internet access we often have to wait until web sites are rendered. Basicly not the searched information itself determines the data transfer volume but the presentation of it and accessory advertising. These product presentations heavy rely on color illustrations or animations. Thus the surfer and manufacturer are interested in efficient data compression methods, too. The potential customer finds the information he was looking for in shorter time and the supplier saves money for server farms to provide huge bandwidth.

In this work we have studied well known state of the art image compression algorithms. These codecs are based on wavelet transformations in most cases. Their compression efficiency is widely acknowledged. The new upcoming JPEG2000 standard will be based on wavelet transformations, too. Hardware implementations of such high performance image compressors are non trivial. Especially the on chip memory requirements and the data transfer volume to external memory banks are tremendous.

We suggest a solution which minimizes the communication time and volume to external random access memory. With negligible internal memory requirements this bottleneck could be avoided using a partitioned approach to wavelet transform images. Based on this idea we propose modifications to the well known algorithm of Said and Pearlman *Set Partitioning in Hierarchical Trees SPIHT* to restrict the necessity of random access to the whole image to a small subimage only, which can be stored on chip. However, the compression performance in terms of visual property (measured with peak signal to noise ratio) compared to the original codec is still the same or nearly the same. The computational power of the proposed circuits targeting to programmable hardware are promising. We have realized a prototype of this codec in a XC4000 Xilinx FPGA running at 40MHz which compresses images 10 times faster than a 1GHz Athlon processor. An application specific integrated circuit based on our approach should be much faster over again.

This thesis is structured as follows.

Mathematical background In Chapter 1 we introduce necessary notations and discuss up/downsampling and delaying of discrete signals in detail. These techniques will be very useful in Chapter 3. Furthermore we give a short overview of the coherence of wavelet transforms and filtering.

Wavelet transform on images The digital representation of images and the tensor product of one dimensional wavelet transforms applied to images are introduced in Chapter 2. We also discuss the implicit storage of the normalization factors, which are necessary to preserve the average brightness of the images.

Range of CDF(2,2) wavelet coefficients In Chapter 3 the dynamic range of coefficients after a wavelet transformation had taken place is analyzed. We deduce lower and upper bounds for the endpoints of the corresponding intervals for each scale and orientation. Theses bounds are used to reduce the memory requirements down to the minimum.

State of the art image compression techniques Before we present the central part of this thesis, we give a survey of state of the art wavelet based image compression techniques in Chapter 4. We shortly outline the embedded zerotree wavelet algorithm of Shapiro and discuss the *SPIHT* algorithm in more detail. This codec is widely acknowledged as a reference for effective image compression methods with excellent visual properties of the reconstructed images.

Partitioned Approach In Chapter 5 we present in detail the main contribution of this thesis, the partitioned approach to wavelet transform images. At first we motivate, why there is a need for such an approach, if one considers implementations of image compression algorithm based on wavelet transforms in programmable hardware. Furthermore to achieve efficient circuits with respect to clock rate and data throughput we present necessary modifications of the original *SPIHT* codec and compare them in terms of visual quality.

FPGA Architectures The designs are discussed in Chapter 6. Here we present VHDL projects for the partitioned approach itself and the appropriate adapted *SPIHT* codec. We discuss our prototyping environment, a PCI card equipped with a Xilinx FPGA, which offers the opportunity to us to derive convincing experimental result.

Conclusions and Related Work To conclude this thesis we summerize our obtained results in terms of theoretical aspects as well as practical experiences. Furthermore the features of the upcoming new JPEG2000 standard and the integrated compressor named EBCOT are exemplified. We balance the advantages and disadvantages of our approach to the proposed method in the JPEG2000 standard.

Acknowledgements I thank my advisor Prof. Dr. Paul Molitor who arouses my interest in wavelet based image compression and for all the fruitful discussions. Special thanks to Stepan Sutter, Henning Thielemann, and Görschwin Fey for their corporation in the last years. Thanks to all my colleagues whose feedback has allowed me to improve the contents of this thesis. A special acknowledgment to Sandro Wefel for his continuing support throughout the whole process of developing and writing this thesis.

Chapter 1

Mathematical Background

Wavelet based image compression techniques are funded on several fundamental mathematical theories. The wavelet transform used to decorrelate the input signal has its theoretical roots in several traditional sciences like Fourier analysis, signal processing, or filter theory. Here we cannot give a detailed introduction to all concepts. We restrict our explanations to the central notations, which are necessary for the understanding of this thesis.

1.1 Discrete Signal and Filters

Discrete signals and filters can be represented by vectors. In many cases we do not distinguish between signals of finite or infinite length. A signal x is written as

$$x \quad = \quad (\dots, x_{-2}, x_{-1}, x_0, x_1, x_2, \dots),$$

with coefficients $x_n \in \mathbb{Z}$, $\mathbb{R}$, or $\mathbb{C}$ for all $n \in \mathbb{Z}$. Analogously, a filter f with coefficients $f_m \in \mathbb{Z}$, $\mathbb{R}$, or $\mathbb{C}$ for all $m \in \mathbb{Z}$ (often called *filter taps*) is declared by

$$f \quad = \quad (\dots, f_{-2}, f_{-1}, f_0, f_1, f_2, \dots).$$

Sometimes it is necessary to locate the coefficient at index zero. We then emphasize that coefficient like this

$$x \quad = \quad (\dots, 0.25, \mathbf{1.25}, -0.5, 0.75, \dots).$$

Usually, the signal and filter coefficients are non zero at finite positions only. If so, the corresponding signal and filters have so called *finite support*. Let f_a and f_b be the right most and left most non zero coefficient, respectively. The *number of taps* or similarly the *filter length* is then defined as

$$|f| = b - a + 1.$$

In the filter theory one distinguishes between *finite impulse response (FIR)* and *infinite impulse response (IIR)* filters. The impulse response of a filter f is the filter output if the filter input is the signal δ defined by

$$\delta_n = \begin{cases} 1 & : \quad n = 0 \\ 0 & : \quad \text{else} \end{cases},$$

where all samples are zero except one. If the impulse response is finite, because there is no feedback in the filter, the filter is called *FIR filter*. This category of filters has some important advantages:

- simple to implement (MAC operations (multiply and accumulate)),
- desirable numeric properties (with respect to fixed and floating point arithmetic), and
- they can be designed to be *linear phase*. Linear phase filters do not distort the phase by introducing some delay if (and only if) its coefficients are symmetrical around the center coefficient.

In this thesis we will restrict to FIR filter and we use the notion *filter* as an acronym of FIR filter.
The *convolution* $y = f * x$ of a signal x with a filter f is the signal y with

$$y_n := \sum_{k\in\mathbb{Z}} f_k x_{n-k} = \sum_{k\in\mathbb{Z}} f_{n-k} x_k.$$

1.1.1 z-Transform

In Chapter 3 we will estimate greatest lower and least upper bounds for the minimal bitwidth necessary to store coefficients as result of wavelet transform. In order to compute accurate estimations we have to analyze filtered signals. A helpful technique will be the so called *z-transform*. This is a generalization of the discrete time Fourier transform. It is defined for filters and signals by

$$F(z) = \sum_{l\in\mathbb{Z}} f_l z^{-l} \qquad \text{and} \qquad X(z) = \sum_{k\in\mathbb{Z}} x_k z^{-k}, \tag{1.1}$$

respectively, where $z \in \mathbb{C}$, i.e., the variable z is of type complex number. Assuming that the *z-transform* and its inverse exist, we will denote by

$$f \longleftrightarrow F \qquad \text{and} \qquad x \longleftrightarrow X$$

a transform pair. Note that for discrete signals with finite support the *region of convergence* (ROC) is the whole z-plane, e.g., the series given in Equation (1.1) converge for all z with $0 < |z| < \infty$. The relation $f \longleftrightarrow F$ is a one-to-one mapping, if f is a discrete signal or filter with finite support. For a detailed discussion of the properties of *z-transforms* we refer to [OS89].

For a filtered signal $y = f * x$ the *convolution theorem*

$$y = f * x \longleftrightarrow Y = F \cdot X,$$

holds since

$$\begin{aligned}
Y(z) &= \sum_{n\in\mathbb{Z}} y_n z^{-n} \\
&= \sum_{n\in\mathbb{Z}} \left(\sum_{k\in\mathbb{Z}} f_k x_{n-k} \right) z^{-n} \\
&= \sum_{k\in\mathbb{Z}} \sum_{n\in\mathbb{Z}} f_k x_{n-k} z^{-n} \\
&= \sum_{k\in\mathbb{Z}} f_k \sum_{n\in\mathbb{Z}} x_{n-k} z^{-n} \\
&= \sum_{k\in\mathbb{Z}} f_k \sum_{n'\in\mathbb{Z}} x_{n'} z^{-n'-k} \\
&= \sum_{k\in\mathbb{Z}} f_k z^{-k} \sum_{n\in\mathbb{Z}} x_n z^{-n} \\
&= F(z)X(z)
\end{aligned}$$

for all $z \in \mathbb{C}$.

1.1.2 Impulse train function

Beside filtering the upsampling, downsampling, and delaying of a signal are of interest, too. Here the *impulse train function* plays an important rule. It is defined by

$$\delta_n^{(m)} = \sum_{k\in\mathbb{Z}} \delta_{n-k\cdot m} = \begin{cases} 1 & : \exists k' \in \mathbb{Z} : n = k' \cdot m \\ 0 & : \text{else}, \end{cases}$$

where m denotes the distance between the one's. This can also be written as

$$\delta_n^{(m)} = \frac{1}{m} \sum_{k=0}^{m-1} e^{\frac{i2\pi kn}{m}}, \tag{1.2}$$

where i is the imaginary unit, using the properties of the mth roots of unity in the complex plane. The equation $z^m = 1$ has m solutions for $z \in \mathbb{C}$ [For83]. These solutions are

$$e^{\frac{i2\pi k}{m}} \quad \text{for} \quad 0 \leq k < m,$$

which are illustrated in Figure 1.1(a) for $m = 7$. The fact, that Equation (1.2) holds, is based on the symmetry of the roots of unity. If n is a multiple of m, then the sum in Equation (1.2) evaluates to one, namely for $n = k' \cdot m$ we obtain

$$\begin{aligned}
\delta_n^{(m)} &= \frac{1}{m} \sum_{k=0}^{m-1} e^{\frac{i2\pi kn}{m}} \\
&= \frac{1}{m} \sum_{k=0}^{m-1} \left(e^{\frac{i2\pi k}{m}}\right)^n \\
&= \frac{1}{m} \sum_{k=0}^{m-1} \left(e^{\frac{i2\pi k}{m}}\right)^{k' \cdot m} \\
&= \frac{1}{m} \sum_{k=0}^{m-1} 1^{k'} = 1.
\end{aligned}$$

In the case, that n is not a multiple of m, you sum up the mth roots of unity in the way as illustrated in Figure 1.1(b). The sum is then equal zero. In the figure we have chosen $m = 5$. For instance, if $n = 1$ then you take all roots. The red arrows shows the order, in which the roots are sum up for $n = 1$, starting at e^0. If $n = 2$ you take only every second root, if $n = 3$ you take only every third root and so on. But these roots are equally spaced around the unit circle. You can examine this behavior for the case $n = 2$ in Figure 1.1(b) following the green arrows. You could also imagine the unit circle as a disc in the three dimensional space mounted only at the center point and further that on each mth root, which appears in the sum a weight is mounted. Then the disc will be stable in terms of gravity.

In the special case $m = 2$ we get the sequence

$$\begin{aligned}
\delta^{(2)} &= (\ldots, 0, 1, 0, \mathbf{1}, 0, 1, \ldots) \text{ or} \\
\delta_n^{(2)} &= \frac{1}{2}(1 + (-1)^n),
\end{aligned}$$

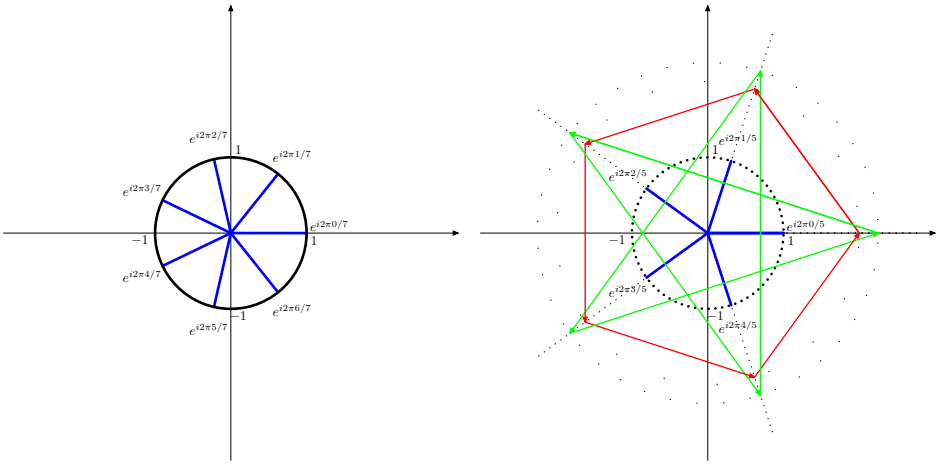
and for $m = 4$

$$\begin{aligned}
\delta^{(4)} &= (\ldots, 0, 0, 0, 1, 0, 0, 0, \mathbf{1}, 0, 0, 0, 1, \ldots) \text{ or} \\
\delta_n^{(4)} &= \frac{1}{4}(1 + i^n + (-1)^n + (-i)^n).
\end{aligned}$$

These considerations will be useful in Section 1.4 and Chapter 3.

1.2 Measure of Information

In this section we give some basics of information theory in order to introduce the notation of *entropy* of a discrete signal. *Randomness* is in this context a basic concept. Consider the tossing of a coin, where the outcome can come up with head or tail, typically this happens randomly. A *random variable* $\mathcal{X}$ is defined to be a quantity, such as the position of a particle, which has many possible outcomes, together with the likelihood of each outcome being specified. If an event is certain to occur, its probability is taken to be 1, and if an event is impossible, its probability is taken to be 0. The likelihood of any event occurring, namely



(a) roots of unity, $z^7 = 1$

(b) representation of $\delta_n^{(5)}$ in Equation (1.2)

Figure 1.1: (a) the roots of unity for $m = 7$, (b) the intersection of the blue lines with the unit circle are the 5th roots of unity, the red and green arrows shows the order, in which the roots are sum up in Equation (1.2) for $\delta_n^{(1)}$ and $\delta_n^{(2)}$ respectively (starting with e^0).

p, is always positive and must lie between 0 (impossibility) and 1 (certainty), that is $0 \leq p \leq 1$. A random variable $\mathcal{X}$ whose outcomes α are discrete, such as that of tossing a coin, is a *discrete random variable*. These outcomes α are usually elements a_i of an alphabet $\mathcal{A}_\mathcal{X} = \{a_1, a_2, \ldots, a_k\}$ where the probability, that the ith outcome is a_i, is given as $p_\mathcal{X}(\alpha == a_i)$. This probability will be in short denoted by $p(a_i)$, if this is clear.
Shannon [Sha] defined a quantity called *self information*. If the outcome of a random variable $\mathcal{X}$ is a_i with probability $p(a_i)$, then the self information is given by

$$\log \frac{1}{p(a_i)} = -\log p(a_i) \tag{1.3}$$

Now we have all necessary terms to define the notion of *entropy*.

Definition 1 *The entropy $H(\mathcal{X})$ of a random variable $\mathcal{X}$ with the given alphabet $A_\mathcal{X}$ and the probabilities $p_\mathcal{X}$ is defined by*

$$\begin{aligned} H(\mathcal{X}) &= -\sum_i p_\mathcal{X}(\alpha == a_i) \log p_\mathcal{X}(\alpha == a_i) \\ &= -\sum_i p(a_i) \log p(a_i) \end{aligned} \tag{1.4}$$

Shannon showed, that the entropy is a measure of the average number of binary symbols needed to encode the output of a source (e.g. a random variable). For lossless compression it follows, that the best we can do is to encode the output of a source with an average number of bits equal to the entropy of the source.

We will use the entropy as a measure of the compression efficiency in order to compare the quality of different lossless encoders.

1.3 Distortion Measures

For lossy compression it is important to know how much a modification of the transformed signal distorts the restored signal. One might expect that a small modification of the transformed signal causes small

distortions in the restored data. But there is an uncertainty in general. Consider an input signal x (e.g., an image data) and the operation W which is performed by a complete wavelet transform (see Section 1.6 for explanations). The transformed signal Wx is now modified by a lossy compression.

Let us assume that the lossy compression process outputs signal α. Because every wavelet transform applied for compression must be invertible, there exists a signal y such that $Wy = \alpha$. Thus input signal y is restored by the decompression step.

We are looking for an accurate estimation of how much the original signal changes if a modification occurs on the transformed signal, in other words, how much distortion is introduced in the compression process. For measuring the difference of two signals or the introduced distortion the *signal to noise ratio* (SNR) and the *peak signal to noise ratio* (PSNR) (see [TM02], [Say96]) are widely used. They provide a compromise between visual perception and easiness of computation.

Both the SNR and the PSNR are logarithmic scaled forms of the EUCLIDean metric of two vectors/signals x and y

$$\|x - y\|_2 = \sqrt{\sum_i (x_i - y_i)^2}$$

where the possible value range of the sampled data and the number of samples have an influence, too.
Let x, y be signals, each consisting of n values with a possible range of $[0, x_{\max}]$ (e.g. $[0, 255]$ for 8 bit images), then

- the *mean squared error* MSE is defined by

$$\mathrm{MSE}(x, y) \quad := \quad \frac{1}{n} \sum_{i=0}^{n-1} (x_i - y_i)^2$$

- the *signal to noise ratio* SNR is defined by

$$\mathrm{SNR}(x, y) \quad := \quad 10 \log_{10} \frac{\|x\|_2^2}{\|x - y\|_2^2} \,\mathrm{dB}$$

- the *peak signal to noise ratio* PSNR is defined by

$$\begin{aligned} \mathrm{PSNR}(x, y) \quad &:= \quad 20 \log_{10} \frac{x_{\max} \cdot \sqrt{n}}{\|x - y\|_2} \,\mathrm{dB} \\ &= \quad 10 \log_{10} \frac{x_{\max}^2}{\mathrm{MSE}(x, y)} \,\mathrm{dB} \end{aligned}$$

where the units of measurement are *decibels* (abbreviated to dB).

1.4 Downsampling, Upsampling, and Delay

Downsampling by m_d and *upsampling by* m_u are used to express wavelet transforms in terms of filter operations ($m_d, m_u \in \mathbb{N}$). That is, after filtering all samples with indices modulo m_d different from zero are discarded or $m_u - 1$ samples are inserted at every index, respectively.

Downsampling a sequence x by m_d can be expressed as

$$y_n = x_{n \cdot m_d}$$

or in z-transform domain

$$Y(z) = \frac{1}{m_d} \sum_{k=0}^{m_d-1} X(e^{\frac{-i2\pi k}{m_d}} z^{\frac{1}{m_d}})$$

with $y \longleftrightarrow Y$ and $x \longleftrightarrow X$. This can be shown using Equation (1.2)

$$\begin{aligned}
Y(z) &= \sum_{k\in\mathbb{Z}} x_{km_d} z^{-k} \\
&= \sum_{n\in\mathbb{Z}} x_n \delta_n^{(m_d)} z^{\frac{-n}{m_d}} \\
&= \sum_{n\in\mathbb{Z}} x_n \left(\frac{1}{m_d} \sum_{k=0}^{m_d-1} e^{\frac{i2\pi kn}{m_d}} \right) z^{\frac{-n}{m_d}} \\
&= \frac{1}{m_d} \sum_{k=0}^{m_d-1} \left(\sum_{n\in\mathbb{Z}} x_n e^{\frac{i2\pi kn}{m_d}} z^{\frac{-n}{m_d}} \right) \\
&= \frac{1}{m_d} \sum_{k=0}^{m_d-1} \left(\sum_{n\in\mathbb{Z}} x_n \left(e^{\frac{-i2\pi k}{m_d}} z^{\frac{1}{m_d}} \right)^{-n} \right) \\
&= \frac{1}{m_d} \sum_{k=0}^{m_d-1} X\left(e^{\frac{-i2\pi k}{m_d}} z^{\frac{1}{m_d}} \right).
\end{aligned}$$

Upsampling a sequence x by m_u can be expressed as

$$y_n = \begin{cases} x_{n/m_u} & : \quad \exists k \in \mathbb{Z} : n = k \cdot m_u \\ 0 & : \quad \text{else} \end{cases}$$

or in z-transform domain

$$\begin{aligned}
Y(z) &= \sum_{n\in\mathbb{Z}} y_n z^{-n} \\
&= \sum_{k\in\mathbb{Z}} x_k z^{-(k\cdot m_u)} \\
&= \sum_{k\in\mathbb{Z}} x_k \left(z^{m_u}\right)^{-k} \\
&= X(z^{m_u})
\end{aligned}$$

with $y \longleftrightarrow Y$ and $x \longleftrightarrow X$.

The downsampling and upsampling operation for signal x are denoted by $x \downarrow m_d$ and $x \uparrow m_u$, respectively. In the following illustrations we will depict both operations with the symbols $\boxed{\downarrow m_d}$ and $\boxed{\uparrow m_u}$, respectively.

In order to discard even or odd indexed samples we also need the term *delay by* m_{dly} where $m_{\text{dly}} \in \mathbb{Z}$. Consider the sequence y defined by $y_n = x_{n-m_{\text{dly}}}$, that is the signal x *delayed by* m_{dly}. In z-transform domain this can be expressed as

$$Y(z) = z^{-m_{\text{dly}}} X(z).$$

This can be easily seen by plugging $x_{n-m_{\text{dly}}}$ into the definition of the z-transform, i.e.,

$$\begin{aligned} Y(z) &= \sum_n y_n z^{-n} \\ &= \sum_n x_{n-m_{\text{dly}}} z^{-n} \\ &= \sum_{n'} x_{n'} z^{-n'-m_{\text{dly}}} \\ &= z^{-m_{\text{dly}}} \sum_n x_n z^{-n} \\ &= z^{-m_{\text{dly}}} X(z). \end{aligned}$$

1.5 Wavelets

Wavelets (little waves) are functions that are concentrated in time as well as in frequency around a certain point. For practical applications we choose wavelets which correspond to a so called *multiresolution analysis* [Dau92] due to the reversibility and the efficient computation of the appropriate transform. Wavelets fulfil certain self similarity conditions. When talking about wavelets, we mostly mean a pair of functions: the scaling function ϕ and the wavelet function ψ [Swe96], [Thi01]. Several extensions to this basic scheme exist, but for the introduction we will concentrate on this case. The self similarity (*refinement condition*) of the scaling function ϕ is bounded to a filter h and is defined by

$$\phi(t) = \sqrt{2} \sum_{k \in \mathbb{Z}} h_k \phi(2t - k) \qquad t, h_k \in \mathbb{R} \tag{1.5}$$

which means that ϕ remains unchanged if you filter it with h, downsample it by a factor of two, and amplify the values by $\sqrt{2}$, successively (see Figure 1.2). One could also say, that ϕ is the eigenfunction with eigenvalue 1 of the linear operator that is described by the refinement. Since eigenfunctions are unique only if the amplitude is given, the scaling function is additionally normalized to

$$\sum_{k \in \mathbb{Z}} \phi(k) = \sqrt{2}$$

to make it unique.
The wavelet function ψ is built on ϕ with help of a filter g (Figure 1.3):

$$\psi(t) = \sqrt{2} \sum_{k \in \mathbb{Z}} g_k \phi(2t - k) \qquad g_k \in \mathbb{R}. \tag{1.6}$$

ϕ and ψ are uniquely determined by the filters h and g.
Variants of these functions are defined, which are translated by an integer, compressed by a power of two, and usually amplified by a power of $\sqrt{2}$:

$$\psi_{j,l}(t) = 2^{j/2} \psi(2^j t - l) \tag{1.7}$$

$$\phi_{j,l}(t) = 2^{j/2} \phi(2^j t - l) \tag{1.8}$$

with $j, l \in \mathbb{Z}, t \in \mathbb{R}$

- j denotes the scale – the bigger j the higher the frequency and the thinner the wavelet peak
- l denotes the translation – the bigger l the more shift to the right, and the bigger j the smaller the steps

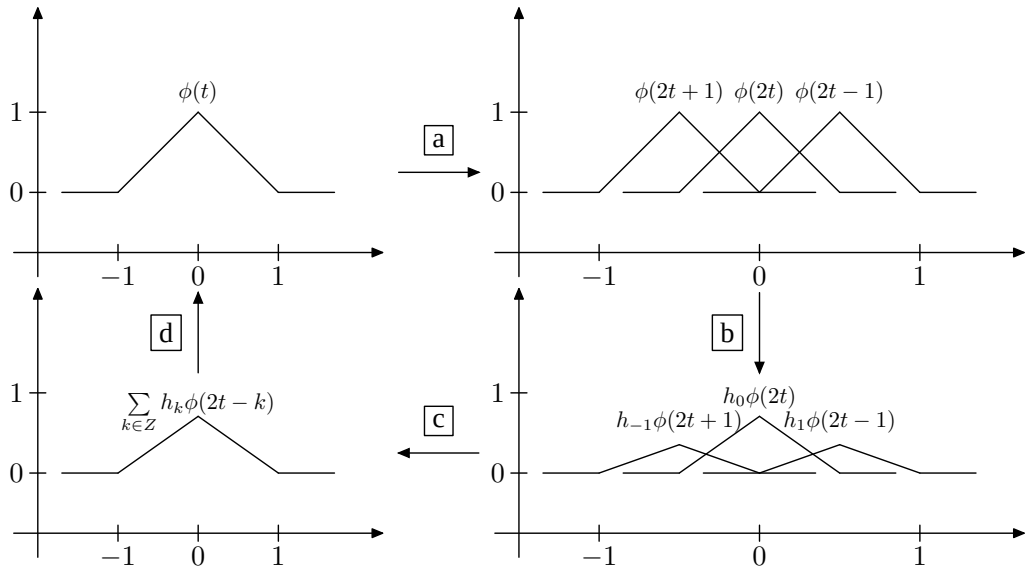


Figure 1.2: Refinement condition of the scaling function – In step [a] the scaling function is duplicated, translated and scaled in abscissa. In step [b] the translated and scaled duplicates are amplified.
The used filter coefficients $h_{-1} = \frac{1}{\sqrt{2}} \cdot \frac{1}{2}, h_0 = \frac{1}{\sqrt{2}} \cdot 1, h_1 = \frac{1}{\sqrt{2}} \cdot \frac{1}{2}$ correspond to the synthesis scaling function filter of the CDF(2,2) wavelet that will be frequently used in this document (see Section 1.7). In step [c] the translated, scaled, and amplified duplicates are added. (step [a] to [b] form the filtering). Step [d] scales the function with $\sqrt{2}$.
A scaling function is characterized by being invariant under the sequence of the steps [a], [b], [c], and [d].

1.6 Discrete Wavelet Transforms

The goal is to represent signals as linear combinations of wavelet functions at several scales and of scaling functions of the widest required scale:

$$\xi(t) \quad = \quad \sum_{l\in\mathbb{Z}} c_{1-J,l}(t)\phi_{1-J,l} + \sum_{j=1-J}^{0} \sum_{l\in\mathbb{Z}} d_{j,l}\psi_{j,l}(t), \quad t \in \mathbb{R}.$$

The choice of wavelet functions as primitives promises to be good, because natural signals x like audio streams or images consist of same structures at different scales and different positions. The coefficients $c_{1-J,l}$ and $d_{j,l}$ for $-J < j \leq 0$ describe the transformed signal we want to feed into a compression routine. J corresponds to the number of different scales we can represent, which is equal to the number of transform levels that will be considered later in detail. The bigger J the more coarse structures can be described. A possible set of scaling and wavelet functions is shown in Figure 1.4.
We usually have to handle with discrete functions, known as sampled audio or image data. For simplicity we consider only one dimensional data. In the case of two dimensional image data we process rows and columns separately. This is explained in detail in Chapter 5.
Its values $\ldots, x_{-1}, x_0, x_1, x_2, \ldots$ represent the amplitudes of pulses. If we want to integrate such signals into the wavelet theory we have to read the x_n as amplitudes of small scaling functions

$$\xi(t) = \quad \sum_{n\in\mathbb{Z}} x_n\phi_{1,n}(t).$$

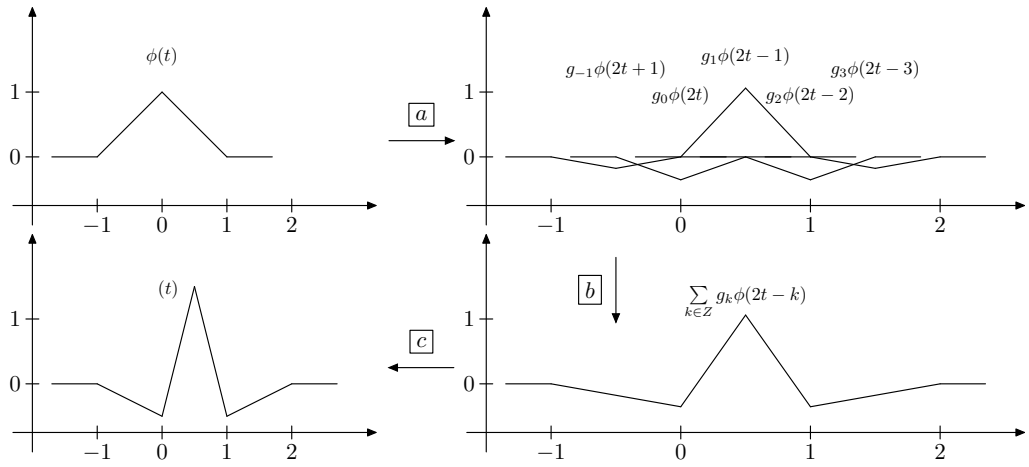


Figure 1.3: Building the wavelet function from scaling functions – In step a the scaling function is duplicated, translated, amplified, and scaled in abscissa. The filter g is borrowed from the CDF(2,2) synthesis wavelet, again, and is determined by the coefficients $g_{-1} = \sqrt{2} \cdot (-\frac{1}{8}), g_0 = \sqrt{2} \cdot (-\frac{1}{4}), g_1 = \sqrt{2} \cdot \frac{3}{4}, g_2 = \sqrt{2} \cdot (-\frac{1}{4}), g_3 = \sqrt{2} \cdot (-\frac{1}{8})$ In step b the functions of step a are added. Step c in this figure represents scaling in ordinate direction.

Therefore we set

$$c_{1,l} := x_l, \text{ for } l \in \mathbb{Z}$$

for discrete signals x. At level J, we use the representation

$$\xi(t) = \sum_{l\in\mathbb{Z}} c_{1-J,l}\phi_{1-J,l}(t) + \sum_{j=1-J}^{0}\sum_{l\in\mathbb{Z}} d_{j,l}\psi_{j,l}(t). \tag{1.9}$$

Given the wavelet decomposition of a signal at level J, we obtain that of level $J-1$ by replacing all functions of level J by their refinements defined in Equation (1.7) and Equation (1.8). Iterating until reaching level 0 results in the searched signal representation. Now let us illustrate this remark and let us start with Equation (1.9).

$$\xi(t) = \sum_{l\in\mathbb{Z}} c_{1-J,l}\phi_{1-J,l}(t) + \sum_{j=1-J}^{0}\sum_{l\in\mathbb{Z}} d_{j,l}\psi_{j,l}(t)$$

According to the definition of $\phi_{j,l}, \psi_{j,l}$ Equation (1.8) we obtain

$$\xi(t) = 2^{\frac{1-J}{2}}\left(\sum_{l\in\mathbb{Z}} c_{1-J,l}\phi(2^{1-J}t-l) + \sum_{l\in\mathbb{Z}} d_{1-J,l}\psi(2^{1-J}t-l)\right) + \sum_{j=2-J}^{0}\sum_{l\in\mathbb{Z}} d_{j,l}\psi_{j,l}(t).$$

Applying refinement conditions Equation (1.5) and Equation (1.6) results in

$$\begin{aligned}\xi(t) = 2^{\frac{2-J}{2}}\Bigg(&\sum_{l\in\mathbb{Z}} c_{1-J,l}\sum_{k\in\mathbb{Z}} h_k\phi(2\cdot(2^{1-J}t-l)-k) \quad + \\ &\sum_{l\in\mathbb{Z}} d_{1-J,l}\sum_{k\in\mathbb{Z}} g_k\phi(2\cdot(2^{1-J}t-l)-k)\Bigg) + \sum_{j=2-J}^{0}\sum_{l\in\mathbb{Z}} d_{j,l}\psi_{j,l}(t)\end{aligned}$$

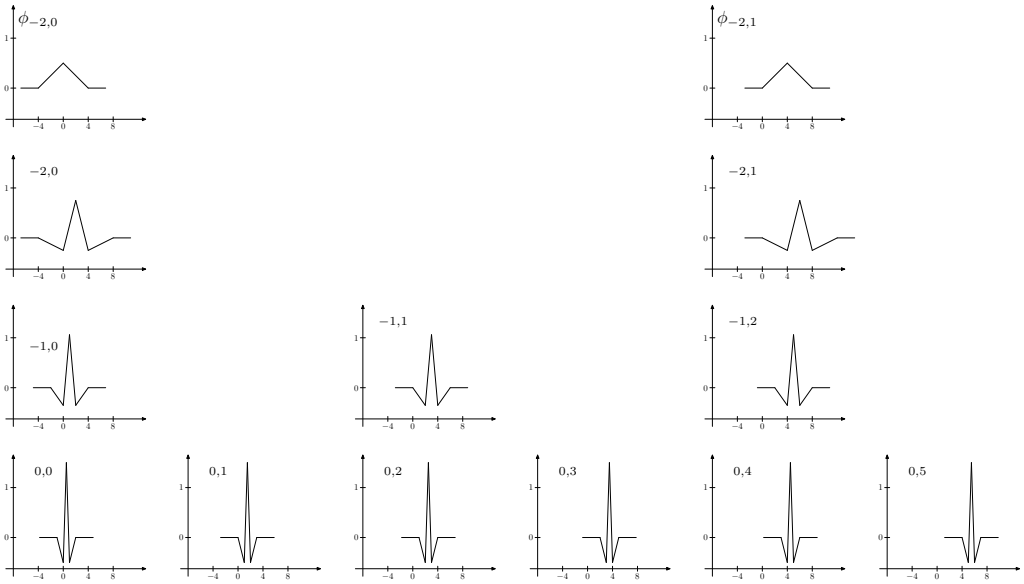


Figure 1.4: A basis consisting of scaling and wavelet functions of the CDF(2,2) wavelet – This basis covers three levels of wavelet functions. Only a finite clip of translates is displayed.

Now, substituting $L := l$, one obtain

$$\begin{aligned}\xi(t) &= 2^{\frac{2-J}{2}} \sum_{k\in\mathbb{Z}} \left(\sum_{L\in\mathbb{Z}} \left(c_{1-J,L} h_k + d_{1-J,L} g_k\right) \phi(2\cdot(2^{1-J}t - L) - k) \right) \\ &\quad + \sum_{j=2-J}^{0} \sum_{l\in\mathbb{Z}} d_{j,l}\psi_{j,l}(t)\end{aligned}$$

and substituting back $l := 2L + k$ results in

$$\begin{aligned}\xi(t) &= 2^{\frac{2-J}{2}} \sum_{l\in\mathbb{Z}} \underbrace{\left(\sum_{L\in\mathbb{Z}} c_{1-J,L} h_{l-2L} + \sum_{L\in\mathbb{Z}} d_{1-J,L} g_{l-2L} \right)}_{c_{2-J,l} :=} \phi(2^{2-J}t - l) \\ &\quad + \sum_{j=2-J}^{0} \sum_{l\in\mathbb{Z}} d_{j,l}\psi_{j,l}(t) \\ &= \sum_{l\in\mathbb{Z}} c_{2-J,l}\phi_{2-J,l}(t) + \sum_{j=2-J}^{0} \sum_{l\in\mathbb{Z}} d_{j,l}\psi_{j,l}(t).\end{aligned}$$

Indeed, this is the signal representation at level $J-1$ of the wavelet decomposition. We see that the new coefficients $c_{2-J,l}$ are derived from $c_{1-J,l}$ and $d_{1-J,l}$ by a kind of filtering. The difference to traditional filtering is, that for even l, $c_{j,l}$ depends only on h_k and g_k with even k, and for odd l, $c_{j,l}$ depends only on h_k and g_k with odd k. This is the reason why we will split both g and h in its even and odd indexed coefficients for most of our investigations. For more details we refer to Section 1.8, where the *Lifting Scheme* is discussed.

It is easy to see that the conversion from wavelet coefficients to signal values is possible without knowing ϕ or ψ, the only information needed, are the filters which belong to them. Under certain conditions, the same is true for the reverse conversion. It allows us to limit our view to the filters g and h and hide the functions ϕ and ψ. Thus the computation of this change in representation can be made with the use of filters.

In the following we have to distinguish between the conversion from the original signal to the wavelet coefficients and from the wavelet coefficients back to the signal or an approximated version of it. The first conversion is usually denoted by *wavelet analysis* or *wavelet decomposition*, and the second by *wavelet synthesis* or *wavelet reconstruction*. We will use these notions throughout the document.
Since the filters and the scaling and wavelet function can differ for wavelet analysis and synthesis (for instance in the case of biorthogonal bases) we will denote the analysis scaling and wavelet functions by $\tilde{\phi}$ and $\tilde{\psi}$, respectively. For the synthesis scaling and wavelet functions we use the symbols ϕ and ψ, respectively. The corresponding filters are denoted accordingly by $\tilde{g}$, $\tilde{h}$ and g, h.

Now, one level of discrete wavelet transform can be expressed

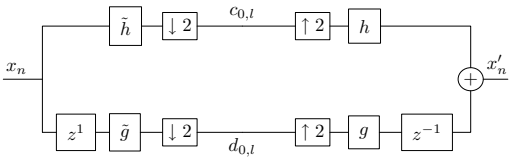


Figure 1.5: one level of wavelet transform expressed using filters

as depicted in Figure 1.5. Usually the filter g and $\tilde{h}$ are low pass filter. The filter h and $\tilde{g}$ are high pass filter. Thus, we can interpret the coefficients $c_{0,l}$ as coarse version of the signal x at half resolution. The coefficients $d_{0,l}$ are the differences or details that are necessary to reconstruct the original signal x from the coarse version. The delay $\boxed{z^{-1}}$ is necessary, to discard even and odd indices after filtering. We can recursively apply this filtering scheme to the coefficients $c_{j,l}$ if we want to perform more than one level of transform. In Figure 1.6 three levels of transform are applied. This scheme is known as *Mallat's algorithm*

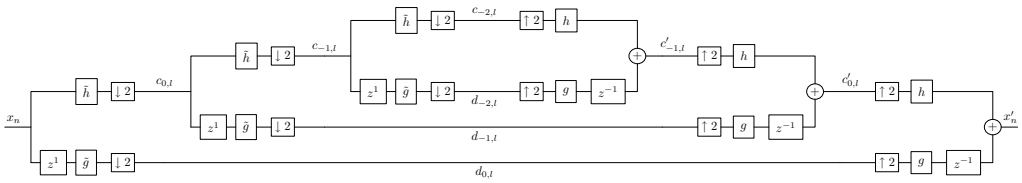


Figure 1.6: tree levels of wavelet transform

[Mal89]. For implementation issues we have to note, that this filter bank approach is not that efficient to compute the coefficients $c_{j,l}$ and $d_{j,l}$. As you can see, every second computed coefficient is dropped after the filtering due to the downsampling. In Section 1.8 we introduce the *Lifting Scheme* presented by Wim Sweldens [Swe96], which provides an efficient implementation of the wavelet decomposition and reconstruction.

1.7 Cohen-Daubechies-Feauveau CDF(2,2) Wavelet

In this thesis we make intensive use of the so called CDF(2,2) wavelet presented by Cohen, Daubechies, and Feauveau [CDF92]. It is also known as the biorthogonal (5,3) wavelet because of the filter length of 5 and 3 for the low and high pass filters, respectively. We have already referred to this specific wavelet while discussing the self similarity properties of scaling functions and the coherence of scaling and wavelet functions.

The CDF(2,2) is a biorthogonal wavelet. In contrast to orthogonal wavelets (except the Haar wavelet [Haa10]) the filters as well as the scaling and wavelet functions for decomposition and reconstructions are

symmetric. A symmetric filter f always has odd filter length and it holds that

$$f_{a+k} = f_{b-k}$$

for all $0 \leq k < \frac{b-a}{2}$ and a and b are the smallest and greatest index l, respectively, where f_l is different from zero. Symmetry is a very important property if we consider image compression, because in the absence of symmetry artefacts are introduced around edges.

Let us first introduce the coefficients of the filters $\tilde{g}, \tilde{h}, g, h$. They are listed in Table 1.1.

Table 1.1: filter coefficients of the CDF(2,2) wavelet, the delay operators $\boxed{z^1}$ and $\boxed{z^{-1}}$ are implicit enclosed in the filters

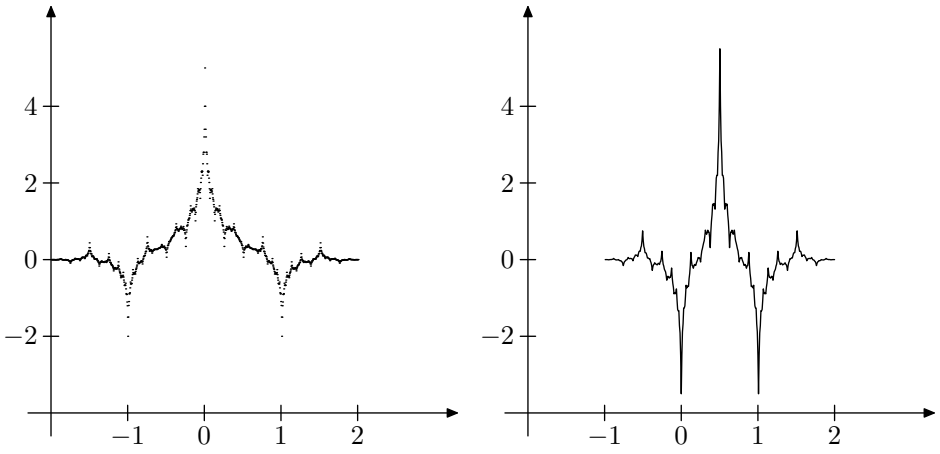
i	-2	-1	0	1	2	3
$\tilde{h}$	$\sqrt{2}\cdot(-\frac{1}{8})$	$\sqrt{2}\cdot\frac{1}{4}$	$\sqrt{2}\cdot\frac{3}{4}$	$\sqrt{2}\cdot\frac{1}{4}$	$\sqrt{2}\cdot(-\frac{1}{8})$	
$\tilde{g}$			$\frac{1}{\sqrt{2}}\cdot(-\frac{1}{2})$	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}\cdot(-\frac{1}{2})$	
h		$\frac{1}{\sqrt{2}}\cdot\frac{1}{2}$	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}\cdot\frac{1}{2}$		
g		$\sqrt{2}\cdot(-\frac{1}{8})$	$\sqrt{2}\cdot(-\frac{1}{4})$	$\sqrt{2}\cdot\frac{3}{4}$	$\sqrt{2}\cdot(-\frac{1}{4})$	$\sqrt{2}\cdot(-\frac{1}{8})$

Note that in an implementation we do not deal with the normalization factors $\sqrt{2}$ and $\frac{1}{\sqrt{2}}$ during the filtering. These normalization factors are applied after all wavelet transforms have taken place. If the input signal is an image we will apply the CDF(2,2) transform as tensor product to the rows and columns independently. Here it will be not necessary to deal with $\sqrt{2}$ or $\frac{1}{\sqrt{2}}$ directly, because their products leads to 2^k for some $k \in \mathbb{Z}$ (see Section 2.3).
Thus, we only need the operations

- addition,
- multiplication with constants, and
- shifts,

which is one of the important facts to use this wavelet in a hardware implementation of wavelet based image codecs. Furthermore, the *Lifting Scheme* applied to CDF(2,2) results in an integer-to-integer mapping (*Lifting* is presented in Section 1.8). The operations above can be implemented using integer arithmetic. We avoid floating point units in our FPGA architectures, which are very expensive in terms of chip area.

For the sake of completeness we have computed the graphs of the functions $\tilde{\psi}$, $\tilde{\phi}$, ϕ, and ψ. Since no closed expression is available for the limit functions $\tilde{\phi}$ and $\tilde{\psi}$, we used the so called *cascade algorithm* [Dau92] to calculate these graphs. To obtain the graph of $\tilde{\phi}$ we start with the discrete impulse δ as input signal. Then we convolve δ with the analysis low pass filter $\tilde{h}$. Let the result be denoted by $\tilde{\phi}^{(1)}$. After that we upsample the filter $\tilde{h}$ and convolve the upsampled filter with $\tilde{\phi}^{(1)}$ resulting in $\tilde{\phi}^{(2)}$. This procedure can be repeated until a sufficient accuracy is achieved. In the literature this is usually done eight or nine times. The resulting graph is shown in Figure 1.7(a). In Appendix A.3 you find the corresponding code in order to produce these graphs using Matlab or METAPOST. The graph for $\tilde{\psi}$ is derived in an analogous manner. The last convolution is done using the appropriate upsampled high pass filter $\tilde{g}$ instead of $\tilde{h}$. For an illustration see Figure 1.7(b). The synthesis scaling and wavelet function ϕ and ψ can be derived analogously. But these two functions were constructed to be linear splines, so we could even depict them directly.



(a) Decomposition / analysis scaling function $\tilde{\phi}$, filter coefficients $\tilde{g}_0 = \frac{1}{\sqrt{2}} \cdot (-\frac{1}{2}), \tilde{g}_1 = \frac{1}{\sqrt{2}} \cdot 1, \tilde{g}_2 = \frac{1}{\sqrt{2}} \cdot (-\frac{1}{2})$

(b) Decomposition / analysis wavelet function $\tilde{\psi}$, filter coefficients $\tilde{h}_{-2} = \sqrt{2} \cdot (-\frac{1}{8}), \tilde{h}_{-1} = \sqrt{2} \cdot \frac{1}{4}, \tilde{h}_0 = \sqrt{2} \cdot \frac{3}{4}, \tilde{h}_1 = \sqrt{2} \cdot \frac{1}{4}, \tilde{h}_2 = \sqrt{2} \cdot (-\frac{1}{8})$

Figure 1.7: the analysis scaling $\tilde{\phi}$ and wavelet function $\tilde{\psi}$ of CDF(2,2)

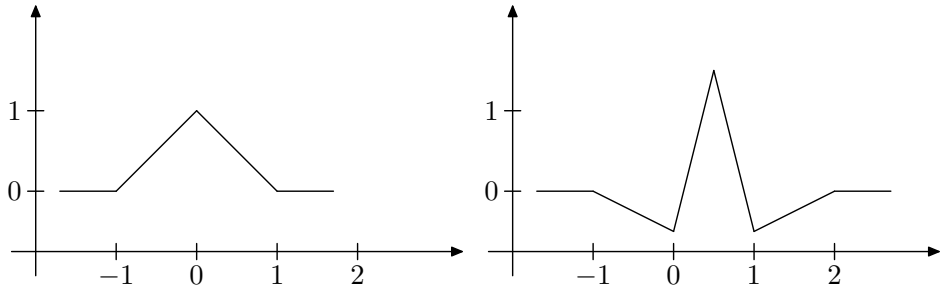
Remark the different support for each of the four functions $\tilde{\psi}$, $\tilde{\phi}$, ϕ, and ψ. The corresponding filters are just the same as listed in Table 1.1. The delay operators $\boxed{z^1}$ or $\boxed{z^{-1}}$ are implicit enclosed in the filters.

For image compression applications it is interesting to discuss the relation between the regularity of the synthesis wavelet and the number of so called *vanishing moments* of the analysis wavelet. A biorthogonal wavelet has m *vanishing moments* if and only if its dual scaling function generates polynomials up to degree m. In other words, vanishing moments tend to reduce the number of significant wavelet coefficients and thus, one should select a wavelet with many of them for the analysis (the notation of significance is introduced in Chapter 4). On the other hand, regular or smooth synthesis wavelets give good approximations, if not all coefficients are used for reconstruction, as it is the case for lossy compression.

To increase the number of vanishing moments of the decomposition wavelet one has to enlarge the filter length of the corresponding analysis low and high pass filters. That is, you have a trade off between filter length and number of vanishing moments of the decomposition wavelet. In terms of image compression you can improve the compression performance at the expense of increasing computational power to calculate the filter operations.

Another way to increase the number of vanishing moments is to use smoother reconstruction wavelets. This corresponds to better compression performance at the expense of enlarged synthesis filters, too. Furthermore, to achieve regular analysis wavelets the filter lengths have to be increased to a greater extend compared to not that smooth reconstruction wavelets.

Both vanishing moments of the decomposition wavelet and regularity of the reconstruction wavelet are important in improving both subjective and objective compression measures. In many cases, increasing the reconstruction regularity, even at great expense in decomposition vanishing moments, improves results. That is the case for the CDF(2,2) wavelet as you can observe immediately comparing Figure 1.7 and Figure 1.8.



(a) Reconstruction (synthesis) scaling function ϕ, filter coefficients $g_{-1} = \frac{1}{\sqrt{2}} \cdot \frac{1}{2}, g_0 = \frac{1}{\sqrt{2}} \cdot 1, g_1 = \frac{1}{\sqrt{2}} \cdot \frac{1}{2}$

(b) Reconstruction (synthesis) wavelet function ψ, filter coefficients $h_{-1} = \sqrt{2} \cdot (-\frac{1}{8}), h_0 = \sqrt{2} \cdot (-\frac{1}{4}), h_1 = \sqrt{2} \cdot \frac{3}{4}, h_2 = \sqrt{2} \cdot (-\frac{1}{4}), h_3 = \sqrt{2} \cdot (-\frac{1}{8})$

Figure 1.8: the synthesis scaling ϕ and wavelet function ψ of CDF(2,2)

1.8 Lifting Scheme

An alternative computation method of the discrete wavelet transform is the so called *Lifting Scheme* originally presented by Wim Sweldens [Swe96]. Usually it is explained while discussing the Haar wavelet transform. In order to be consistent we base our introduction to Lifting on the CDF(2,2) wavelet, which is taken as explanation example too. The Lifting Scheme is composed of three steps, namely:

- Split (also called Lazy Wavelet Transform),
- Predict,
- and Update.

The first step is splitting the input signal x into even and odd indexed samples.
Then we try to predict the odd samples based on the evens. If the original signal has local correlation, then the prediction should be of high accuracy. The odd samples are replaced by the old ones minus the prediction. Now we can interpret them as the detail coefficients, to which we are now familiar with. On the other hand we want to think of the even samples as the coarser version of the input sequence at half the resolution. But we have to ensure that the average of the signal is preserved, that is

$$\sum_{k\in\mathbb{Z}} c_{j,k} = \frac{1}{2} \sum_{k\in\mathbb{Z}} c_{j+1,k}$$

for all $-J < j \leq 0$
This task is performed in the so called update step, where the detail coefficients are used to update the evens in order to preserve the average. In the left side of Figure 1.9 these three steps are depicted. We have used the symbols $\boxed{P}$ and $\boxed{U}$ for the Predict and Update operator, respectively.
The inverse procedure is really simple. Just exchange the sign for the predict and update step and apply all operations in reversed order as shown on the right side of Figure 1.9.
Note, that with *z-transform* notation we could express the split and merge step using downsampling, delay, and upsampling, respectively. This is illustrated in Figure 1.10
Let us now look at the predict and update steps for the CDF(2,2) wavelet [SS96]. Here the predictor is chosen to be linear, that is, if the input signal is a polynomial of degree one, the prediction is perfect. In that case all detail coefficients will be zero. Therefore we have

$$d_{j,k} = c_{j+1,2k+1} - \frac{1}{2}(c_{j+1,2k} + c_{j+1,2k+2}) \tag{1.10}$$

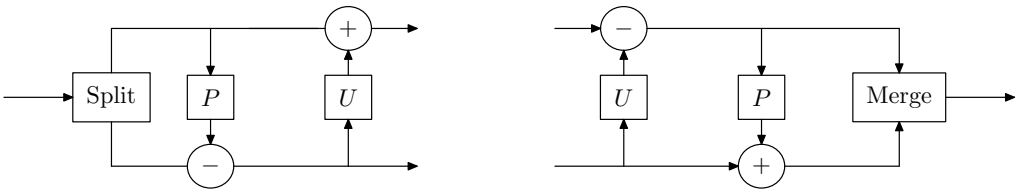


Figure 1.9: The lifting scheme

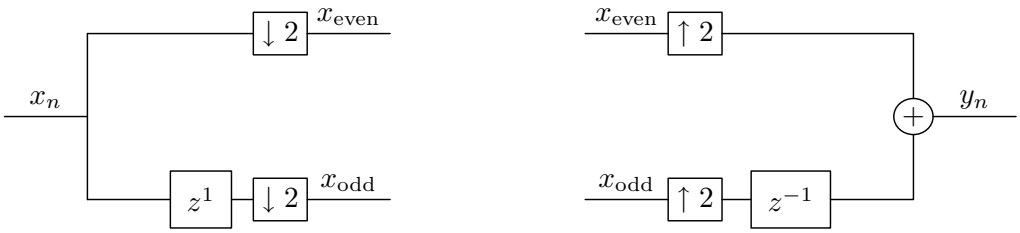


Figure 1.10: split and merge step described using down/upsampling and delay

for the prediction step and

$$c_{j,k} = c_{j+1,2k} + \frac{1}{4}(d_{j,k} + d_{j,k-1}) \tag{1.11}$$

for the update step.
Indeed, this is equivalent to the filter given in Table 1.1 (up to the normalization factors), if we insert Equation (1.10) in Equation (1.11)

$$\begin{aligned} c_{j,k} &= c_{j+1,2k} + \frac{1}{4}(d_{j,k} + d_{j,k-1}) \\ &= c_{j+1,2k} + \frac{c_{j+1,2k+1} - \frac{c_{j+1,2k}+c_{j+1,2k+2}}{2} + c_{j+1,2k-1} - \frac{c_{j+1,2k-2}+c_{j+1,2k}}{2}}{4} \\ &= c_{j+1,2k} + \frac{c_{j+1,2k+1}}{4} - \frac{c_{j+1,2k} + c_{j+1,2k+2}}{8} + \frac{c_{j+1,2k-1}}{4} - \frac{c_{j+1,2k-2} + c_{j+1,2k}}{8} \\ &= -\frac{1}{8}c_{j+1,2k-2} + \frac{1}{4}c_{j+1,2k-1} + \frac{3}{4}c_{j+1,2k} + \frac{1}{4}c_{j+1,2k+1} - \frac{1}{8}c_{j+1,2k+2} \\ &= \sum_n g_n c_{j+1,2k-n}. \end{aligned} \tag{1.12}$$

What are the advantages of this method ?

1. The most important fact is that we do not throw away already computed coefficients as in the filter bank approach.

2. It is also remarkable, that the wavelet transform can now be computed in place. This means, that given a finite length signal with n samples we need exactly n memory cells, each of them capable to store one sample, to compute the transform.

3. Furthermore we reduce the number of operations in order to compute the coefficients of the next coarser or finer scale, respectively. For the CDF(2,2)-wavelet we save three operations using the Lifting Scheme in comparison with the traditional filter bank approach.

1.8.1 Integer-to-Integer Mapping

Obviously, the application of the filter bank approach or the Lifting Scheme leads to coefficients, which are not integers in general. In the field of hardware image compression it would be convenient, that coefficients and the pixel of the reconstructed image are integers too. Chao et.al.[CFH96] and Cohen et.al.[CDSY97] have introduced techniques for doing so. The basic idea is to modify the computation of the Predict and Update step in the following way

$$\begin{aligned} d'_{j,k} &= c'_{j+1,2k+1} - \lfloor P \rfloor \\ c'_{j,k} &= c'_{j+1,2k} + \lfloor U \rfloor , \end{aligned}$$

where $c'_{1,k} = c_{1,k} = x_k$ for all $k \in \mathbb{Z}$. Here the symbols P and U represent any reasonable predictor or update operator, respectively.

Remark It is easy to verify, that perfect reconstruction in case of lossless compression is guaranteed, since that same value of the modified predictor and update operator is added and subtracted. However, since we lose the linearity of the transform due to the rounding, the influence of this modification in terms of image compression efficiency is hard to estimate. Fortunately, our practical experiences do not show any remarkable differences between the Lifting Scheme and the modified version.

For the special case of the CDF(2,2) wavelet we therefore use the prediction and update steps as follows:

$$d'_{j,k} = c'_{j+1,2k+1} - \left\lfloor \frac{1}{2}(c'_{j+1,2k} + c'_{j+1,2k+2}) \right\rfloor \tag{1.13}$$

$$c'_{j,k} = c'_{j+1,2k} + \left\lfloor \frac{1}{4}(d'_{j,k} + d'_{j,k-1}) \right\rfloor . \tag{1.14}$$

As a consequence the coefficients of all scales $-J < j \leq 0$ can be stored as integers and for all operations integer arithmetic is sufficient. Note, that the coarser the scale the more bits are necessary to store the corresponding coefficients. To overcome the growing bitwidth at coarser scales modular arithmetic can be used in the case of lossless compression.

1.8.2 Lifting Scheme and Modular Arithmetic

Chao et.al. suggest to use modular arithmetic in combination with the Lifting Scheme [CFH96]. We use the symbols $\oplus$ and $\ominus$ for the modular addition and subtraction, which are defined as

$$\begin{aligned} a \oplus b &:= a + b \bmod m \quad \text{and} \\ a \ominus b &:= a - b \bmod m \end{aligned}$$

for reasonable $m \in \mathbb{N}$, respectively. Equation (1.13) and Equation (1.14) now look like

$$\begin{aligned} d'_{j,k} &= c'_{j+1,2k+1} \ominus \left\lfloor \frac{1}{2}(c'_{j+1,2k} \oplus c'_{j+1,2k+2}) \right\rfloor \\ c'_{j,k} &= c'_{j+1,2k} \oplus \left\lfloor \frac{1}{4}(d'_{j,k} \oplus d'_{j,k-1}) \right\rfloor . \end{aligned}$$

As a consequence, all coefficients at all scales as well as the reconstructed signal have the same bitwidth as the original sequence, if we initialize $m = 2^{\mathrm{dpth}}$, where dpth is the given bitwidth of the original samples. Additionally, the computational units for addition and subtraction have to be provided for m bit operands only. With respect to the hardware implementation we save memory resources and logic in the arithmetic logic units.
Remark, that this modification is only feasible in case of lossless compression. The approach of Chao et.al. is limited to lossless compression, since in fact transform coefficients of large magnitude become small due to modular arithmetic. This is counterproductive for quantization purposes in wavelet based compression techniques, where coefficients with large magnitude are considered as important and small coefficients tend to be neglected (see Chapter 4 for a detailed discussion of wavelet based image codecs).

Chapter 2

Wavelet transforms on images

Until now we have discussed one dimensional wavelet transforms. Images are obviously two dimensional data. To transform images we can use two dimensional wavelets or apply the one dimensional transform to the rows and columns of the image successively as separable two dimensional transform. In most of the applications, where wavelets are used for image processing and compression, the latter choice is taken, because of the low computational complexity of separable transforms.

Before explaining wavelet transforms on images in more detail, we have to introduce some notations. We consider an $N \times N$ image as two dimensional pixel array $\mathcal{I}$ with N rows and N columns. We assume without loss of generality that the equation $N = 2^r$ holds for some positive integer r.

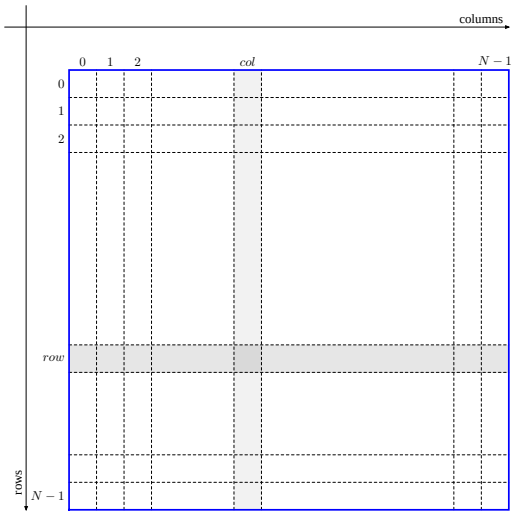


Figure 2.1: images interpretation as two dimensional array $\mathcal{I}$, where the rows are enumerated from top to bottom and the columns from left to right, starting at index zero

In Figure 2.1 we illustrate, how the pixels of the images are arranged in the corresponding array $\mathcal{I}$. The rows are enumerated from top to bottom and the columns from left to right. The index starts with zero and therefore the largest index is $N - 1$. The image pixels themself at row i and column j will be denoted by $\mathcal{I}[i, j]$ or $\mathcal{I}_{i,j}$. The wavelet transformed image will be denoted by $\hat{\mathcal{I}}$ and the coefficients are addressed with $\hat{\mathcal{I}}[k, l]$ or $\hat{\mathcal{I}}_{k,l}$. For the reconstructed image we will use $\tilde{\mathcal{I}}$ and address the corresponding reconstructed pixels as $\tilde{\mathcal{I}}[n, m]$ or $\tilde{\mathcal{I}}_{n,m}$.

The pixels and coefficients themselves are stored as signed integers in two's complementary encoding.

Therefore the range is given as

$$\mathcal{I}[\text{row}, \text{col}] \in \left[-2^{\text{dpth}-1}, 2^{\text{dpth}-1} - 1\right],$$

where $0 \leq \text{row}, \text{col} < N$, assuming a dpth-bit greyscale resolution. Thus, we can distinguish between 2^{dpth} different values of brightness. For an illustration see Figure 2.2. The smallest value $-2^{\text{dpth}-1}$ and

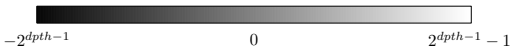


Figure 2.2: greyscales and the corresponding pixel values for dpth-bit resolution

the largest value $2^{\text{dpth}-1} - 1$ correspond to black and white, respectively. As a consequence pixels with magnitude around zero appear as grey color.

In color images each pixel is represented by several color components. Typically there are three of them per pixel. In the RGB color space, e.g., there is one component for red, green, and blue, respectively. Other choices are the YUV color space (luminance and chrominance) and the CMYK color space (cyan, magenta, yellow, black). Note, that there exist YUV based image and video formats, where the size N of the different components is different (e.g. 4:2:2 and 4:1:1). In the case of the 4:1:1 format for instance, we obtain three pixel arrays of size N, $\frac{N}{4}$, and $\frac{N}{4}$.

Throughout this thesis we will treat each color component of color images as separate greyscale image.

Now, let us come back to wavelet transforms on images. As already mentioned the one dimensional transform will be applied to rows and columns successively. Consider a row $r = (r_0, \ldots, r_{N-1})$ of an image $\mathcal{I}$. This row has finite length in contrast to the signals or sequences we have considered until now. In order to convolve such a row r with a filter f we have to extend it to infinity in both directions. Let r' be the extended row defined by

$$r' = (\ldots, r'_0, \ldots, r'_{N-1}, \ldots),$$

where $r'_k = r_k$ for all $0 \leq k < N - 1$. But, how do we have to set the values of r' at positions k with $k \notin [0, N-1]$? In some sense we are free to choose these remaining samples. In the next section we will explain, why reflection at the image boundary should be used in horizontal and in vertical direction.

2.1 Reflection at Image Boundary

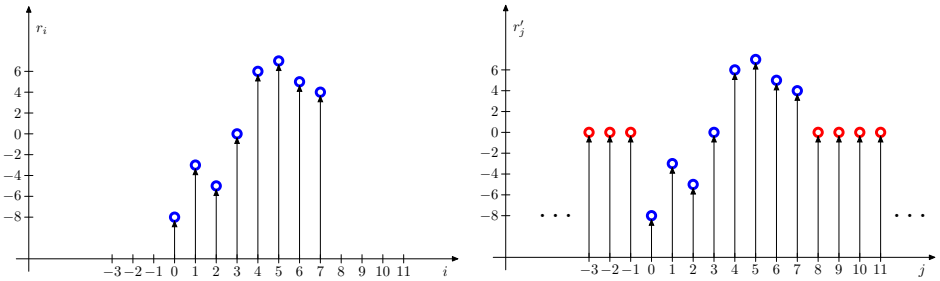
There are several choices to choose the values of r'_k from outside the interval $[0, N-1]$. The most popular one's are

- padding with zeros,
- periodic extension, or
- symmetric extension.

The simplest choice is to set all remaining r'_k to zero. For an illustration see Figure 2.3. In Figure 2.3(a) a sequence r of length $N = 8$ is shown, where $r = (-\mathbf{8}, -3, -5, 0, 6, 7, 5, 4)$. In Figure 2.3(b) this sequence is padded with zeros in order to obtain the infinite sequence r':

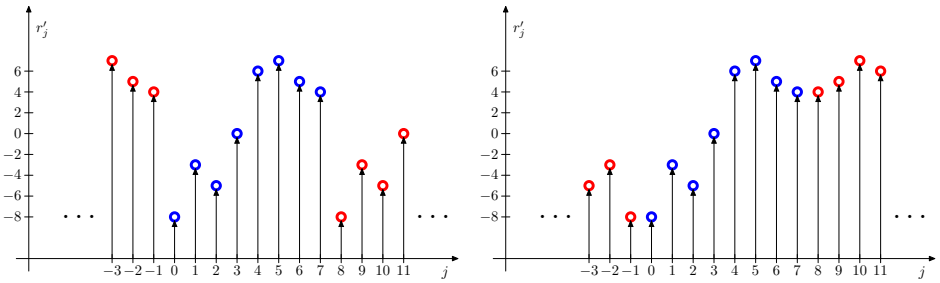
$$r'_k = \begin{cases} r_k & : \quad \text{for all } 0 \leq k < N \\ 0 & : \quad \text{else .} \end{cases}$$

We can observe, that in general there will be discontinuities at the boundary.
The substantial difference between the value of the border coefficients and zero leads to coefficients of large amount in the high frequency subbands. These differences decrease the compression efficiency and introduce artefacts at the boundaries since the reconstructed pixel values depend on the values of the coefficients from outside, too, if lossy compression is considered.



(a) the original sequence r(e.g. image row)

(b) r' obtained from r by zero padding



(c) r' obtained from r by periodic extension

(d) r' obtained from r by symmetric extension

Figure 2.3: different choices for the boundary treatment of finite length sequences, here $r = (-\mathbf{8}, -3, -5, 0, 6, 7, 5, 4)$ of length $N = 8$

The computation of the coefficients $c_{0,0}, c_{0,1}$ and $d_{0,N-1}$ at the first level of a wavelet transform using the CDF(2,2) wavelet depends on pixels from outside. In general, for a symmetric wavelet with corresponding analysis filters $\tilde{h}$ and $\tilde{g}$ the computation of the coefficients

$$\begin{aligned} c_{0,0}, c_{0,1}, \ldots, c_{0,l} \qquad & \text{with} \quad l = \frac{\left|\tilde{h}\right| + 1}{2} \\ d_{0,N-1-l}, d_{0,N-l}, \ldots, d_{0,N-1} \qquad & \text{with} \quad l = \frac{|\tilde{g}| + 1}{2} \end{aligned}$$

depends on pixels from outside.

A different approach known from Fourier techniques is periodic extension of the signal, that is

$$r'_{k \cdot N + l} = r_l$$

where $0 \leq l < N$ and $k \in \mathbb{Z}$. Here we encounter the same drawbacks as in the case of padding with zeros (cf. Figure 2.3(c)). The introduced differences at the boundary are considerable as well. Another drawback arises in hardware implementations. Here we have to buffer the first samples of r in order to perform the operations at the end of the sequence r. This can result in large buffers.

The most preferred method for the choice of the coefficients from outside of r is based on symmetric extension. Figure 2.3(d) depicts this method applied to our example sequence. More formally, symmetric extension r' is defined by

$$r'_{k \cdot N + l} = \begin{cases} r_{N-1-l} & : \text{ if } k \text{ is an odd value} \\ r_l & : \text{ if } k \text{ is an even value.} \end{cases}$$

where $0 \leq l < N$ and $k \in \mathbb{Z}$. The already mentioned difference between coefficients at the boundary do not appear using this kind of extension. Furthermore due to the locally known coefficients no significant, additional amount of buffer memory is required.

In the following we assume, that the boundary treatment was done using symmetric extension. Therefore we do not distinguish between finite and infinite sequence anymore.

2.2 2D-DWT

Now we are able to discuss the separable two dimensional wavelet transform in detail. Consider again a row r of a given image of size $N \times N$. Recall from Section 1.8 the computation of a specific wavelet transform using the Lifting Scheme. After one level of transform we obtain $\frac{N}{2}$ coefficients $c_{0,l}$ and $\frac{N}{2}$ coefficients $d_{0,k}$ with $0 \leq l, k < \frac{N}{2}$. These are given in interleaved order, that is

$$(c_{0,0}, d_{0,0}, c_{0,1}, d_{0,1}, \ldots, c_{0,N-1}, d_{0,N-1}), \tag{2.1}$$

because of the split in odd and even indexed positions in the Lifting Scheme. Usually the row of (2.1) is rearranged to

$$r^{(0)} \quad = \quad (c_{0,0}, c_{0,1}, \ldots, c_{0,N-1}, d_{0,0}, d_{0,1}, \ldots, d_{0,N-1}),$$

because we will apply the transform to the low frequency coefficients $c_{0,l}$ recursively.
Suppose we have already transformed and rearranged all rows of a given image as described above. If we store the computed coefficients in place, that is in the memory space of the original image, we obtain a new array with a structure as shown in Figure 2.4.

(a) *lena*, $N = 512$, $\mathrm{dpth} = 8$

(b) CDF(2,2) wavelet transform applied to the rows of (a)

(c) low (L, $c_{0,k}$) and high (H, $d_{0,k}$) frequency coefficient blocks

Figure 2.4: one dimensional CDF(2,2) wavelet transform applied to the rows and columns of the benchmark image *lena* with reflection at the boundaries

On the left the well known benchmark image *lena*[1] is shown. To the right of it we have applied the CDF(2,2) wavelet transform to the rows of the image. The corresponding result is interpreted as image again (Figure 2.4(b)) and is composed of a coarse and scaled version of the original and the details, which are necessary to reconstruct the image under consideration. On the right we have illustrated this interpretation as low and high frequency coefficients blocks, denoted by L and H, respectively. Remark that most of the

[1]For the curious: 'lena' or 'lenna' is a digitized Playboy centerfold, from November 1972. (Lenna is the spelling in Playboy, Lena is the Swedish spelling of the name.) Lena Soderberg (ne Sjööblom) was last reported living in her native Sweden, happily married with three kids and a job with the state liquor monopoly. In 1988, she was interviewed by some Swedish computer related publication, and she was pleasantly amused by what had happened to her picture. That was the first she knew of the use of that picture in the computer business. (`http://www.lenna.org`)

high frequency coefficients $d_{0,k}$ are shown in grey color, which corresponds to small values around zero (cf. Figure 2.2).

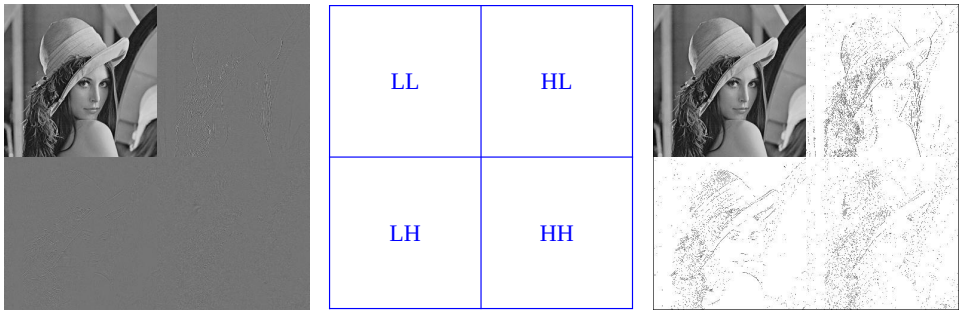
The one dimensional wavelet transform can be applied to the columns of the already horizontal transformed image as well. The result is shown in Figure 2.5 and is decomposed into four quadrants with different interpretations.

LL: The upper left quadrant consists of all coefficients, which were filtered by the analysis low pass filter $\tilde{h}$ along the rows and then filtered along the corresponding columns with the analysis low pass filter $\tilde{h}$ again. This subblock is denoted by LL and represents the approximated version of the original at half the resolution.

HL/LH: The lower left and the upper right blocks were filtered along the rows and columns with $\tilde{h}$ and $\tilde{g}$, alternatively. The LH block contains vertical edges, mostly. In contrast, the HL blocks shows horizontal edges very clearly.

HH: The lower right quadrant was derived analogously to the upper left quadrant but with the use of the analysis high pass filter $\tilde{g}$ which belongs to the given wavelet. We can interpret this block as the area, where we find edges of the original image in diagonal direction.

The two dimensional wavelet transform can be applied to the coarser version at half the resolution, recursively, in order to further decorrelate neighboring pixels of the input image. For an illustration we refer to Figure 2.6. The subbands in the next higher transform levels l will be denoted by $\mathrm{LL}^{(l)}$, $\mathrm{LH}^{(l)}$, $\mathrm{HL}^{(l)}$, and $\mathrm{HH}^{(l)}$, where $\mathrm{LL}^{(1)} = \mathrm{LL}$, $\mathrm{LH}^{(1)} = \mathrm{LH}$, $\mathrm{HL}^{(1)} = \mathrm{HL}$, and $\mathrm{HH}^{(1)} = \mathrm{HH}$, respectively.



(a) CDF(2,2) wavelet applied as tensor product to the rows and columns of the image *lena*

(b) the different frequency blocks

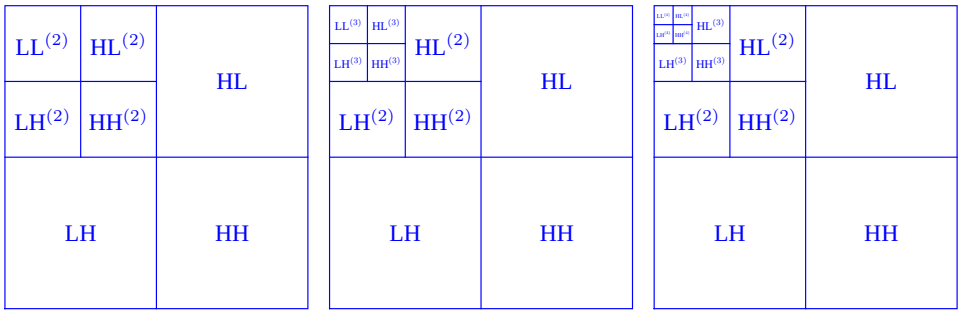
(c) set all coefficients v in LH, HL, HH with $-20 < v < 20$ to white color

Figure 2.5: one dimensional CDF(2,2) wavelet transform applied to the rows of the benchmark image *lena* with reflection at the image boundaries

Since we have restricted the images to be of quadratic size $N = 2^l$ for $l \in N$, we can perform at most $l = \log_2 N$ levels of transform. Thereafter the coefficient in the upper left corner represents the average greyscale value of the whole image and is called *DC coefficient* (DC : direct current). In practice, usually four up to six level of wavelet transform level will be performed.
Due to the local similarities between neighboring pixels, many coefficients in the LH, HL, and HH subbands at different scales will be small. As a consequence only a few samples, especially those of the LL block at the coarsest scale, represents most out of the *images energy*. The *energy* $\mathcal{E}(\mathcal{I})$ of an image $\mathcal{I}$ is defined as

$$\mathcal{E}(\mathcal{I}) \quad = \quad \sum_{r=0}^{N-1} \sum_{c=0}^{N-1} (\mathcal{I}_{r,c})^2 \tag{2.2}$$

This observation is the starting point of wavelet based image compression algorithms, which are explained in Chapter 4.



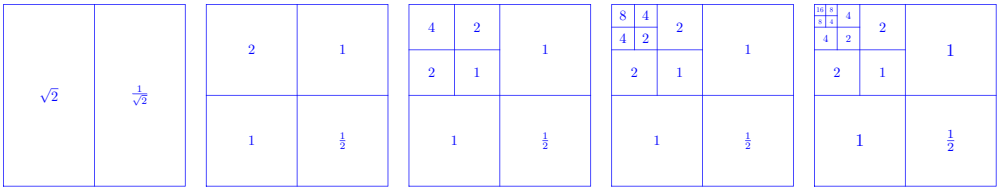
(a) two levels of 2D-DWT (b) three levels of 2D-DWT (c) four levels of 2D-DWT

Figure 2.6: multiresolution scheme after several levels of wavelet transform

In this thesis we will focus on the tree structured decomposition as shown in Figure 2.6, where only the LL blocks are subdivided. This type of image decomposition is also known as *multiresolution scheme* and *multiscale representation*. Other decomposition types are possible and known under the terms of *wavelet packets* [CMQW94].

2.3 Normalization Factors of the CDF(2,2) Wavelet in two Dimensions

In Section 1.7 we have already mentioned the normalization factors of the CDF(2,2)-wavelet for one dimension. To simplify the calculation of the transform we have extracted the factors $\sqrt{2}$ and $\frac{1}{\sqrt{2}}$, which allows us to use efficient integer arithmetic units in hardware implementations.
In order to preserve the average of a one dimensional signal, or the average of the brightness of images, we have to consider the normalization factors after the wavelet transform has taken place.



(a) 1D-DWT (rows) (b) 1D-DWT (columns) (c) 2nd level (d) 3rd level (e) 4th level

Figure 2.7: normalization factors of the CDF(2,2) wavelet in two dimension for each level l, $0 \leq l < 5$

In one dimension we have to scale the low pass coefficients with $\sqrt{2}$ and the high pass coefficients with $\frac{1}{\sqrt{2}}$, which is shown in Figure 2.7(a). Thereafter the same has to be done in vertical direction. Here, the normalization factors become integer powers of two as you can easily verify in Figure 2.7, where each subblock is indexed with the corresponding factor.

To summarize, we can abstract from those normalization factors during the implementation of the CDF(2,2) wavelet transform. Afterwards they will implicit stored and processed. Note, that these implicit factors have

no influence to the growth of bitwidth, which is necessary to store the wavelet coefficients.

Chapter 3

Range of CDF(2,2) Wavelet Coefficients

As we have seen in Section 1.6 wavelet transforms can be expressed using filters.
Recursively filtering a signal with low and high pass FIR filters generally results in growing bitwidth of the scaling and wavelet coefficients. If we focus on hardware implementation this is equivalent with growing memory requirements to store the coefficients with additional bits. Thus we are interested in finding the smallest bitwidth, which is necessary to store the coefficients without any loss of information.
More formally, we are searching for minimal intervals $[\underline{c_j}, \overline{c_j}]$ and $[\underline{d_j}, \overline{d_j}]$ where $\underline{c_j}, \underline{d_j}, \overline{c_j}, \overline{d_j} \in \mathbb{Z}$, $\underline{c_j} \leq \overline{c_j}$, and $\underline{d_j} \leq \overline{d_j}$ for all $-J < j \leq 0$, such that

$$c_{j,l} \in [\underline{c_j}, \overline{c_j}] \quad \text{and} \quad d_{j,l} \in [\underline{d_j}, \overline{d_j}]$$

holds for all $l \in \mathbb{Z}$ and $-J < j \leq 0$.

Assume an image with a color depth of dpth bits using two's complement number representation is given. Then the range of the pixel values is given as the interval of integers

$$[-2^{\mathrm{dpth}-1}, 2^{\mathrm{dpth}-1} - 1].$$

Thus the minimum and maximum values of all coefficients of $c_{0,i}$ are

$$\begin{aligned}
\overline{c_0} \leq \max_i(\tilde{h} * x)_i &= -\tfrac{-2^{\mathrm{dpth}-1}}{8} + \tfrac{2^{\mathrm{dpth}-1}-1}{4} + \tfrac{3\left(2^{\mathrm{dpth}-1}-1\right)}{4} + \tfrac{2^{\mathrm{dpth}-1}-1}{4} - \tfrac{-2^{\mathrm{dpth}-1}}{8} \\
&= \tfrac{3}{2}2^{\mathrm{dpth}-1} - \tfrac{5}{4} \\
\underline{c_0} \geq \min_i(\tilde{h} * x)_i &= -\tfrac{2^{\mathrm{dpth}-1}-1}{8} + \tfrac{-2^{\mathrm{dpth}-1}}{4} + \tfrac{3\left(-2^{\mathrm{dpth}-1}\right)}{4} + \tfrac{-2^{\mathrm{dpth}-1}}{4} - \tfrac{2^{\mathrm{dpth}-1}-1}{8} \\
&= -\tfrac{3}{2}2^{\mathrm{dpth}-1} + \tfrac{1}{4}
\end{aligned}$$

The same computation can be done for the coefficients $d_{0,j}$:

$$\begin{aligned}
\overline{d_0} \leq \max_j((\tilde{g} * x)_j) &= -\tfrac{-2^{\mathrm{dpth}-1}}{2} + 2^{\mathrm{dpth}-1} - 1 - \tfrac{-2^{\mathrm{dpth}-1}}{2} \\
&= 2^{\mathrm{dpth}} - 1 \\
\underline{d_0} \geq \min_j((\tilde{g} * x)_j) &= -\tfrac{2^{\mathrm{dpth}-1}-1}{2} - 2^{\mathrm{dpth}-1} - \tfrac{2^{\mathrm{dpth}-1}-1}{2} \\
&= -(2^{\mathrm{dpth}} - 1)
\end{aligned}$$

Therefore the range of the coefficients $c_{0,i}$ is

$$\left[\quad \left\lfloor -\frac{3}{2}2^{\mathrm{dpth}-1} + \frac{1}{4} \right\rfloor, \left\lceil \frac{3}{2}2^{\mathrm{dpth}-1} - \frac{5}{4} \right\rceil \quad \right],$$

and the range of the coefficients $d_{0,j}$ is

$$\left[\quad -(2^{\mathrm{dpth}}-1), 2^{\mathrm{dpth}}-1 \quad \right].$$

To determine the range of the coefficients of the next levels we could use the already obtained range and compute the maximum and minimum values based on them. This obviously give us a lower and an upper bound for the left and right endpoint of the interval after each wavelet transform level, respectively.
In Table 3.1 we have summarized the range of the coefficients for the one dimensional case. We assume that the original sequence has a color depth of 8 bit. This is a serious choice, because we focus on greyscale images only. Furthermore, color images in 24 bit color depth are treated as three 8 bit greyscale images. For detailed introduction of the representation of images in the digital world we refer to Chapter 2. Therefore the same considerations apply for color images. To compute lower and upper bounds for further bitwidths, we refer to the Matlab code in Appendix A.3.

Table 3.1: lower and upper bounds of the endpoints of the coefficient range after each level of transform and the corresponding bitwidth, necessary to store these coefficients

level j	coefficients				bitwidth	
	$\underline{c_{-j+1}}$	$\overline{c_{-j+1}}$	$\underline{d_{-j+1}}$	$\overline{d_{-j+1}}$	$c_{-j+1,k}$	$d_{-j+1,k}$
1	-192	191	-255	255	9	9
2	-288	288	-384	384	9	10
3	-432	432	-576	576	10	11
4	-648	648	-864	864	11	11
5	-972	972	-1296	1296	11	12
6	-1458	1458	-1944	1944	12	12

The given bounds in Table 3.1 can be improved significantly. Until now we have calculated the bounds for each scale independently of each other. We have taken the resulting interval of scale j as input for the computation of the new bounds for the next scale $j-1$. As a result the intervals grow significantly.
Consider the recursive convolution of the signal with the analysis filter $\tilde{h}$ of a given wavelet, which is a low pass filter in general. Therefore in each recursion step more and more of the high frequency components of the signals will vanish. In this context high frequency components are neighboring samples (pixels) with large differences. That means, that those samples become smaller with respect to there magnitude.
In order to calculate better bounds we determine new filters, which lead directly to scale $1-J$. Consider the direct calculation of the coefficients c_{1-J} and d_{1-J} as

$$c_{1-J} = (\underbrace{\tilde{h} * ((\ldots \tilde{h} * ((\tilde{h} * ((\tilde{h}}_{J \text{ times}} * c_1) \downarrow 2)) \downarrow 2) \ldots) \downarrow 2)) \downarrow 2$$

and

$$d_{1-J} = (\tilde{g} * ((\ldots \underbrace{\tilde{h} * ((\tilde{h} * ((\tilde{h}}_{J-1 \text{ times}} * c_1) \downarrow 2)) \downarrow 2) \ldots) \downarrow 2)) \downarrow 2$$

respectively.
If we can interchange the filtering and downsampling operations then the analyse of the resulting filter will give us the facility to compute improved bounds. The following proposition will be very useful.

Lemma 3.1 (Interchange of filtering and up/downsampling) *[VK95]*

i) Downsampling by n followed by filtering with a filter having z-transform $F(z)$ is equivalent to filtering with the upsampled filter $F(z^n)$ before the downsampling.

ii) Filtering with a filter having z-transform $G(z)$ followed by upsampling by n is equivalent to upsampling followed by filtering with $G(z^n)$.

Proof:

We have to show that the equation

$$f * (x \downarrow n) = ((f \uparrow n) * x) \downarrow n \tag{3.1}$$

holds in order to prove part i) of Lemma 3.1. In z-transform domain the left side of Equation (3.1) is obviously equivalent to

$$F(z)\frac{1}{n}\sum_{k=0}^{n-1} X\left(e^{\frac{-i2\pi k}{n}} z^{\frac{1}{n}}\right).$$

On the right side of Equation (3.1) we let $f' = (f \uparrow n) * x$. Then we have in z-transform domain

$$\frac{1}{n}\sum_{k=0}^{n-1} F'\left(e^{\frac{-i2\pi k}{n}} z^{\frac{1}{n}}\right).$$

Since $F'(z) = F(z^n)X(z)$ we obtain

$$\frac{1}{n}\sum_{k=0}^{n-1} F\left(\left(e^{\frac{-i2\pi k}{n}} z^{\frac{1}{n}}\right)^n\right) X(e^{\frac{-i2\pi k}{n}} z^{\frac{1}{n}}) = \frac{1}{n}\sum_{k=0}^{n-1} F\left(e^{-i2\pi k} z\right) X\left(e^{\frac{-i2\pi k}{n}} z^{\frac{1}{n}}\right)$$

which can be transformed into

$$F(z)\frac{1}{n}\sum_{k=0}^{n-1} X(e^{\frac{-i2\pi k}{n}} z^{\frac{1}{n}})$$

by using

$$e^{i2\pi} = 1, e^{-i2\pi k} = (e^{i2\pi})^{-k} = 1, \text{ for } k \in \mathbb{Z}.$$

The second proposition of Lemma 3.1 is trivial, since both sides of the equation

$$(f * x) \uparrow n = (f \uparrow n) * (x \uparrow n) \tag{3.2}$$

leads in z-transform domain to $F(z^n)X(z^n)$.

□

With the help of the proposition of Lemma 3.1 we can interchange the filtering and up/downsampling operations to make the computation of the bounds independent of the underlying signal x.
Let $s^{(0)}$ be the original sequence c_1 filtered with the analysis low pass filter $\tilde{h}$, that is

$$s^{(0)} = \tilde{h} * c_1.$$

Note, that downsampling by two of this intermediate signal $s^{(0)}$ leads to c_0. We can obtain $s^{(j)}$ as

$$s^{(j)} = \tilde{h} * (s^{(j+1)} \downarrow 2)$$

for all $J < j < 0$.

Using Equation (3.1) we can exchange the filtering and the downsampling operation as follows

$$s^{(j)} = ((\tilde{h} \uparrow 2) * s^{(j+1)}) \downarrow 2.$$

That is, we can first compute the filter $\tilde{h}' = (\tilde{h} \uparrow 2)$ before considering signal $s^{(j+1)}$. Since $s^{(j+1)}$ can be derived from $s^{(j+2)}$ in the same manner, we have

$$\begin{aligned} s^{(j)} &= ((\tilde{h} \uparrow 2) * s^{(j+1)}) \downarrow 2 \\ &= ((\tilde{h} \uparrow 2) * (\tilde{h} * (s^{(j+2)} \downarrow 2))) \downarrow 2 \\ &= ((\tilde{h} \uparrow 2) * ((\tilde{h} \uparrow 2) * s^{(j+2)}) \downarrow 2) \downarrow 2 \\ &= ((((\tilde{h} \uparrow 2) \uparrow 2) * (\tilde{h} \uparrow 2) * s^{(j+2)}) \downarrow 2) \downarrow 2. \end{aligned}$$

Since the operation $*$ is associative, we get

$$= (((\tilde{h} \uparrow 4) * (\tilde{h} \uparrow 2)) * s^{(j+2)}) \downarrow 4.$$

After J level we obtain

$$s^{(1-J)} = (((\tilde{h} \uparrow 2^J) * (\tilde{h} \uparrow 2^{J-1}) * \ldots * (\tilde{h} \uparrow 2^1)) * s^0) \downarrow 2^J. \tag{3.3}$$

We will prove this by induction over the number of levels J.

Proof:

First we have to verify that the statement holds for $J = 1$

$$\begin{aligned} s^{(-1)} &= \tilde{h} * (s^{(0)} \downarrow 2) \\ &= ((\tilde{h} \uparrow 2) * s^{(0)}) \downarrow 2, \end{aligned}$$

which was already mentioned.

Now assume, that

$$s^{(1-J)} = (((\tilde{h} \uparrow 2^J) * (\tilde{h} \uparrow 2^{J-1}) * \ldots * (\tilde{h} \uparrow 2^1)) * s^{(0)}) \downarrow 2^J \tag{3.4}$$

holds for some J with $J > 1$. We now have to show that $s^{(-(J+1))}$ can be represented accordingly. Let

$$\tilde{h}^{(J)} = (\tilde{h} \uparrow 2^J) * (\tilde{h} \uparrow 2^{J-1}) * \ldots * (\tilde{h} \uparrow 2^1)$$

be the iterated and upsampled filter corresponding to level J.

Using Equation (3.4) we obtain

$$\begin{aligned} s^{(-(J+1))} &= \tilde{h} * (s^{(-J)} \downarrow 2) \\ &= \tilde{h} * \left(\left[(\tilde{h}^{(J)} * s^{(0)}) \downarrow 2^{(J)} \right] \downarrow 2 \right) \\ &= \tilde{h} * \left[(\tilde{h}^{(J)} * s^{(0)}) \downarrow 2^{(J+1)} \right] \\ &= \left((\tilde{h} \uparrow 2^{(J+1)}) * (\tilde{h}^{(J)} * s^{(0)}) \right) \downarrow 2^{(J+1)} \\ &= \left((\tilde{h} \uparrow 2^{(J+1)}) * \left[((\tilde{h} \uparrow 2^J) * (\tilde{h} \uparrow 2^{J-1}) * \ldots * (\tilde{h} \uparrow 2^1)) * s^{(0)} \right] \right) \downarrow 2^{(J+1)} \\ &= \left(((\tilde{h} \uparrow 2^{(J+1)}) * (\tilde{h} \uparrow 2^J) * (\tilde{h} \uparrow 2^{J-1}) * \ldots * (\tilde{h} \uparrow 2^1)) * s^{(0)} \right) \downarrow 2^{(J+1)} \end{aligned}$$

which proves the supposition. □

In an analogous manner we define the filter $\tilde{g}^{(J)}$ for the high pass filter $\tilde{g}$. In Table 3.2 we have summarized the range of the coefficients for the one dimensional case. We again assume that the original sequence has a color depth of 8 bit.

Table 3.2: Range of coefficients after each level of transform

level j	coefficients				bitwidth	
	$\underline{c_{-j+1}}$	$\overline{c_{-j+1}}$	$\underline{d_{-j+1}}$	$\overline{d_{-j+1}}$	$c_{-j+1,k}$	$d_{-j+1,k}$
1	-192	191	-255	255	9	9
2	-208	207	-319	319	9	10
3	-216	215	-351	351	9	10
4	-217	216	-358	358	9	10
5	-219	218	-360	360	9	10
6	-217	216	-363	363	9	10

3.1 Estimation of Coefficients Range using Lifting with Rounding

The above computed ranges of coefficients at different scales and orientation refer to filter bank or Lifting implementations of the CDF(2,2) wavelet transform. We do not yet considered the case of a integer-to-integer mapping of this wavelet transform, which will be used in our hardware architecture. Let us consider the modified Lifting Scheme with respect to rounding from Section 1.8.1. The prediction step is given as

$$d'_{j,k} = c'_{j+1,2k+1} - \left\lfloor \frac{1}{2}(c'_{j+1,2k} + c'_{j+1,2k+2}) \right\rfloor,$$

and the update step as

$$c'_{j,k} = c'_{j+1,2k} + \left\lfloor \frac{1}{4}(d'_{j,k} + d'_{j,k-1}) \right\rfloor.$$

The proven lower and upper bounds for the endpoints of the coefficient interval could be violated by rounding. Therefore, let us estimate the maximum error that can occur, if we use rounded predict and update steps.
Consider the maximum distance between $d'_{j,k}$ and $d_{j,k}$ and $c'_{j,k}$ and $c_{j,k}$ for all $k \in \mathbb{Z}$ and $-J < j \leq 0$, that is

$$\left|d'_{j,k} - d_{j,k}\right| \qquad \text{and} \qquad \left|c'_{j,k} - c_{j,k}\right|,$$

respectively.
We can deduce estimations of these distances using the triangle inequality, that is

$$\begin{aligned}
\left|d'_{j,k} - d_{j,k}\right| &= \left| c'_{j+1,2k+1} - \left\lfloor \frac{c'_{j+1,2k} + c'_{j+1,2k+2}}{2} \right\rfloor - c_{j+1,2k+1} + \frac{c_{j+1,2k} + c_{j+1,2k+2}}{2} \right| \\
&\leq \left| c'_{j+1,2k+1} - \frac{c'_{j+1,2k} + c'_{j+1,2k+2}}{2} - c_{j+1,2k+1} + \frac{c_{j+1,2k} + c_{j+1,2k+2}}{2} \right| + 1 \\
&\leq \left| c'_{j+1,2k+1} - c_{j+1,2k+1} \right| + \left| \frac{c'_{j+1,2k} - c_{j+1,2k}}{2} \right| + \left| \frac{c'_{j+1,2k+2} - c_{j+1,2k+2}}{2} \right| + 1
\end{aligned}$$

and

$$\begin{aligned}
\left|c'_{j,k} - c_{j,k}\right| &= \left| c'_{j+1,2k} + \left\lfloor \frac{d'_{j,k} + d'_{j,k-1}}{4} \right\rfloor - c_{j+1,2k} - \frac{d_{j,k} + d_{j,k-1}}{4} \right| \\
&\leq \left| c'_{j+1,2k} - c_{j+1,2k} \right| + \left| \frac{d'_{j,k} - d_{j,k}}{4} \right| + \left| \frac{d'_{j,k-1} - d_{j,k-1}}{4} \right| + 1
\end{aligned}$$

in general for $-J < j < 0$ and $k \in \mathbb{Z}$.

The distance of $\left|d'_{0,k} - d_{0,k}\right|$ can be estimated somewhat better as

$$\begin{aligned}\left|d'_{0,k} - d_{0,k}\right| &= \left|c'_{1,2k+1} - \left\lfloor \frac{c'_{1,2k} + c'_{1,2k+2}}{2} \right\rfloor - c_{1,2k+1} + \frac{c_{1,2k} + c_{1,2k+2}}{2}\right| \\ &= \left|\frac{c_{1,2k} + c_{1,2k+2}}{2} - \left\lfloor \frac{c'_{1,2k} + c'_{1,2k+2}}{2} \right\rfloor\right| \\ &\leq \frac{1}{2},\end{aligned}$$

because $c_{1,2k}$ and $c_{1,2k+2}$ are integers and $c'_{1,k} = c_{1,k}$ for all $k \in \mathbb{Z}$.
The special case for the distance $\left|c'_{0,k} - c_{0,k}\right|$ is estimated by

$$\begin{aligned}\left|c'_{0,k} - c_{0,k}\right| &\leq \left|c'_{1,2k} - c_{1,2k}\right| + \left|\frac{d'_{0,k} - d_{0,k}}{4}\right| + \left|\frac{d'_{0,k-1} - d_{0,k-1}}{4}\right| + 1 \\ &= \frac{1}{4}\cdot\frac{1}{2} + \frac{1}{4}\cdot\frac{1}{2} + 1 = \frac{5}{4}\end{aligned}$$

as $c_{1,k} = c'_{1,k}$ for all $k \in \mathbb{Z}$.
In summary, let $\mathrm{dist}_c(l)$ and $\mathrm{dist}_d(l)$ be the estimation for the distance $\left|c'_{-l,k} - c_{-l,k}\right|$ and $\left|d'_{-l,k} - d_{-l,k}\right|$, respectively. Then it holds, that

$$\mathrm{dist}_d(l) = 2\,\mathrm{dist}_c(l-1) + 1 \tag{3.5}$$

and

$$\begin{aligned}\mathrm{dist}_c(l) &= \mathrm{dist}_c(l-1) + \frac{1}{2}\,\mathrm{dist}_d(l) + 1 \\ &= 2\,\mathrm{dist}_c(l-1) + \frac{3}{2}\end{aligned} \tag{3.6}$$

for $l > 1$ and

$$\mathrm{dist}_d(0) = \frac{1}{2} \quad \text{and} \quad \mathrm{dist}_c(0) = \frac{5}{4}$$

for $l = 0$.
These are *first order recurrence equations* [And01].

Let us give a definition of those types of equations first.

Definition 2 *Let $f : \mathbb{N}_0 \longrightarrow \mathbb{R}$ be a mapping from integers to the real numbers, then the relation*

$$f(n) = a \cdot f(n-1) + b, f(0) = c$$

with $a, b, c \in \mathbb{R}$ and $n \in \mathbb{N}_0$ is called first order recurrence.

The explicit solution for such $f(n)$ is given in the following theorem.

Theorem 1 *Let f be a real valued sequence, such that*

$$f(n) = a \cdot f(n-1) + b, f(0) = c$$

with $a, b, c \in \mathbb{R}$ and $n \in \mathbb{N}_0$ and $a \neq 1$ then it holds that

$$f(n) = a^n \cdot f(0) + b \cdot \frac{a^n - 1}{a - 1} \tag{3.7}$$

for $n > 0$.

Using Equation (3.7) and setting $a = 2$ and $b = \frac{3}{2}$ we obtain explicit version of Equation (3.5) and Equation (3.6) leading to

$$\mathrm{dist}_c(l) = \frac{11}{4}2^l - \frac{3}{2} \tag{3.8}$$

$$\mathrm{dist}_d(l) = \frac{11}{4}2^l - 2 \tag{3.9}$$

for all $0 < l < J$.

In Table 3.3 you can see the estimations for dist_c and dist_d up to level five.

Table 3.3: error estimation of the coefficients range using Lifting with rounding in comparison to Lifting without rounding with respect to the bitwidth

level l	$\mathrm{dist}_c(l-1)$	$\mathrm{dist}_d(l-1)$
1	1.25	0.5
2	4	3.5
3	9.5	9
4	20.5	20
5	42.5	42

With the help of Equation (3.8) and Equation (3.9) we have computed the estimations for dist_c and dist_d up to level 5. These estimations were added to the intervals from Table 3.2 and are illustrated in Figure 3.1 and Figure 3.2. As you can see, up to level four there is no need to provide additional bits as the intervals still

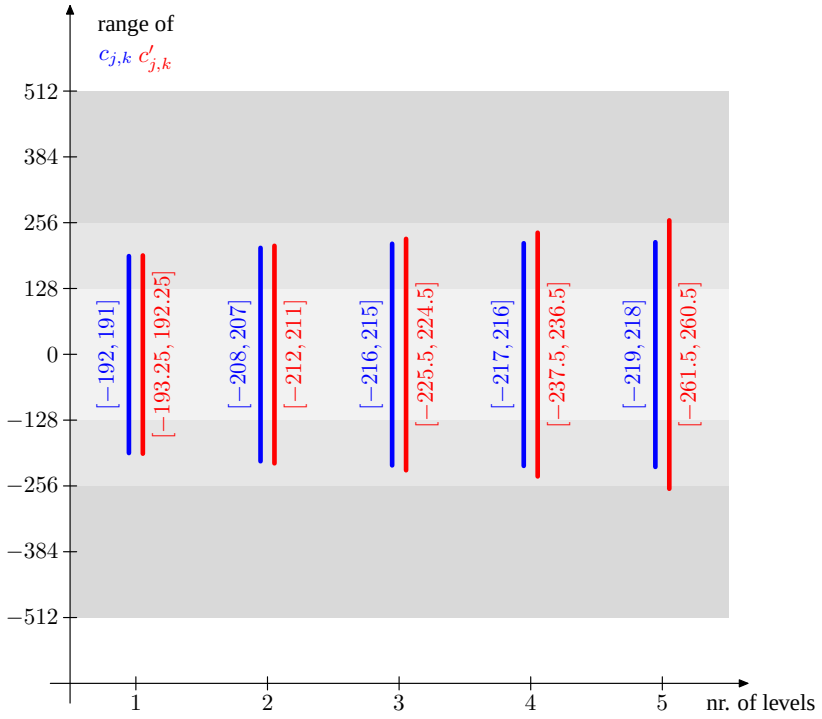


Figure 3.1: range of the coefficients $c_{j,k}$ in comparison to $c'_{j,k}$ with respect to the range of the two's complement number representation

reside inside the range $\left[-2^9, 2^9 - 1\right]$ and $\left[-2^{10}, 2^{10} - 1\right]$, respectively. Due to the partitioned approach

to wavelet transforms discussed in Chapter 5 and the given prototyping platform with limited on chip memory resources we will perform up to 4 levels. Therefore the given bounds of Table 3.2 still holds for our application.

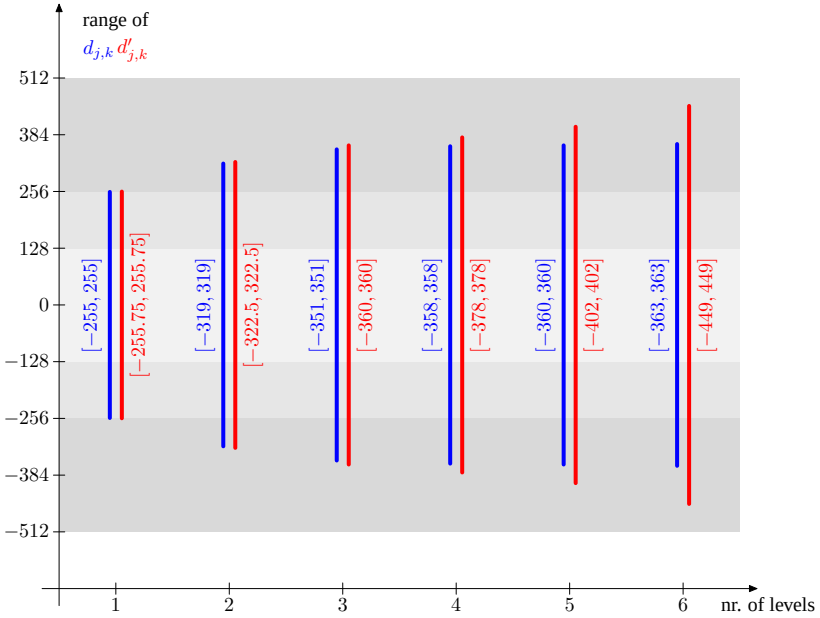


Figure 3.2: range of the coefficients $d_{j,k}$ in comparison to $d'_{j,k}$ with respect to the range of the two's complement number representation

The careful reader would have noticed, that the range for the coefficients $d'_{0,k}$ for all $k \in \mathbb{Z}$ remains the same in Figure 3.2. In contrast to the estimation of the error $\mathrm{dist}_d(0) = 0.5$ we can show that the smallest and greatest coefficients do not exceed the range $[-255, 255]$ as

$$\begin{aligned} \overline{d_0} &= 127 - \frac{-128 - 128}{2} \\ &= 127 - \left\lfloor \frac{-128 - 128}{2} \right\rfloor \\ &= \max_k d'_{0,k} \end{aligned}$$

and

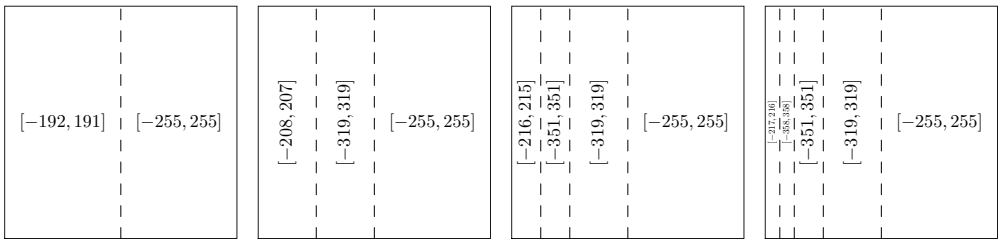
$$\begin{aligned} \underline{d_0} &= -128 - \frac{127 + 127}{2} \\ &= -128 - \left\lfloor \frac{127 + 127}{2} \right\rfloor \\ &= \min_k d'_{0,k} \end{aligned}$$

Thus, we see that the computed bitwidths of Table 3.2 are large enough if the computation of the one dimensional CDF(2,2) wavelet transform was done using the Lifting Scheme with or without rounding.

3.2 Range of coefficients in the two dimensional case

If we are talking about image wavelet transforms we apply tensor products of one dimensional transforms. Remember that in the traditional case vertical and horizontal transforms are executed alternatively. Now

we apply the filters $\tilde{h}$ and $\tilde{g}$ to coefficients which are already filtered by either $\tilde{h}$ or $\tilde{g}$. This influences the precomputation of the filters as it was shown in Equation (3.4). Accordingly to the multiresolution scheme, which was introduced in Section 2.2, we get coefficient ranges, which are different for each scale and each orientation. In Figure 3.3 we have repeated the intervals of the different subbands from Table 3.2. The

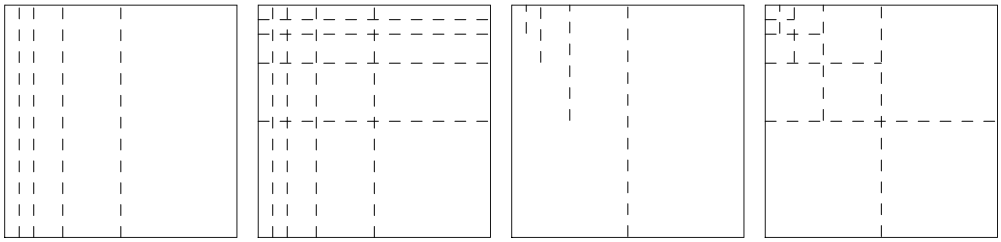


(a) (b) (c) (d)

Figure 3.3: coefficient range after four level of the CDF(2,2) wavelet transform in horizontal direction, (the bounds are taken from Table 3.2)

bounds of the endpoints of these coefficient ranges were deduced, since we have performed all horizontal transforms first (in our example four). This results in a slightly different decomposition scheme as the tree structured one introduced in Section 2.2.

In order to obtain good estimations of the coefficient ranges in the two dimensional case we want to apply all vertical transforms at once, too. Our starting point is the decomposition scheme shown Figure 3.3(d) with the corresponding maximum and minimum values. The order of calculation is depicted in Figure 3.4(a) and Figure 3.4(b). The coefficient ranges can now be obtained by taking the maximum and minimum over all different ranges in a given subblock of the tree structured decomposition.



(a) (b) (c) (d)

Figure 3.4: (a) subdivision in coefficient blocks with different ranges after all horizontal transforms have taken place, (b) after all vertical transforms, (c) different rows transformed with different number of levels, (d) the traditional multiscale representation

Another way to preserve the traditional decomposition type is to apply only that much levels of transform in horizontal and vertical direction, that are necessary. This is illustrated in Figure 3.4(c) and Figure 3.4(d). Note, that we have to take the maximum and minimum over all rows in the column under consideration before calculating the bounds of the endpoints of the coefficients range in vertical direction.

All calculations were executed using Matlab. We refer to Appendix A.3 for the corresponding scripts. As a consequence our memory modules for the 2D-DWT should have at minimum the bitwidth given in Figure 3.5.

The hardware implementation of these memory modules is done using slices of different bitwidth. These slices are shown in Figure 3.5 as dashed lines. Note, that we take the maximum bitwidth of the several subbands of the slice. For further details of the implementation itself see Section 6.3.2.

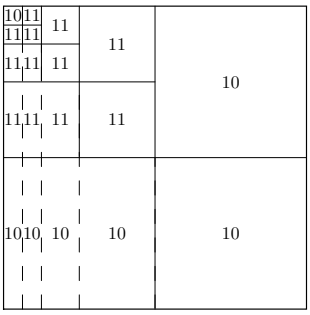


Figure 3.5: minimal affordable memory bitwidth

Chapter 4

State of the art Image Compression Techniques

In this thesis we focus mainly on the adaption of state of the art wavelet based image compression techniques to programmable hardware. Thus, an understanding of these methods is basically. Therefore we take a closer look at these algorithms in this chapter.
In 1993, Shapiro has presented an efficient method to compress wavelet transformed images. This *embedded zerotree wavelet* (*EZW*) encoder exploits the properties of the multiscale representation. An significant improvement of this central idea was introduced by Said and Pearlman in 1996. Their algorithm *Set Partitioning In Hierarchical Trees* (*SPIHT* (pronounced: *spite*)) is explained and analyzed in detail in the following.

4.1 Embedded Zerotree Wavelet Encoding

To explore the special properties of the multiresolution scheme after a wavelet transform of several levels, Shapiro [Sha93] has proposed an algorithm called *embedded zerotree wavelet encoder*. This algorithm takes effort from the self similarities of the different scales in each orientation. For illustration see Figure 4.1. The key idea behind this special codec is to model the self similarities as zerotrees and try to exclude huge areas in lower scales of the image from compression due to only one coefficient located in higher scales of the multiresolution scheme.
This results in a so called embedded bit stream. In such a bit stream, the important information comes first. Thus, in terms of compression any prefix of the bit stream represents a coarse version of the original. The visual quality of the reconstructed images increases as the prefix becomes longer.

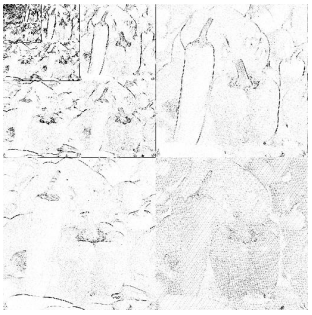
4.1.1 Wavelet Transformed Images

Before embedded zerotree like algorithms are applied, a wavelet transform is performed on the image. This results in a multiscale representation. The transform reduces the correlation between neighboring pixels. The energy of the original image is concentrated in the lowest frequency band of the transformed image. We have already mentioned the specific properties of this decomposition in Section 2.2.
Additionally, self similarities between different scales which result from the recursive application of the wavelet transform step to the low frequency band can be observed (see Figure 4.1). Consequently, based upon these facts good compression performance can be achieved if those coefficients are first transmitted which represent most of the image energy.

4.1.2 Shapiro's Algorithm

The overall encoding procedure is basicly a kind of *bitplane* coding. Thereby the kth bits of the coefficients constitute a *bitplane*. In general, a bitplane encoder starts coding with the most significant bit of each

(a) *peppers*, $N = 512$, $\mathrm{dpth} = 8$



(b) *peppers* CDF(2,2) transformed, (six levels)

Figure 4.1: self similarities illustrated at the wavelet transformed image *peppers* up to six levels

coefficient. When all those bits are coded, the next bitplane is considered until the least significant bits are reached. Within a bitplane the bits of the coefficients with largest magnitude come first. We have illustrated this ordering in Figure 4.2 as bar chart. The coefficients are shown in decreasing order from left to right. Each coefficient is represented with eight bit, where the least significant bit is in front.

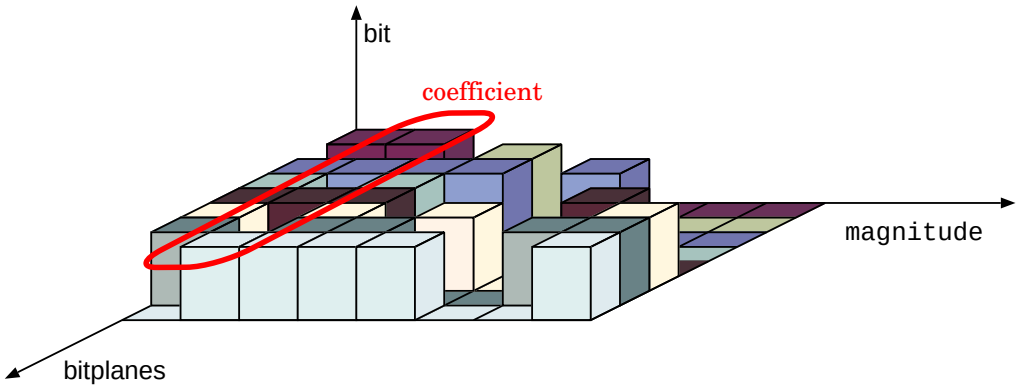


Figure 4.2: bitplane coding

As a consequence, such a bitplane coding scheme encodes the important information in terms of compression efficiency first, if a energy compacting transform was applied before (see Equation (2.2)). On the other hand, we have to store or transmit the ordering information, which can scatter the compression effect.

Shapiro does not reorder the wavelet transform coefficients. He proposes to scan the samples from left to right and from top to bottom within each subband, starting in the upper left corner. The subbands LH, HL, and HH at each scales are scanned in that order. Furthermore, in contrast to traditional bitplane coders he has introduced data dependent examination of the coefficients. The idea behind is, if there are large areas with unimportant samples in terms of compression, they should be excluded from exploration. The addressed self similarities are the key to perform such exclusions of large areas.

In order to exploit the self similarities during the coding process, oriented trees of outdegree four are taken for the representation of a wavelet transformed image. Each node of the trees represents a coefficient of the transformed image. The levels of the trees consist of coefficients at the same scale. The trees are rooted at the lowest frequency subbands of the representation. In Figure 4.3 we have only one coefficient in the lowest frequency band $\mathrm{LL}^{(4)}$. The corresponding node has three children, namely $\hat{\mathcal{I}}(0,1)$, $\hat{\mathcal{I}}(1,0)$, and

$\hat{\mathcal{I}}(1,1)$, depicted with blue, green, and red color, respectively. Each coefficient in the LH, HL, and HH subbands of each scale has four children. The corresponding trees with all their coefficients are drawn with appropriate brightened color. The coefficients at the highest frequency subbands have no children.

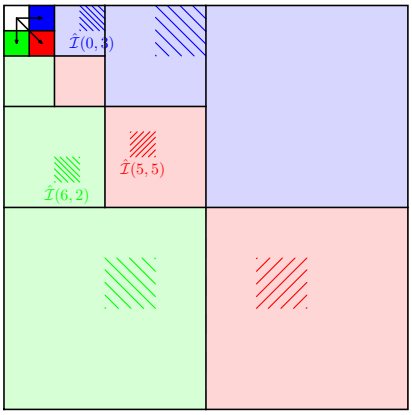


Figure 4.3: oriented quad tree's, four transform level, $N = 16$

To address the data structure more precisely, we have emphasized three sample nodes of different trees, namely at index positions $\hat{\mathcal{I}}(0,3)$, $\hat{\mathcal{I}}(5,5)$, and $\hat{\mathcal{I}}(6,2)$. Each of them represents a root of a subtree, too. The four children are marked with hatching in the corresponding color.

Shapiro assumes that each coefficient c is a good predictor of the coefficients which are represented by the subtree rooted by c. The overall procedure is controlled by an attribute assigned to each coefficient c of the quad trees which gives information on the significance of the coefficients rooted by c.

Definition 3 *A coefficient of the wavelet transformed image is insignificant with respect to a threshold* th *if its magnitude is smaller than* 2^{th}, $\mathrm{th} \in \mathbb{N}_0$.
Otherwise it is called significant with respect to the threshold th.

In the following we have written Shapiro's algorithm in pseudo code notion.

```
th := k_max;
while (th > 0) {
     dominant_pass();
     th--;
     subordinate_pass();
}
```

At first we compute the maximum threshold $k_{\max}$.

$$k_{\max} = \lfloor \log_2 \max_{\{(i,j)|0 \le i,j < N\}} |\hat{\mathcal{I}}_{i,j}| \rfloor .$$

It is easy to see that the bits $k_{\max}+1$, $k_{\max}+2$, ..., $\mathrm{dpth}-1$ of all the coefficients are zero so that they have not to be transmitted, i.e., the bitplanes $k_{\max}+1, \ldots, \mathrm{dpth}-1$ must not be processed.

Dominant pass In the dominant pass, the coefficients are scanned in raster order (from left to right and from top to bottom) within the quadrants. The scan starts with the quadrants of the highest transform level. In each transform level the quadrants are scanned in the order HL, LH, and HH. The coefficients are coded by symbol **P**, **N**, **ZTR**, or **IZ**. This is done in the following manner. A coefficient is coded by

- **P**, if it is greater than the given threshold 2^{th} and is positive,
- **N**, if its absolute value is greater than the given threshold 2^{th} and it is negative,

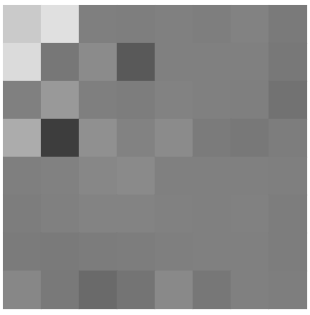
- **ZTR**, if its absolute value is smaller than the given threshold 2^{th} and the absolute value of all coefficients in the corresponding quad tree are smaller than the threshold, too
- **IZ**, if its absolute value is smaller than the given threshold 2^{th} and there exists at least one coefficient in the corresponding quad tree that is greater than the given threshold with respect to the absolute value.

Furthermore there is a symbol **Z** which is used within the high frequency bands of level one only, because all coefficients in these quadrants could not be root of a zerotree. This symbol **Z** can thus be seen as the combination of **ZTR** and **IZ** for this special case.
Once a coefficient is encoded as the symbol **P** or **N** it is not included in the determination of zerotrees.

Subordinate pass Each coefficient, that has been coded as **P** or **N** in the previous dominant pass, is now refined, while coding the th-bit of its binary representation. As already mentioned this corresponds to a bitplane coding, where the coefficients are refined in data dependent manner. The most important fact hereby is, that no indices of the coefficients under consideration have to be coded. This is done implicitly, due to the order in which they become significant (coded as **P** or **N** in the dominant pass).

Let us conclude the discussion of Shapiro's algorithm with a detailed explanation of the following simple example.

Example 1 *Suppose we want to encode the wavelet transformed image given in Figure 4.4 using the embedded zerotree wavelet algorithm of Shapiro. At first we obtain* $k_{\max} = 6$ *as the maximum threshold.*



75	97	−1	−2	1	−2	3	−6
92	−9	12	−39	1	0	1	−8
1	26	−1	−3	3	1	−1	−14
45	−67	17	3	12	−5	−8	−3
−1	2	8	11	1	1	1	−1
−3	0	4	4	2	1	2	−3
−5	−6	−4	−3	−1	0	0	−3
8	−7	−22	−12	10	−9	1	−2

(a) greyscales (b) coefficients

Figure 4.4: wavelet transformed portion of an image, $N = 8$, $\text{dpth} = 8$

Thus, in the first dominant pass we are looking for coefficients whose absolute value is greater than or equal $2^6 = 64$*. The encoding starts as follows:*

Scanning $\text{LL}^{(3)}$: read $75 \Rightarrow$ output **P**
Scanning $\text{HL}^{(3)}$: read $97 \Rightarrow$ output **P**
Scanning $\text{LH}^{(3)}$: read $92 \Rightarrow$ output **P**
Scanning $\text{HH}^{(3)}$: read $-9 \Rightarrow$ output **ZTR**
Scanning $\text{HL}^{(2)}$: read $-1 \Rightarrow$ output **ZTR**
read $-2 \Rightarrow$ output **ZTR**
read $12 \Rightarrow$ output **ZTR**
read $-39 \Rightarrow$ output **ZTR**
Scanning $\text{LH}^{(2)}$: read $1 \Rightarrow$ output **ZTR**
read $26 \Rightarrow$ output **ZTR**
read $45 \Rightarrow$ output **ZTR**
read $-67 \Rightarrow$ output **N**
Scanning $\text{LH}^{(1)}$: read $-4 \Rightarrow$ output **ZTR**

```
read −3 ⇒ output ZTR
read −22 ⇒ output ZTR
read −12 ⇒ output ZTR
```

As you can see, the several subbands are scanned in the given order. In contrast to a bitplane coder, e.g., the subbands $HH^{(2)}$ *and HH are completely excluded from the scan, because* $\hat{\mathcal{I}}_{1,1}$ *represents a zerotree. Furthermore the subband HL is excluded and the subband LH is scanned only partially. The corresponding block is depicted in Figure 4.4(b).*

In the subordinate pass we have to refine four coefficients, which were encoded with the symbols **P** *and* **N** *in the previous dominant pass. These are the coefficients* $\hat{\mathcal{I}}_{0,1}$, $\hat{\mathcal{I}}_{1,0}$, $\hat{\mathcal{I}}_{0,0}$, *and* $\hat{\mathcal{I}}_{3,1}$ *producing the output*

$$1\ 0\ 0\ 0.$$

They are ordered in decreasing order of the reconstructed values. The reconstructed coefficients are initialized to zero at the decoder. If a coefficient is coded with the symbol **P** *or* **N** *using threshold* k *then the reconstructed value is set to the middle of the interval* $[2^k, 2^{k+1}]$. *Depending on the next refinement bit the reconstructed value is adapted accordingly, resulting in an interval of half the size. In our example the first refinement bit of the coefficient* $\hat{\mathcal{I}}(3,1)$ *is one (it became significant in bitplane* 6*). Since* $\hat{\mathcal{I}}(3,1) = 97 > 96 = 64 + 32$ *the* 5*th bit of the binary representation is set. Therefore the new reconstructed value is set to* 112*, the middle of the interval* $[96, 128)$. *The ordering of the refinement bits improves the creation of an embedded compressed bitstream.*

4.2 *SPIHT*- Set Partitioning In Hierarchical Trees

Said and Pearlman have significantly improved the codec of Shapiro. The main idea is based on partitioning of sets, which consists of coefficients or representatives of whole subtrees [SP96]. They classify the coefficients of a wavelet transformed image in three sets:

1. the **l**ist LIP of **i**nsignificant **p**ixels which contains the coordinates of those coefficients which are insignificant with respect to the current threshold th.
2. the **l**ist LSP of **s**ignificant **p**ixels which contains the coordinates of those coefficients which are significant with respect to th, and
3. the **l**ist LIS of **i**nsignificant **s**ets which contains the coordinates of the roots of insignificant subtrees.

During the compression procedure, the sets of coefficients in LIS are refined and if coefficients become significant they are moved from LIP to LSP.
The first difference to Shapiro's EZW algorithm is the distinct definition of the significance. Here, the root of the tree is excluded from the computation of the significance attribute, which can be explained more simply using the following notations.

4.2.1 Notations

For all m with $0 < m \leq \log_2 N$, let $\mathcal{H} = \mathcal{H}(m)$ be the set of the coordinates of the tree roots after m wavelet transform steps, i.e.,

$$\mathcal{H} := \{(i,j) \mid 0 \leq i,j < \frac{N}{2^{m-1}}\}. \tag{4.1}$$

Furthermore, let

$$\mathcal{O}(i,j) := \left\{ \begin{array}{ccc} (2i,2j) & , & (2i,2j+1), \\ (2i+1,2j) & , & (2i+1,2j+1) \end{array} \right\} \tag{4.2}$$

be the set of the coordinates of the children of node (i,j), in the case that this node has children. The set $\mathcal{D}(i,j)$ is composed of the descendants of node (i,j), and

$$\mathcal{L}(i,j) := \mathcal{D}(i,j) \backslash \mathcal{O}(i,j) \tag{4.3}$$

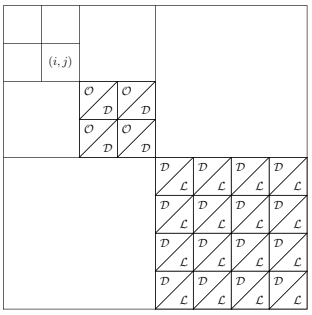


Figure 4.5: the sets $\mathcal{O}(i,j)$, $\mathcal{D}(i,j)$ and $\mathcal{L}(i,j)$ with $i = 1$ and $j = 1$, the labels $\mathcal{O}$, $\mathcal{D}$, $\mathcal{L}$ show, that this coefficients is a member of the corresponding set, $(N = 8)$

is the set of the descendants excluding the four children. Figure 4.5 illustrates these sets.
Each element of LIS will have as attribute either A or B. If the type of an entry $(i,j) \in$ LIS is A, then the entry (i,j) represents set $\mathcal{D}(i,j)$. If the type is B, then it represents $\mathcal{L}(i,j)$.

4.2.2 Significance Attribute

Now let us define the significance of these sets.

Definition 4 *A set $\mathcal{T}$ of coefficients is called* insignificant with respect to the threshold th, $\mathrm{th} \in \mathbb{N}_0$, *if all of its elements are insignificant with respect to the threshold* th.
More formally, we define a function S_{th}

$$S_{\mathrm{th}}(\mathcal{T}) := \begin{cases} 1, & \max\limits_{(i,j)\in\mathcal{T}} |\hat{\mathcal{I}}_{i,j}| \geq 2^{\mathrm{th}} \\ 0, & \textit{else} \end{cases} \tag{4.4}$$

which characterizes the significance of $\mathcal{T}$.
Therefore, a set $\mathcal{T}$ is called insignificant if and only if $S_{\mathrm{th}}(\mathcal{T}) = 0$.
The significance of a single coefficient $S_{\mathrm{th}}(\{\hat{\mathcal{I}}(i,j)\})$ is denoted as $S_{\mathrm{th}}(i,j)$, in short.

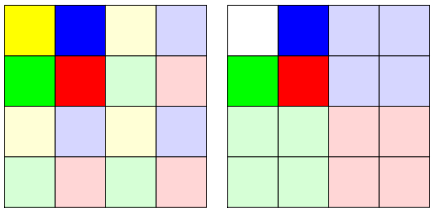
In the *SPIHT* algorithm the signification is computed for the sets $\mathcal{D}(i,j)$ and $\mathcal{L}(i,j)$. As you will see, the root of each quadtree is, in contrast to the algorithm presented by Shapiro, not included in the computation of the significance.

4.2.3 Parent-Child Relationship of the LL Subband

Shapiro proposed, that each element in the LL subband has three children as depicted in Figure 4.6(a). For example, the coefficient $\hat{\mathcal{I}}_{0,1}$ (blue color) has the three children $\hat{\mathcal{I}}_{0,3}$, $\hat{\mathcal{I}}_{2,1}$, and $\hat{\mathcal{I}}_{2,4}$ (depicted with brighten color). Said and Pearlman have changed this parent-child relationship in the LL subband as shown in Figure 4.6(b). Here, each fourth coefficient is not a tree root. In the example the coefficient $\hat{\mathcal{I}}_{0,0}$ does not have any children. More formally, all coefficients in the set $\mathcal{H}$ with even row and even column index have no children.

4.2.4 The basic Algorithm

In general, the decoder duplicates the execution path of the encoder as it was also the case in Shapiro's algorithm. To ensure this behavior, the coder sends the result of a binary decision to the decoder before a branch is taken in the algorithm. Thus, all decisions of the decoder are based on the received bits.
The name of the algorithm is composed of the words *set* and *partitioning*. The sets $\mathcal{O}(i,j)$, $\mathcal{D}(i,j)$ and $\mathcal{L}(i,j)$ were already mentioned. Now we introduce the set partitioning rules of Said and Pearlman.



(a) in Shapiro's EZW algorithm

(b) in the *SPIHT* algorithm

Figure 4.6: different parent-child relationships of the LL band

- the initial partition is formed with the sets
  - $\{(i,j)\}$ for all $(i,j) \in \mathcal{H}$ and
  - $\mathcal{D}(i,j)$ for all $\{(i,j)|(i,j) \in \mathcal{H}$ with $\mathcal{D}(i,j) \neq \emptyset\}$
- if $S_k(\mathcal{D}(i,j)) = 1$ then $\mathcal{D}(i,j)$ is partitioned into $\mathcal{L}(i,j)$ and the four sets $\{(e,f)\} \in \mathcal{O}(i,j)$
- if $S_k(\mathcal{L}(i,j)) = 1$ then $\mathcal{L}(i,j)$ is partitioned into the four sets $\{(e,f)\}$, with $(e,f) \in \mathcal{O}(i,j)$

The pseudo code of the basic *SPIHT* algorithm is shown in Figure 4.7.
It can be divided into three parts. The first part initializes the lists, computes, and output the initial threshold similar to the algorithm of Shapiro.

Afterwards, the sorting phase and the refinement phase are executed $k_{\max} + 1$ times starting with bitplane $k_{\max}$ down to bit plane 0. In pass k, the sorting phase first outputs the kth bits of the coefficients in LIP. If the kth bit of some coefficient $\hat{\mathcal{I}}_{i,j}$ is 1, i.e., $S_k(i,j) = 1$, coefficient $\hat{\mathcal{I}}_{i,j}$ is significant with respect to k and insignificant with respect to $k+1$. Therefore, the sign of $\hat{\mathcal{I}}_{i,j}$ has to be output.

Sorting phase The elements of LIS are now processed using the set partitioning rules. First, the significance bit for set $\mathcal{D}(i,j)$ or $\mathcal{L}(i,j)$ is written out, respectively.
The set corresponding to an element (i,j) of type A, which is now significant with respect to the current threshold, is partitioned into four single element sets $\{(e,f)\}$ for each $(e,f) \in \mathcal{O}(i,j)$. Each of the coefficients (e,f) are appended to LIP or LSP, depending on there significance. The element (i,j) itself is changed to type B and moved to the end of the list LIS, if the set $\mathcal{L}(i,j)$ is non empty.
The elements (i,j) of type B in LIS are refined to the four elements $(e,f) \in \mathcal{O}(i,j)$ of type A, if the $S_k(\mathcal{L}(i,j)) = 1$.

Refinement phase The refinement phase outputs the kth bit of each elements of list LSP, if it was not included in the last sorting phase.

Another advantage of *SPIHT* in comparison to Shapiro's algorithm is, that the complete outcome of the encoder is binary. Therefore, the compression rate can be achieved exactly and arithmetic encoders for binary alphabets can be applied to further increase the compression efficiency.

```
(0)  Initialization
       compute and output k = ⌊log2 max_{{(i,j)|0≤i,j<N}} |Î_{i,j}|⌋;
       set LSP := ∅ and LIP := H(nrOfLevels)
       add those elements (i,j) ∈ H(nrOfLevels) with D(i,j) ≠ ∅ to LIS as type A;
(1)  Sorting phase
(2)    foreach (i,j) ∈ LIP
(3)      output S_k(i,j);
(4)      if S_k(i,j) = 1 then
           move (i,j) to LSP and output the sign of I_{i,j}
(5)    foreach (i,j) ∈ LIS
(6)      if (i,j) is of type A then
(7)        output S_k(D(i,j));
(8)        if S_k(D(i,j)) = 1 then
(9)          foreach (e,f) ∈ O(i,j)
(10)           output S_k(e,f);
(11)           if S_k(e,f) = 1 then
                 add (e,f) to LSP and output the sign of Î_{e,f}
(12)           else
                 append (e,f) to LIP;
(13)         if L(i,j) ≠ ∅ then
(14)           move (i,j) to the end of
               LIS as an entry of type B and go to step (5)
(15)         else remove (i,j) from LIS
(16)     if (i,j) is of type B then
(17)       output S_k(L(i,j));
(18)       if S_k(L(i,j)) = 1 then
(19)         append each (e,f) ∈ O(i,j)
             to LIS as an entry of type A
             and remove (i,j) from LIS;
(20) Refinement phase
       foreach (i,j) ∈ LSP except those included in the last sorting phase
         output the kth bit of |Î_{i,j}|;
(21) Quantization-step update
(22)   decrement k; if k < 0 stop; else goto step (1);
```

Figure 4.7: The *SPIHT* algorithm

Chapter 5

Partitioned Approach

In this chapter we present the partitioned approach to wavelet transform images. This partitioned approach itself was developed in 1998[1].
At first we motivate, why there is a need for such an approach, if one considers implementations of image compression algorithm based on wavelet transforms in programmable hardware. We discuss accurately why these wavelet based image compression algorithms need to be adapted at all to implement them into programmable hardware. Then we present the application of the partitioned approach to lossless image compression. The more interesting investigation is the extension to lossy image compression, where we have to deal with boundary effects between the image partitions. Here we present different solutions in order to avoid block artefacts known from compression methods based on discrete cosine transforms.

In Section 5.3 of this chapter we present the modifications to the original *SPIHT* algorithm, which correspond to the accommodation to the partitioned approach and the necessities for hardware architectures with minimal memory requirements. Then we compare the two algorithms in terms of mean squared error in Section 5.4.

5.1 Drawbacks of the Traditional 2D-DWT on Images

Let us review the traditional two dimensional discrete wavelet transform on images introduced in Chapter 2. We have seen, that images are wavelet transformed using the one dimensional transform applied to rows and columns of the image, separately. In Section 1.8 we have introduced the Lifting Scheme as a technique to compute the transform in place.
In the case, that the image can be stored completely in main memory (software solution) of a PC or in on chip memory (hardware solution) the transform can easily be implemented. The only additional buffer we need has a size comparable to the size of a row or a column, respectively, to reorder the low and high pass coefficients. To compute the 2D-DWT we have to load the image into the memory. Then we transform it using the mentioned buffer, if enough internal memory is available. Typically, images have hundreds of rows and columns and each pixel is represented by one up to 24 bits or even more. Therefore we have to deal with data volumes of several kbytes up to a couple of megabytes. Obviously, this exceeds the memory limits, if we consider hardware solutions. Thus, we have to limit the number of rows and columns and the maximum bitwidth. As a consequence images of larger sizes must be split into manageable pieces. Remember from Section 2.1 that we then have to deal with boundary effects even at the boundary of each piece of the original image, when targeting to lossy compression.

[1] After making first experiences with the corresponding hardware architecture for lossless image compression on FPGAs we have published the basic ideas at the Custom Integrated Circuits Conference in 2000 (CICC2000) [RM00]. The extension to lossy image compression was presented at the ACM conference on Field Programmable Gate Arrays in 2001 (FPGA2001) [RM01]. The FPGA implementation of a modified *SPIHT* algorithm based on the partitioned approach was published at the ACM conference on Field Programmable Gate Arrays (FPGA2002) [RFM02a] and the 45th Midwest Symposium on Circuit and Systems (MWSCAS2002) in 2002 [RFM02b].

On the other hand we can store the image data externally and compute the transform row by row and column by column, reducing the internal memory to the size of a row or column. In that case, the data transfer from and to the internal memory will become the bottleneck of the implementation. Remark, that already transformed coefficients have to be saved and loaded again during the next level of computation. Furthermore, after the whole transform had taken place, the transformed image is stored in external memory, but an embedded zerotree wavelet coder needs random access to it.

To overcome these difficulties we will present the partitioned approach to wavelet transform images, which is applicable to both lossless and lossy compression.

5.2 Partitioned 2D-DWT

Let us start with some notations. Remember from Chapter 2 that we consider an $N \times N$ image as two dimensional pixel array $\mathcal{I}$ with N rows and N columns and assume a dpth-bit greyscale resolution. Furthermore, we assume without loss of generality that the equation $N = 2^k$ holds for some positive integer k.
Partitioning an image $\mathcal{I}$ into $(\frac{N}{q})^2$ quadratic subimages

$$\mathcal{I}^{(0,0)}, \ldots, \mathcal{I}^{(\frac{N}{q}-1, \frac{N}{q}-1)}$$

of size q with $q = 2^r$ for some positive integer $1 \leq r < k$ results in subimages with

$$\mathcal{I}^{(s,t)}[i,j] = \mathcal{I}[s \cdot q + i, t \cdot q + j]$$

for all $0 \leq s, t < \frac{N}{q}$ and $0 \leq i, j < q$.

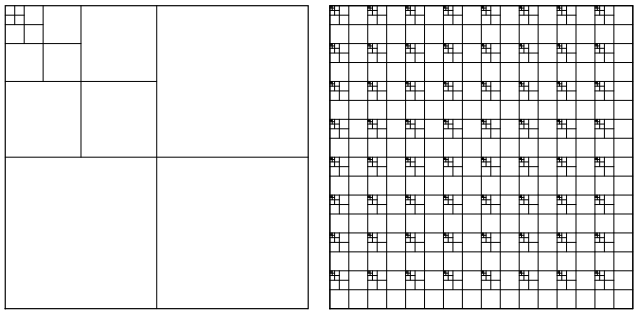
Before the transform takes place, we partition the original image $\mathcal{I}$ into quadratic subimages $\mathcal{I}^{(s,t)}$ as described above. Then each subimage is transformed in the conventional way. We refer for an illustration to Figure 5.1. The first idea is to transform the subimages independently of the neighboring subimages. However, this approach is only applicable, if no quantization takes place, otherwise artefacts are introduced. What are the advantages and disadvantages of this partitioned approach?

+ The internal memory requirements are dramatically reduced.
+ The subimages can be transformed independently of each other.
+ The traditional 2D-DWT can be directly applied to the subimages without any modifications.
− It is not applicable for lossy compression.

For lossy compression we have to take into account the coefficients of the neighboring subimages in order to guarantee that the partitioned approach is as efficient as the traditional one. We will denote this type of transformation as *partitioned approach with boundary treatment*. Suppose we have given a wavelet transformed image $\hat{\mathcal{I}}$ using the traditional approach as shown in Figure 5.1(a). We have to manage that the partitioned transformed image is a permutation of the coefficients of the traditional transformed image. For illustration consider Figure 5.2(b). Consider the LL subband on the left of Figure 5.2(a), which corresponds to the approximated original at half the resolution, and the 64 LL subbands of all partitions on the right. We observe, that composing all LL subbands on the right results in the LL subband on the left.

We can compute the partitioned 2D-DWT with boundary treatment even without the traditional one given first. We will discuss it later in this chapter. Let us summarize first the advantages and disadvantages for this kind of partitioned transform, too.

+ The internal memory requirements are dramatically reduced, too. In contrast to the approach with reflection at the partition boundaries we need additional memory for coefficients of neighboring subimages or we have to deal with increased data transfer volume.
+ It is applicable for lossy compression.



(a) traditional 2D-DWT scheme

(b) 2D-DWT scheme based on the partitioned approach

Figure 5.1: traditional versus partitioned two dimensional wavelet transform of images, $N = 512$, $q = 64$, five level of transform

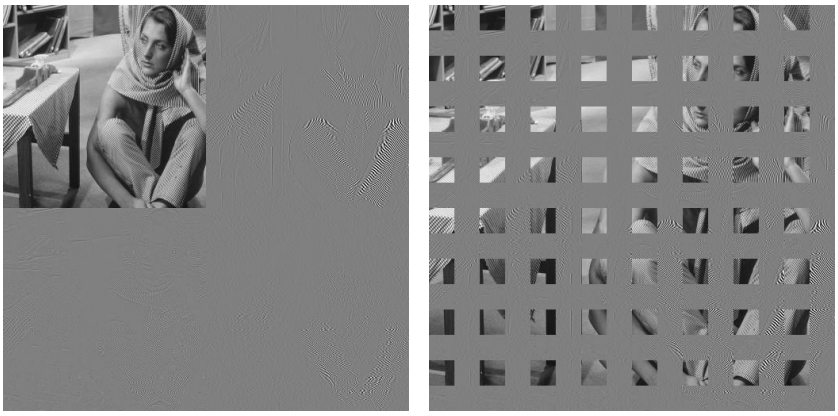
– We reduced the internal memory at the expense of increased computational load or larger data transfer volume. The amount of additional complexity depends on the partition size q and the filter length of the corresponding low and high pass filter of the wavelet.

5.2.1 Lossless Image Compression

Lossless image compression of partitioned images is quite easy. The subimages $\mathcal{I}^{(s,t)}$ can be wavelet transformed independently of the other subimages. Of course, the resulting transformed image which is composed by $\hat{\mathcal{I}}^{(1,1)}, \ldots, \hat{\mathcal{I}}^{(\frac{N}{q},\frac{N}{q})}$ differs from the conventionally wavelet transformed image $\hat{\mathcal{I}}$. However, reversibility is ensured. Note that this simple solution works for lossless image compression as no quantization takes place. Figure 5.1 illustrates this simple approach and demonstrates the potential to parallelize the computation of the wavelet transform.
According to the chosen partition size $q \in \{16, 32, 64\}$ computing up to five levels of transform are possible and reasonable. After having applied the partitioned wavelet transform, we can apply an EZW algorithm on each partition. Thus, random access is only necessary on the subimage itself which is stored in internal memory. This makes the partitioned approach suitable for a FPGA hardware implementation.
To quantify the effectiveness of approach, we have measured the entropy of both, the conventionally wavelet transformed image $\hat{\mathcal{I}}$ and the partitioned wavelet transformed image $\hat{\mathcal{I}}^{(p)}$. We used the integer-to-integer CDF(2,2)-wavelet with modular arithmetic as introduced in Section 1.8.2. Thus, the transformed image also has dpth-bit greyscale resolution. Boundary treatment has been done by reflection at image/partition boundaries. The benchmark images used are the common 8-bit greyscale images of size $N = 512$. The size of the subimages has been set to $q = 64$.
Recall from Section 1.2, that the entropy H is defined as

$$H(\mathcal{I}) \quad = \quad -\sum_i p(\mathcal{I}_{i,j}) \log p(\mathcal{I}_{i,j}),$$

where $p(\mathcal{I}_{i,j})$ represents the cumulative percentage of the corresponding greyscale value.
Table 5.1 summarizes the results. The first column specifies the image under consideration. The second, third, and fourth column gives the entropy H of the original image $\mathcal{I}$, the conventionally wavelet transformed image $\hat{\mathcal{I}}$, and the partitioned wavelet transformed image $\hat{\mathcal{I}}^{(64)}$, respectively. We observe that the entropy of a partitioned wavelet transformed image is only slightly higher than the entropy of the conventionally wavelet transformed image. Thus, the software and hardware solution based on this partitioned approach with reflection at partition boundaries allows compression ratios comparable to those guaranteed by traditional wavelet based solutions.



(a) barbara, traditional 2D-DWT (b) barbara, partitioned 2D-DWT

Figure 5.2: the two schemes can be obtained from each other using a permutation, $N = 512$, $q = 64$, one level of transform

Table 5.1: entropy of the image themself, the traditional, and partitioned discrete wavelet transformed images, $N = 512$, $q = 64$, computed with reflection at image and partition boundaries

image	$H(\mathcal{I})$	$H(\hat{\mathcal{I}})$	$H(\hat{\mathcal{I}}^{(64)})$
airplane	6.70589	4.26506	4.28643
baboon	7.35795	6.16171	6.17883
goldhill	7.47778	4.89972	4.92269
lena	7.44551	4.42012	4.45058
peppers	7.59245	4.71864	4.74100

5.2.2 Lossy Image Compression

If stronger requirements on transmission time or storage space are postulated, higher compression ratios are needed. The only way to meet these requirements is allowing loss of information. This is usually done by quantization after the wavelet transform had taken place. If we apply the approach presented in the previous section, namely transforming each subimage independently of the neighboring subimages by simply using reflection at the boundaries of the subimage, we obtain block artefacts known from JPEG compressed images [Wal91].

Figure 5.3(b) shows the image *lena* ($N = 512$, $\mathrm{dpth} = 8$) transformed with the partitioned approach presented in the previous section. The resulting image $\hat{\mathcal{I}}^{(32)}$ was compressed by a factor 21 using the *SPIHT* algorithm. Thereby we have compressed each subimage separately. This shows that reflection at partition boundaries is no longer feasible.

Our objective is that the partitioned transformed and compressed image does not differ from the conventionally compressed image. Figure 5.3(a) shows the image compressed by a factor 25 if the original image is transformed by a *non partitioned* DWT algorithm and compressed by a factor 25 by the *SPIHT* encoder. This emphasizes the necessity not to use reflection at partition boundaries, but to work with the original pixels.

(a) lena transformed by the traditional 2D-DWT and compressed with *SPIHT* by a factor 25.

(b) lena transformed by the partitioned 2D-DWT presented in Section 5.2.1 and *SPIHT* compressed by a factor 21.

Figure 5.3: partitioned wavelet transform without boundary treatment introduces block artefacts targetting to lossy compression

5.2.3 Boundary Treatment

Let us examine which pixels have to be considered in order to compute the low and high pass coefficients using the CDF(2,2)-wavelet. Considering the filter bank approach to compute the coefficients (compare Equation (1.12)) we obtain the equations

$$\begin{aligned} c_{j,k} &= -\frac{1}{8}c_{j+1,2k-2} + \frac{1}{4}c_{j+1,2k-1} + \frac{3}{4}c_{j+1,2k} + \frac{1}{4}c_{j+1,2k+1} - \frac{1}{8}c_{j+1,2k+2} \text{ and} \\ d_{j,k} &= -\frac{1}{2}c_{j+1,2k} + c_{j+1,2k+1} - \frac{1}{2}c_{j+1,2k+2}. \end{aligned}$$

Now, take a look at a one dimensional discrete signal (a pixel row) r of length 32 which consists of pixels $r_0, \ldots, r_{31}$. The signal r is transformed using the CDF(2,2) wavelet. We obtain 16 low pass coefficients $c_{0,0}, \ldots, c_{0,15}$ and 16 high pass coefficients $d_{0,0}, \ldots, d_{0,15}$. These coefficients depend on the pixels $r_{-2}, r_{-1}, r_0, r_1, \ldots, r_{31}, r_{32}$. Note that in the partitioned approach proposed in Section 5.2.1 the pixels r_{-2}, r_{-1}, and r_{32} are not available in the internal memory. Thus, in order to compute the coefficients $c_{0,0}, \ldots, c_{0,15}$ and $d_{0,0}, \ldots, d_{0,15}$, the transform has to look out over the right and left boundary of the pixel row r by one and two pixels, respectively. In the following transform levels further pixels from outside of the pixel row r have to be accessed. Figure 5.4 shows on which pixels of row r the new coefficients depend on. Note that only the low pass coefficients are transformed during the next levels of a wavelet transform and one has to look out over the neighboring subimages only if more than four levels of transform are performed (in the case that $q = 32$ and the CDF(2,2) wavelet is used).
After four levels of transform, the wavelet transform of the row r of length 32 has to look out over the left and right boundary by 30 and 15 pixels, respectively. Since the same arguments can be deduced for the columns of an image, we have illustrated the effect in two dimensions in Figure 5.5. As one can observe, the number of coefficients from neighboring subimages depends on the low and high pass filter lengths, not on the partition size q. In general, we have to look out over the left and top boundary by

$$\beta = \max\left\{\frac{|\tilde{h}|-1}{2}, \frac{|\tilde{g}|-1}{2} - 1\right\},$$

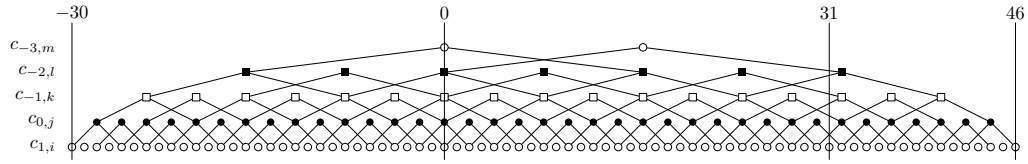
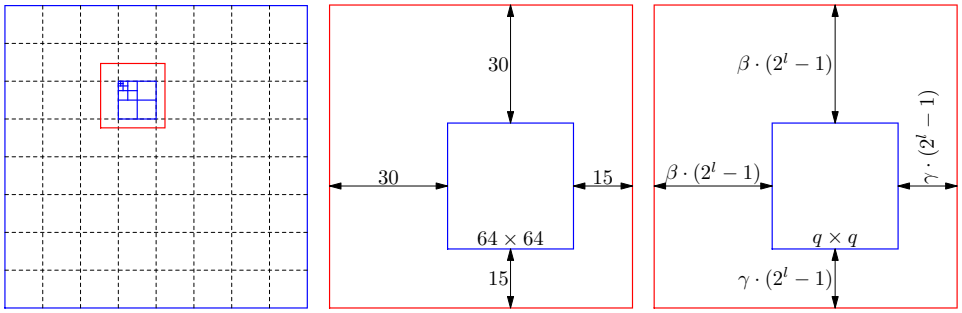


Figure 5.4: the computation of the low pass coefficients and their dependences on the coefficients of the previous scales



(a) $N = 512$, $q = 64$, four levels

(b) $N = 512$, $q = 64$, four levels, CDF(2,2) wavelet

(c) $N = 512$, l level

Figure 5.5: area of coefficients, which have to be incorporated into the computation of the transform of the partition under consideration

and over the right and bottom boundary

$$\gamma \quad = \quad \max\left\{\frac{|\tilde{h}|-1}{2}-1, \frac{|\tilde{g}|-1}{2}\right\}$$

after the first transform level, if we assume that the filters $\tilde{h}$ and $\tilde{g}$ correspond to a symmetric biorthogonal wavelet. The enlargement after l levels is shown in Figure 5.5(c).
Thus, a first solution to implement a partitioned 2D-DWT with boundary treatment is to provide all necessary neighboring coefficients in addition to each subimage. Unfortunately, it can be too expensive to simply store all these additional coefficients in internal memory, too. This would result in an internal memory which is capable to store

$$(q + \beta \cdot (2^l - 1) + \gamma \cdot (2^l - 1))^2$$

coefficients. In our example this results in a extended subimage of size 109×109, which is about 3 times larger than the internal memory used till now (cf. Figure 5.5(b)).

In Chapter 6 we present two approaches to avoid the enlargement of the internal memory when targeting to hardware implementations. These two methods are based on *divide and conquer* techniques and *pipelined architectures*, respectively.

5.2.4 Future Work: Taking Advantage of Subimage and QuadTree Similarities

It is interesting to investigate, whether the approach presented can be used to improve the compression ratio. We were looking for methods allowing the partitioned approach to result in higher compaction ratios than the non partitioned algorithms.

One point of departure could be to classify the subimages with respect to similarity, to compress only one representative $\mathcal{J}_C$ of each class C, and to represent the remaining subimages $\mathcal{I}$ of class C by the image $\mathcal{I} - \mathcal{J}_C$ which specifies the differences between subimage $\mathcal{I}$ and its representative $\mathcal{J}_C$. Another point of departure could be to exploit similarities between the trees of the multiscale representation. There are similarities between the quadtrees. The difficulty is a convincing measure of similarity, which is of adequately computational complexity. We have made some experiences with the mean squared error and the gain of reduced output of the *SPIHT* algorithm. Unfortunately, the computational overload was to large in order to obtain useful results.

5.3 Modifications to the *SPIHT* Codec

In this section we present the modifications necessary to obtain an efficient hardware implementation of the *SPIHT* compressor based on the partitioned approach to wavelet transform images.
At first, we exchange the sorting phase with the refinement phase to save memory for status information. However, the greatest challenge is the hardware implementation of the three lists LIP, LSP, and LIS. In the following we deduce estimations of the memory requirements of these lists. This section will be concluded with the outline of the modified *SPIHT* encoder.

5.3.1 Exchange of Sorting and Refinement Phase

In the basic *SPIHT* algorithm status information has to be stored for the elements of LSP specifying whether the corresponding coefficient has been added to LSP in the current iteration of the sorting phase (see Line 20 in Figure 4.7). In the worst case all coefficients become significant in the same iteration. Consequently, we have to provide a memory capacity of q^2 bits to store this information. However, if we exchange the sorting and the refinement phase, we do not need to consider this information anymore. The compressed data stream is still decodable and it is not increased in size. Of course, we have to consider that there is a reordering of the transmitted bits during an iteration. We will take a closer look at this problem in Section 5.4.

5.3.2 Memory Requirements of the Ordered Lists

To obtain an efficient realization of the lists LIP, LSP, and LIS, we first have to specify the operations that take place on these lists and deduce worst case space requirements in a software implementation.

Estimations for LIS

We have to provide the following operations for LIS:

- initialize as empty list (Line 0),
- append an element (Line 0 and 19),
- sequentially iterate the elements (Line 5),
- delete the element under consideration (Line 15 and 19),
- move the element under consideration to the end of the list and change the type (Line 14).

Note that the initialization phase cannot be neglected because there are several partitions to compress. Dependent on the realization chosen the cost of the initialization phase can be linear in the number of elements.
The estimation of the space requirement for LIS is simplified by the fact that LIS is an independent set, i.e., for all elements $v, w \in$ LIS it holds that v is neither a predecessor nor a successor of w with respect to the corresponding quad tree. This can be easily proved by complete induction [Fey01]. Note that if the list is longest, then all its elements are of type A. Consequently, the number of elements of LIS is smaller

than or equal to the number of nodes in the next to the last tree level. Thus, the corresponding coefficients lie in the left upper quadrant LL. However, none of them lies in the left upper quadrant of LL. Thus, LIS contains at most

$$\left(\frac{q}{2}\right)^2 - \left(\frac{q}{4}\right)^2 = \frac{3}{16}q^2$$

elements. To specify the coordinates of an element, $2\log_2\left(\frac{q}{2}\right)$ bits are needed. A further bit to code the type information has to be added per element. This results in an overall space requirement for LIS of

$$\frac{3}{16}q^2 \cdot \left(2\log_2\left(\frac{q}{2}\right) + 1\right)$$

bits.

Estimations for LIP **and** LSP

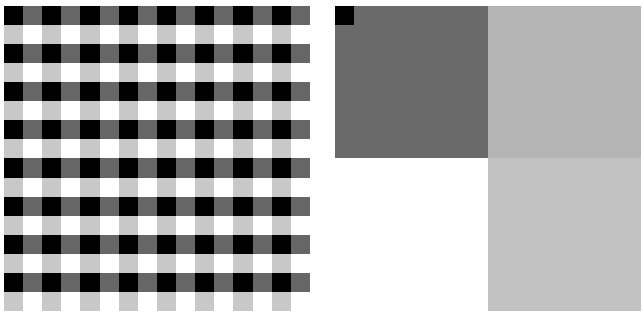
Now, let us consider both the list LIP and the list LSP because they can be implemented together. Again, we start with the operations applied to both lists:

- initialize as empty list (Line 0),
- append an element (Line 0, 4, 11, and 12),
- sequentially iterate the elements (Line 20),
- delete the element under consideration from LIP (Line 4).

The maximum size of both lists is at most q^2. Furthermore, it holds, that $\mathrm{LIP} \cap \mathrm{LSP} = \emptyset$. Thus,

$$|\,\mathrm{LIP}\,| + |\,\mathrm{LSP}\,| \leq q^2$$

also holds. q^2 is a tight upper bound for $|\,\mathrm{LIP}\,| + |\,\mathrm{LSP}\,|$. This worst case is attained by the image shown in Figure 5.6. (**Remark:** For illustration purposes we have changed the greyscales in the figure, so that they do not correspond to the actual values.) It is easy to verify that all coefficients at the lower levels are greater than 126. It follows, that $S_6(\mathcal{D}(i,j)) = 1$ for all $\mathcal{D}(i,j)$ and $S_6(\mathcal{L}(i,j)) = 1$ for all $\mathcal{L}(i,j)$. In the 6th iteration of the algorithm, all sets are split and all coefficients are moved to either LIS or LSP.



(a) Original image (q=16, 8 bit, four greyscales -128, -61, -60, 126 (from black to white))

(b) Image after four transform steps (q=16, 9 bit, five greyscales -33, 0, 126, 127, 128)

Figure 5.6: q^2 is a tight upper bound for $|\,\mathrm{LIP}\,| + |\,\mathrm{LSP}\,|$

Thus, the overall space requirement for both lists is $q^2 \cdot 2\log_2 q$ bits.

(0) **Initialization**
  compute and output $k := \lfloor \log_2 \max_{\{(i,j)|0 \le i,j < N\}} |c_{i,j}| \rfloor$;
  foreach $0 \le i,j \le N-1$ set
    $LSP(i,j) = 0$
    $LIP(i,j) = \begin{cases} 1, & \text{if } (i,j) \in \mathcal{H} \\ 0, & \text{else} \end{cases}$
  foreach $0 \le i,j \le \frac{N}{2} - 1$ set
    $LIS(i,j) = \begin{cases} A, & \text{if } (i,j) \in \mathcal{H} \text{ and } \mathcal{O}(i,j) \neq \emptyset \\ 0, & \text{else} \end{cases}$
(1) **Refinement and sorting phase for list** LIP
(2) for $i = 0 \ldots N-1$
(3) for $j = 0 \ldots N-1$
(4) if $LSP(i,j) = 1$ then
        output the k-th bit of $|c_{i,j}|$;
(5) if $LIP(i,j) = 1$ then
(6) output $S_k(i,j)$;
(7) if $S_k(i,j) = 1$ then
          $LSP(i,j) = 1$; $LIP(i,j) = 0$;
          output the sign of $c_{i,j}$;
(8) **Sorting phase for list** LIS
(9) for $i = 0 \ldots \frac{N}{2} - 1$
(10) for $j = 0 \ldots \frac{N}{2} - 1$
(11) if $LIS(i,j) = A$ then
(12) output $S_k(\mathcal{D}(i,j))$;
(13) if $S_k(\mathcal{D}(i,j)) = 1$ then
(14) foreach $(e,f) \in \mathcal{O}(i,j)$
(15) output $S_k(e,f)$;
(16) if $S_k(e,f) = 1$ then
            $LSP(e,f) = 1$;
            output the sign of $c_{e,f}$;
(17) else $LIP(e,f) = 1$;
(18) if $\mathcal{L}(i,j) \neq \emptyset$ then
(19) $LIS(i,j) = B$;
(20) else $LIS(i,j) = 0$;
(21) if $LIS(i,j) = B$ then
(22) output $S_k(\mathcal{L}(i,j))$;
(23) if $S_k(\mathcal{L}(i,j)) = 1$ then
(24) foreach $(e,f) \in \mathcal{O}(i,j))$
          $LIS(e,f) = A$;
(25) $LIS(i,j) = 0$;
(26) **Quantization-step update**
(27) decrement k; if $k < 0$ stop; else goto step (1);

Figure 5.7: The modified *SPIHT* algorithm

5.4 Comparison between the Original and the Modified *SPIHT* Algorithm

In the previous section we have presented which modifications have been applied to the basic algorithm and the dynamic data structures in order to obtain an efficient hardware implementation. It was ensured that the compressed stream can still be decoded and that exactly the same number of bits are produced. However, the reordering of the embedded stream which is due to the realization of the lists LIP, LSP, and LIS by bitmaps can have an effect on the visual quality of the reconstructed images. In this section, we will investigate this problem in detail. More precisely, we reflect on how much the reconstructed images of both the basic compressor and the modified compressor differ in terms of visual quality.

Notations

Let b be the number of bits already encoded and $\tilde{\mathcal{I}}(b)$ the corresponding reconstructed image after receiving b bits. The basic *SPIHT* algorithm and our modified one can be compared by

$$E(\tilde{\mathcal{I}}(b), \tilde{\mathcal{I}}'(b)) := \left| \mathrm{MSE}(\mathcal{I}, \tilde{\mathcal{I}}(b)) - \mathrm{MSE}(\mathcal{I}, \tilde{\mathcal{I}}'(b)) \right|$$

where $E(\tilde{\mathcal{I}}(b), \tilde{\mathcal{I}}'(b))$ is the difference between the *SPIHT* algorithm and the modified algorithm after coding b bits in terms of mean squared error.
There are two main cases to consider. Figure 5.8 illustrates schematically the following considerations.

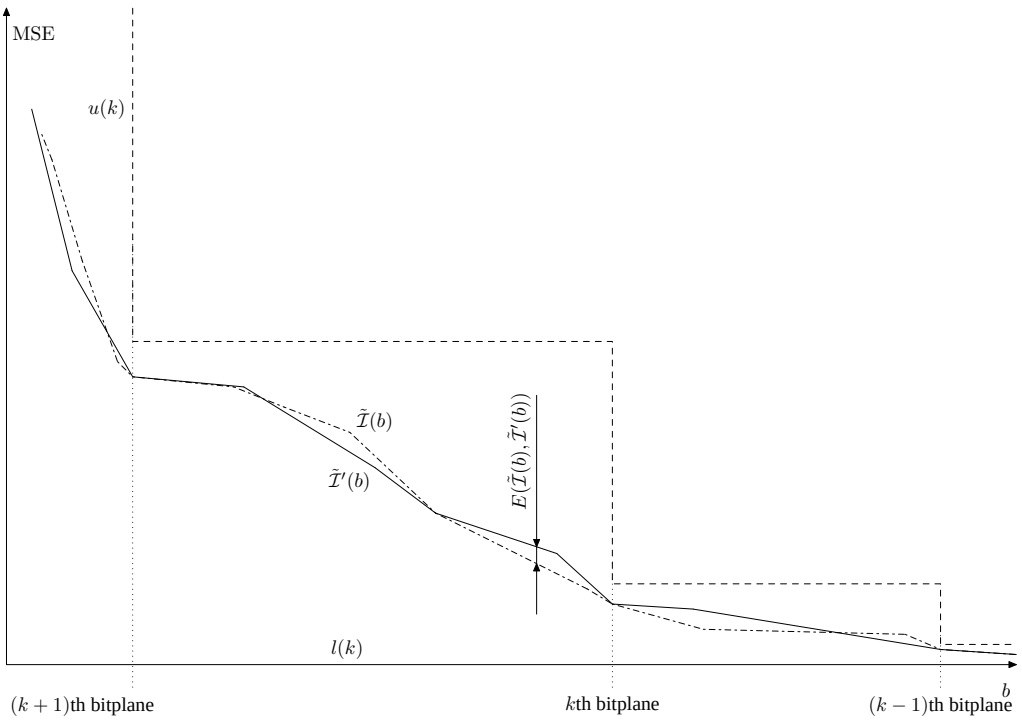


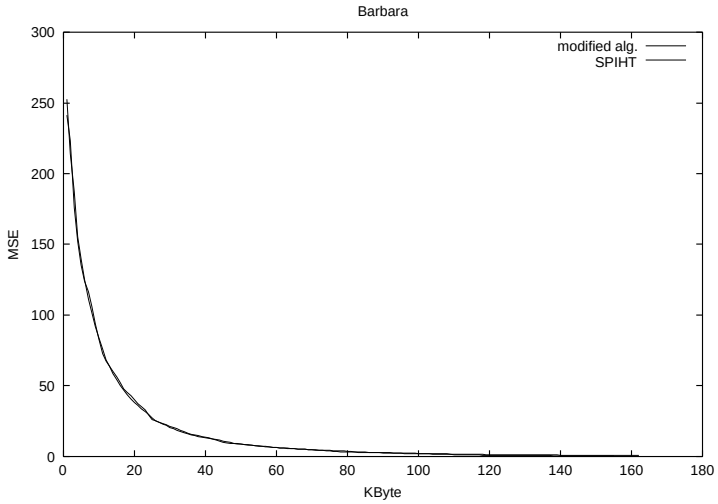
Figure 5.8: comparison of the mean squared error, $l(b)$ and $u(b)$ are upper and lower bounds for $E(\tilde{\mathcal{I}}(b), \tilde{\mathcal{I}}'(b))$

First, we have to compare the original and the reconstructed image after a complete bit plane is coded. Because the same information is produced by both algorithms (even though in different order),

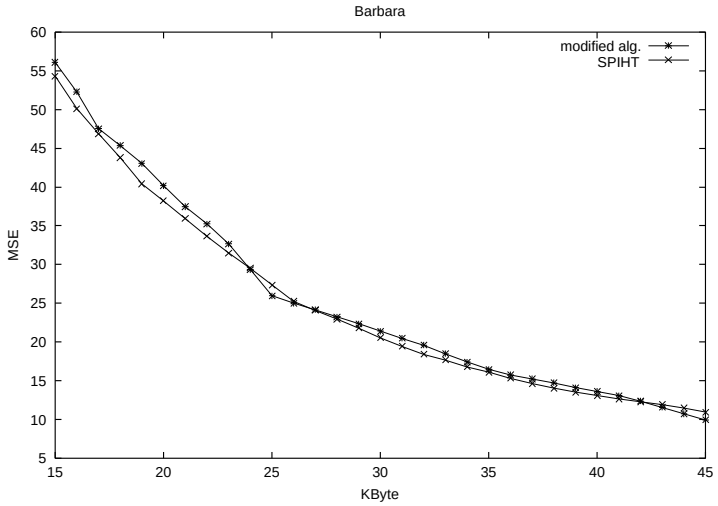
$$E(\tilde{\mathcal{I}}(b), \tilde{\mathcal{I}}'(b)) \quad = \quad 0$$

holds.

The more interesting case is the difference of the two algorithms during the coding of a bit plane. The maximum difference during the coding of the kth bit plane can be estimated with lower and upper bounds $l(k)$ and $u(k)$. Obviously, the lower bound $l(k)$ is zero. This is a tight lower bound. In his master thesis Fey [Fey01] proves $u(k) = 4^k$. Note that this is a rough estimate because it is independent of the image under consideration. For this reason, we have measured the differences on a set of known benchmark images. However, it is very difficult to distinguish the reconstructed images with the naked eye. To circumstantiate this remark, we have applied both algorithms on *barbara*. We have measured the mean squares error after each kbyte of compressed output. Figure 5.9 shows the outcome of the experiment, which confirms our informal observation. The mean squared errors are nearly equal.



(a) complete graph

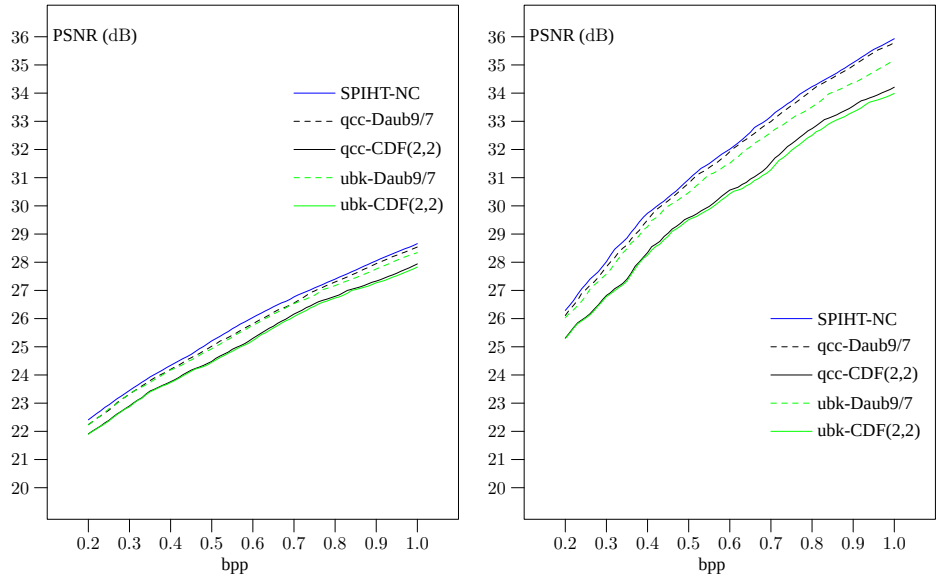


(b) detail of Figure 5.9(a)

Figure 5.9: comparison of the MSE, image *barbara*, $N = 512$, $\mathrm{dpth} = 8$

The comparison is done using our software implementation (named *ubk*) of the *SPIHT* codec. That is, we

compare the compression result between our traditional and our modified *SPIHT* encoder. The original algorithm of Said and Pearlman is available as binary version for educational purposes only. They use the Daubechies 9/7 wavelet and perform always five level of wavelet decomposition.



(a) baboon, $N = 512$, $\mathrm{dpth} = 8$

(b) barbara, $N = 512$, $\mathrm{dpth} = 8$

Figure 5.10: comparison of the compression efficiency between the binary version of the original *SPIHT* algorithm, the public available QccPack version, and our software implementation, five levels of wavelet transform are performed

Therefore we compared our implementation with the original *SPIHT* and the free available version called QccPack [Fow02]. QccPack provides an open-source collection of library routines and utility programs for quantization, compression, and coding of data. In Figure 5.10 the compression results are shown for the images *baboon* and *barbara*. As long as we use the same wavelet, we achieve nearly the same compression efficiency as the original *SPIHT* and are almost as good as the QccPack version. Using the CDF(2,2) wavelet reduces the PSNR values in our implementation as well as in the QccPack version.

Chapter 6

FPGA architectures

In this chapter we present several architectures and complete designs which are all based on the developed partitioned approach for wavelet-based image compression for programmable hardware. All designs were specified using the hardware description language VHDL. We turned our attention to rapid prototyping without heavy manual optimizations after synthesis. As a consequence the experimental results mainly show the mightiness of the partitioned approach[1].
Each design was developed to run at our prototyping environment. This environment consists of a PC equipped with a PCI card where a Xilinx XC4000XLA FPGA is mounted. The advantages and disadvantages of this prototyping environment are explained in Section 6.1.
All designs were described in VHDL language. As synthesis tool we decided to use the FPGA compiler from Synopsys. The resulting netlists in EDIF format are taken as input to the Xilinx implementation flow (mapping, place and route, timing analysis and bit stream generation) to generate a configuration bit stream. We dispensed with manually optimizations like floor planning. These bitstreams are further processed with an utility called *Hap-Generator* from the vendor of the prototyping environment. The resulting files are used to configure the FPGA using the provided software interface over the PCI bus.
In Section 6.2 we present a complete design for partitioned two dimensional discrete wavelet transform for lossless compression capable of running at 40MHz at our prototyping environment. Then we show several architectures to compute a partitioned wavelet transform for lossy compression in programmable hardware. These architectures can be combined with the design for lossy compression, which is shown in detail in Section 6.3. This design implements the well known *SPIHT* algorithm of Said and Pearlman adapted to our partitioned approach. Furthermore an arithmetic encoder can be included in the architecture to further improve the compression rates.

6.1 Prototyping Environment

In order to obtain experimental results on real world data we have used a PC equipped with a FPGA PCI board as prototyping platform. This PC is running under WinNT® and Linux operating system. The FPGA prototyping board was manufactured by Silicon Software GmbH, Mannheim, Germany [Gmb99]. It is equipped with a Xilinx device of the XC4000XLA series. The device has the speed index -09 and is mounted on the PCI board in a HQ240 package. The data path diagram of the board is shown in Figure 6.1.
Beside the Xilinx device the PCI card is equipped with 2Mbyte SRAM which is connected directly to the FPGA with 36 data lines. This memory bank can be accessed with at most 80MHz frequency. Furthermore there is a programmable clock generator and a device to manage the communication with the PCI bus.
The prototyping environment provides several mechanism to exchange data between the mounted Xilinx device, the local SRAM and the PC main memory. These could be categorized into

1. direct access to registers/latches (configured into the FPGA)

[1] Stephan Sutter and Görschwin Fey, students of mine, assist me to implement, simulate, synthesize and validate the VHDL designs in the scope of their master theses [Sut99], [Fey01].

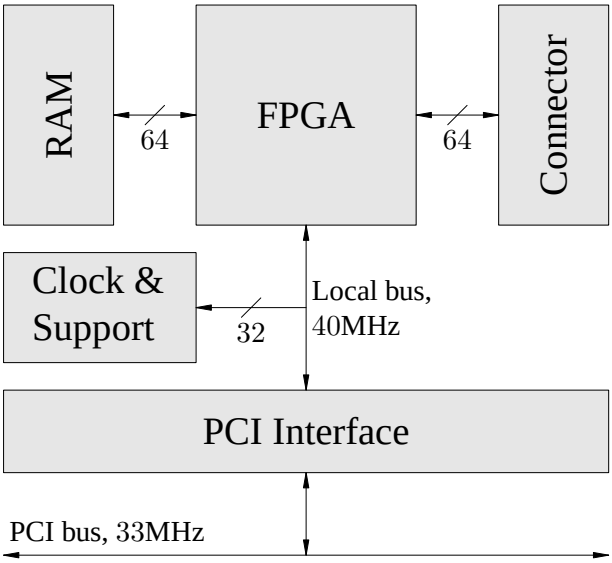


Figure 6.1: Data path diagram of the FPGA prototyping board

2. access to memory cells of the local SRAM (all data transfers are going through the FPGA),
3. DMA transfers and DMA on demand transfers,
4. and interrupts.

To communicate with the PCI card one has to write a simple C/C++ program, which initializes the card, configures the FPGA, set the clock rate and starts the data transfer or the computation. See Appendix A.1 for an example. In our applications the images are transfered to the FPGA or the local SRAM. Then the compression algorithms configured in the FPGA is started by setting a FPGA status register. While compressing, the specific codec takes over the control of the data transfer at all using the DMA on demand and interrupt features. The local bus on the PCI card can be clocked with a frequency of at most 40 MHz, whereas the FPGA design itself can be clocked with at most 80MHz. We tried to achieve a clock rate of 40MHz, which is a strength constraint if internal memory is configured into the Xilinx device.
The FPGA itself can be configured with a circuit of your choice. Due to the fixed data path on the PCI card one has to consider some limitations. Each design has to be wrapped with a so called *top level module*. This module (there are 9 different modules each with different complexity) provides the following VHDL-interface

```
entity extDesign is
        address     : in  std_logic_vector( 21 downto 0 );
        dataFromPci : in  std_logic_vector( 31 downto 0 );
        dataToPci   : out std_logic_vector( 31 downto 0 );
        rdRdy1      : in  std_logic;
        wrRdy1      : in  std_logic;
        rdRdy2      : in  std_logic;
        wrRdy2      : in  std_logic;
        rdTransfer2 : in  std_logic;

        interfaceClk   : in  std_logic;
        designClk      : in  std_logic;
        designClk2     : in  std_logic;
        cmcClk         : in  std_logic;

        interruptClk    : out std_logic;
```

```
        interruptSignal : out std_logic;
        interruptStatus : in std_logic

      );

end extDesign;
```

To communicate with both the PC main memory and the local SRAM one has to multiplex the busses `dataFromPci` and `dataToPci`. This decreases the routability of the design in a significant manner.
Another disadvantage with respect to our application are the fixed alignments of the busses to the primary ports of the device at north and south side. Since arithmetic components like adder should be aligned vertical due to the fast carry chain in XC4000 devices data ideally flows in horizontal direction. This gives a contradiction to the port alignments at the north and south side.
The prototyping environment had influence to our designs, too. Using huge internal RAM modules leads to a substantial drawback using devices without BlockRAM®like the devices of the XC4000XLA series. The RAM has to be configured into the CLBs. Each CLB can represent a 16x1 bit or a 32x1 bit RAM block with 4 or 5 address lines, respectively. These address lines have to be connected to each CLB of the corresponding RAM module. As a consequence huge fanout is introduced. This is problematical in terms of routability and overall performance constraints. Newer devices provide special components like RAM cells and arithmetical units in the FPGA. Since there was no budget to buy a second prototyping environment, we can only provide experimental result based on the given μEnable product.

6.1.1 The Xilinx XC4085 XLA device

The Xilinx XC4085 device consists of a matrix of *Configurable Logic Blocks* (CLB) with 56 rows and 56 columns. These CLBs are SRAM based and can be configured many times. After shut down the power supply they have to be reconfigured. The CLBs could be interconnected with horizontal and vertical signals.
Figure 6.2 shows the schematic of one CLB [Xil97]. It can be configured to represent any function with 4 or 5 inputs (see Figure 6.2: function generator F, G, or H). Furthermore there are mainly two flip-flops, two tristate drivers and the so called *carry-logic* available. The routing is done using *programmable switch matrices* (PSM), see Figure 6.3(b). Remark that each switch corresponds to a multiplexer with a specific delay. Each CLB can be configured to provide internal RAM. In order to do this, conceive the four input signals to F and G as address lines. Thus a CLB provides two 16×1 or one 32×1 random access memory module, respectively. At maximum we have 12kbyte RAM available. Each RAM block can be configured with different behavior:

- synchronous RAM: edge-triggered
- asynchronous RAM: level-sensitive
- single-port-RAM
- dual-port-RAM

The provided tristate drivers are somewhat counter productive in our specific application. In fact, they could be connected to horizontal (especially long) lines only. In contrast, the wide address and data busses, where switching between different signal source (the typical application field of tristate drivers) is necessary, have to be routed in vertical direction, due to the generated modules from the Xilinx tools *Logiblox*®or *Core-generator*®.
Using multiplexer instead of tristate drivers reduces the time for place and route dramatically or makes it even possible to place and route the design at all.
Arithmetic operations are supported by the dedicated *carry logic* in the XC4000 family [New96]. Each CLB contains hard-wired carry logic to accelerate arithmetic operations. Fast adders can be constructed as simple ripple carry adders, using the special carry logic to calculate and propagate the carry between full adders in adjacent CLBs. One CLB is capable of realizing two full adders.

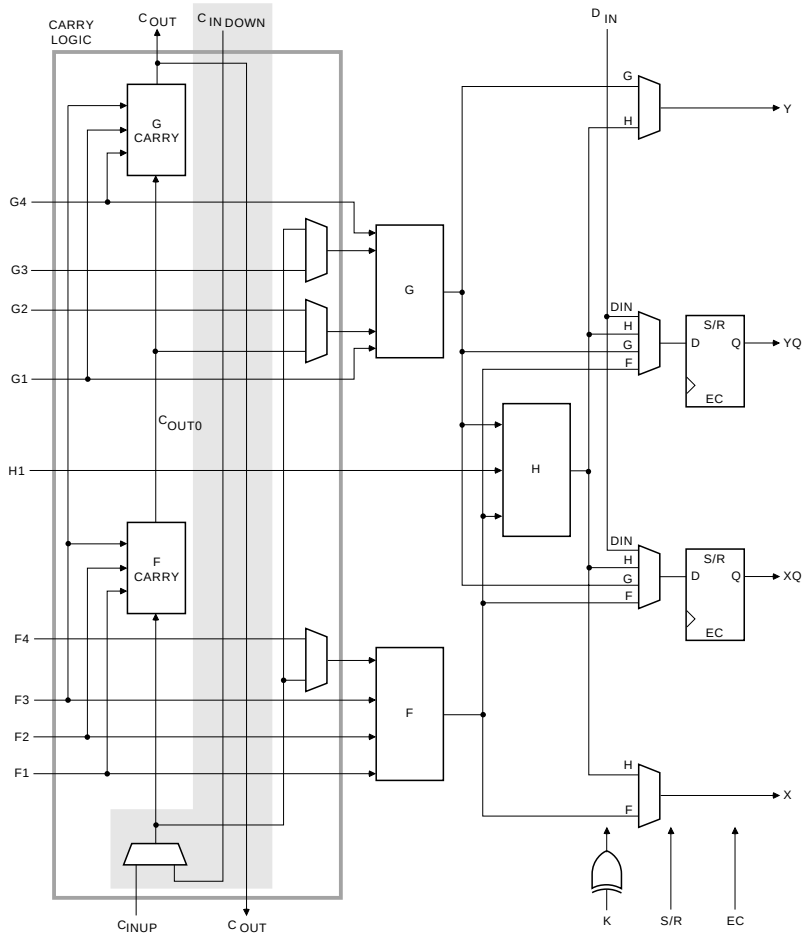
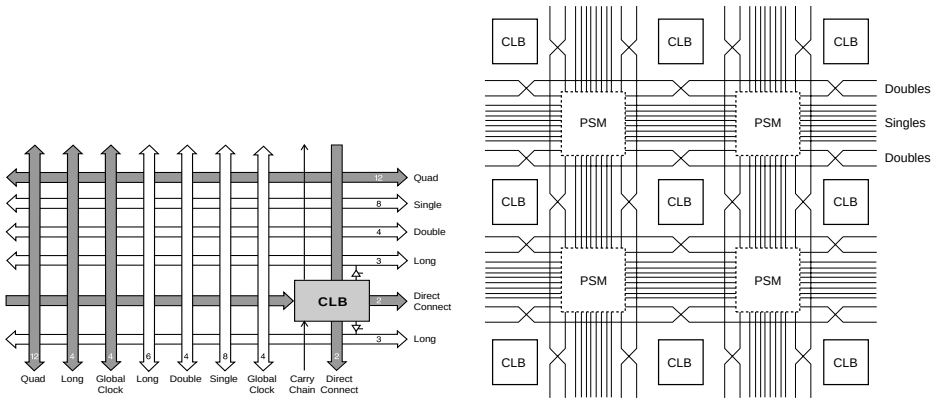


Figure 6.2: simplified schematic of XC4000 CLBs

6.2 2D-DWT FPGA Architectures targeting Lossless Compression

To show the computational power of the partitioned approach for discrete wavelet transforms, we have firstly implemented circuits for the two dimensional DWT in case of lossless compression. As already mentioned in Section 5.2.1 we do not have to consider boundary effects and could use modular arithmetic in combination with the lifting scheme. We have implemented several VHDL designs. All designs can wavelet transform greyscale images (wavelet decomposition) and make the inverse transform (wavelet reconstruction), too. The architectures differs in the number of one dimensional Lifting units and the way of transferring data between the prototyping board and the PC. In the following all available architectures are listed.

- one Lifting unit
  - data transfer from the PC directly to the internal memory and vice versa
  - data transfer from the PC to the SRAM on the prototyping board vice versa
- four Lifting units working in parallel
  - data transfer from the PC to internal memory and vice versa
  - data transfer from the PC to SRAM on the prototyping board vice versa



(a) interconnections between the CLBs

(b) switch matrices in XC4085XLA

Figure 6.3: XC4000 device, routing resources

The implementations with four Lifting units introduce parallelism with respect to the rows of a subimage. Once the FPGA is configured, a whole image can be transfered to the SRAM of the PCI card (maximum size 2Mbyte). Each subimage is then loaded into the internal memory, wavelet transformed, and the result is stored in the SRAM where it can be transfered back to the PC.
Another choice is to write a subimage directly into the internal memory of the FPGA, start the transform, and read the wavelet transformed subimage back to the PC.
The computation itself is started if an internal status register is set by the software interface. The FPGA causes an interrupt, if the computation has finished.
In Figure 6.4 you can see the global data path diagram of a circuit with four parallel working lifting units. To distinguish between data and address signals they are labeled with D and A, respectively. As you can observed, there are wide busses necessary for communications between the RAM modules and the Lifting units. The RAM itself is further decomposed in smaller units to support the parallel execution of the four Lifting units. Its is capable to store a subimage of size 32×32. There exists two separate finite state machines to control the wavelet transform and its inverse, respectively.
We achieved clock rates in an order of magnitude of 40 MHz which result in a computation time for wavelet transformation of a 512×512 image of less than 20 milliseconds. Most of the running time is due to huge routing delays caused by the relative large internal memory blocks in the XC4085XLA.

6.3 2D-DWT FPGA Architectures targeting Lossy Compression

As mentioned in Chapter 5 we will distinguish between two different architectures for the partitioned discrete wavelet transform on images. The first one is based on *divide and conquer* techniques. The second one is based on *pipelining*.

6.3.1 2D-DWT FPGA Architecture based on Divide and Conquer Technique

Recall from Section 5.2.3 that in order to compute one level of CDF(2,2)-wavelet transform in one dimension we need two pixels to the left and one pixel to the right of each rows of an image, respectively.
Thus, we append a row of an image by one and two memory cells on the right and on the left, respectively. We called such a enlarged row *extended row*.
Let r be a row of an image of length 16. Now, the problem is split in two subproblems which are solved interlocked. The first module computes the 'inner' coefficients $c_{0,0}, \ldots, c_{0,15}$ and $d_{0,0}, \ldots, d_{0,15}$ which

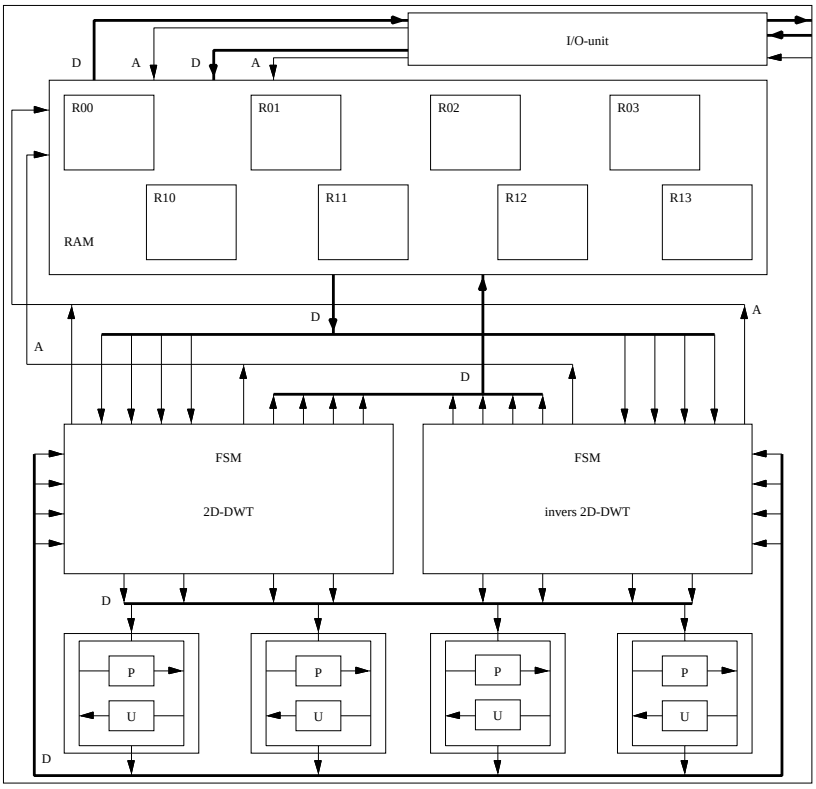


Figure 6.4: schematic dataflow of the 2D-DWT architecture

only depend on the extended row. In order to proceed to the next level of the wavelet transform, the coefficients at appended positions have to be made topical, i.e., the coefficients $c_{0,-2}$, $c_{0,-1}$, and $c_{0,16}$ have to be computed. The first module which is responsible for the inner coefficients computes the subtotals of $c_{0,-2}$, $c_{0,-1}$, and $c_{0,16}$ that only depend on the extended row. The second module computes the subtotals of $c_{0,-2}$, $c_{0,-1}$, and $c_{0,16}$ that depend on pixels not in row. Adding up corresponding subtotals results in the wavelet coefficients needed in the next level of the wavelet transform. As example, let us consider coefficient $c_{0,16}$. The module which is responsible for the inner coefficients computes the subtotal $-\frac{1}{8}r_{30} + \frac{1}{4}r_{31} + \frac{3}{4}r_{32}$. The other module computes $\frac{1}{4}r_{33} - \frac{1}{8}r_{34}$ in parallel. Adding up these two subtotals gives the coefficient $c_{0,16}$ which has to be stored in the right appended memory position. Note that the second module loads the external pixels needed one by one and processes them without storing them all the time. Only four temporal memory positions have to be stored in addition in order to perform four levels of transform. The generalization to two dimensions which is needed for wavelet transform on images goes straightforward as the same technique can be applied to the columns of the image, too.
The drawback of this technique is that random access to the external memory is basically and that we have to provide a complex control unit to synchronize the parallel memory accesses to the external SRAM. We have decided to use the second architecture based on pipelining in our VHDL implementation.

6.3.2 2D-DWT FPGA Pipelined Architecture

The proposed architecture in this section for the partitioned 2D-DWT mainly consists of two one dimensional DWT units (1D-DWT) for horizontal and vertical transforms, a control unit realized as a finite state machine, and an internal memory block. For illustration see Figure 6.5. To process a subimage, all rows are transfered to the FPGA over the PCI bus and transformed on the fly in the horizontal 1D-DWT unit using pipelining. The coefficients computed in this way are stored in internal memory of different types. The coefficients corresponding to the rows of the subimage itself are stored in single port RAM. Now the

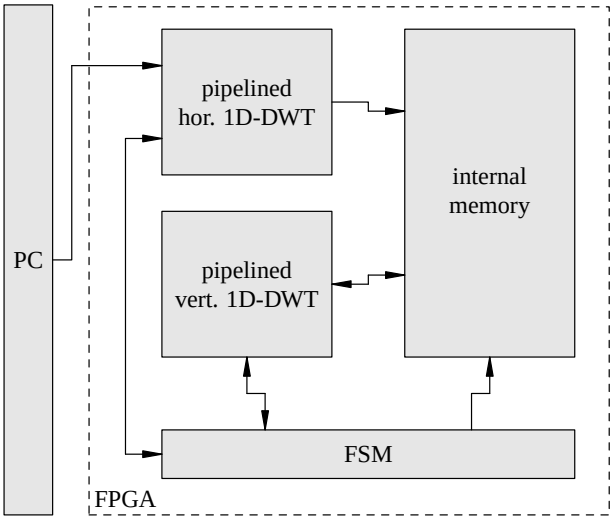


Figure 6.5: block diagram of the proposed architecture

vertical transform levels can take place. This is done by the vertical 1D-DWT unit. The control unit coordinates these steps in order to process a whole subimage and is responsible for generating enable signals, address lines, and so on.
At the end, the wavelet transformed subimage is available in the internal RAM. At this point an EZW-algorithm can be applied to the multiscale representation of the subimage. Since all necessary boundary information was included in the computation, no block artefacts are introduced by the following quantization.
The following sections describe the horizontal 1D-DWT unit, in detail.

Horizontal DWT unit

As opposed to the conventional 2D-DWT where horizontal and vertical 1D-DWT alternate, we compute first all four horizontal transform levels as described in Section 3.2. This allows a pipelined approach because the intermediate results do not have to be transposed. The whole horizontal transform is done for the 16 rows of the subimage under consideration. In addition, 30 rows of the neighboring subimage in the north and 15 rows of the southern subimage are transformed in the same manner. These additional computations are required by the vertical DWT applied next.
This unit has to take four pixels of a row at each clock cycle and must perform 4 levels of horizontal transforms. Figure 6.6 illustrates that the unit consists of four pipelined stages, one for each transform level.
The data throughput is mainly dominated by the first stage, since the number of coefficients is down sampled to half after each level of transform. Therefore the first and second level are performed by a module with higher throughput than the following two levels. The first and second stage outputs two low and two high frequency coefficients at one clock cycle, respectively. After the first stage, the two high frequency coefficients (the ones at even and odd positions) are merged together to output a continuous stream of $d_{0,j}$. The two low frequency coefficients are merged together in the same manner. Furthermore they are combined into a four pixels wide bus as input of the second stage. This is done in the module named *2to4* by a simple delay element.
The second stage operates similar to the first one but at only half speed. The computed low frequency coefficients of level two are merged together into a single data stream. This is the input for the third stage. The high frequency coefficients are merged in the same way and are shifted out.
The transform units of stage three and four (named *1i2o* in Figure 6.6) take only one coefficient of the previous level as input and alternately outputs a low or a high frequency coefficient at one clock cycle. The

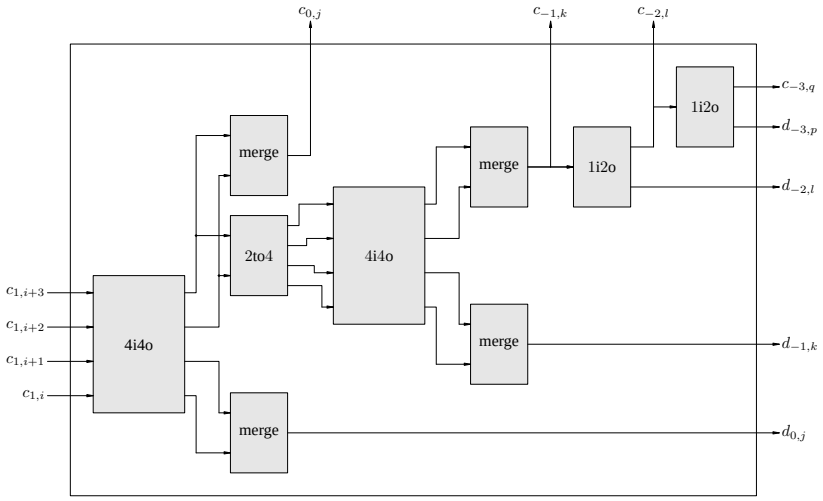


Figure 6.6: 4 level horizontal 1D-DWT unit

horizontal 1D-DWT unit processes a pixel row of length 16 in only 32 input clock cycles including the boundary pixel.
In the following two subsections we specify the modules *1i2o* and *4i4o* used in the different transform levels.

The w-bit input 1D-DWT unit (1i2o)

To implement one level of DWT using the lifting method (see Chapter 5) the following steps are necessary:

- split the input into coefficients at odd and even positions,
- perform a *predict-step*, that is the operation given in Equation (1.13),
- perform an *update-step*, that is the operation given in Equation (1.14).

An efficient realization of the last two steps is given in Figure 6.7 and Figure 6.8. The computation is split

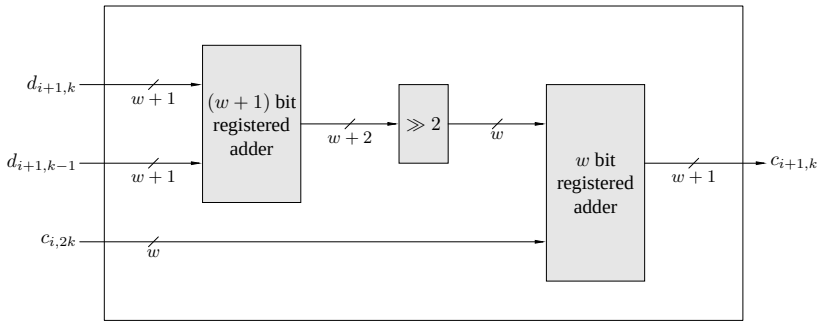


Figure 6.7: predict unit

into the elementary pieces, which are additions, subtractions, and shifts. With the given arrangement, the combinational functions between two flip-flops fits into one column of CLBs. Furthermore the registered adder/subtracters generated by Logiblox are configured such that the dedicated carry logic in the XC4000 series is used.

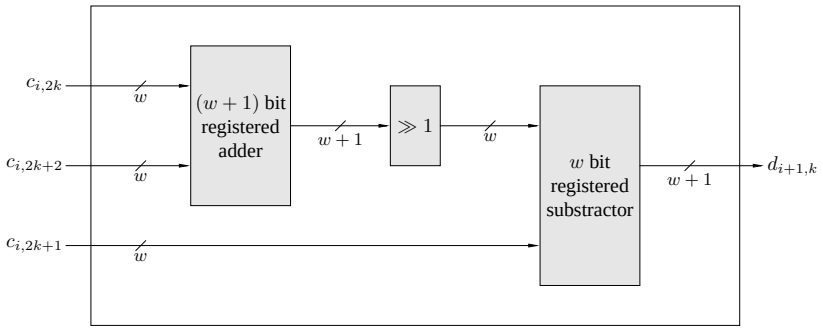


Figure 6.8: update unit

The whole w-bit 1D-DWT unit is constructed accordingly to the lifting scheme [Swe96]. Figure 6.9 sketches the architecture. The unit consists of two register chains. The registers in the upper chain are enabled at even, the registers in the lower chain at odd clock edges. This splits the input into words at even and odd positions. Now the predict and update steps can be applied straightforward.

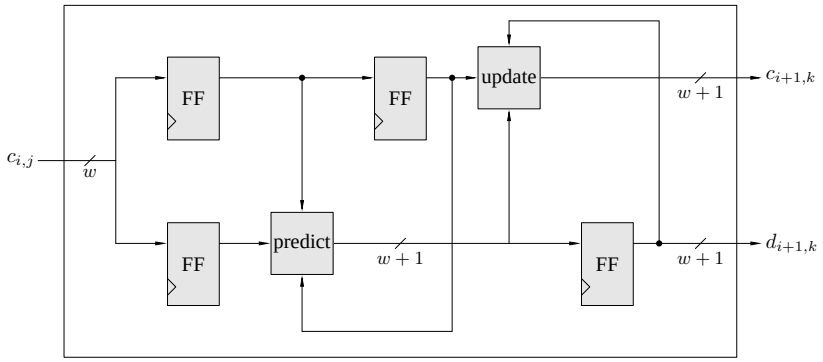


Figure 6.9: w-bit input 1D-DWT unit

The $4w$-bit input 1D-DWT unit (4i4o)

In order to perform a faster transform, which is needed during the first and second level, the w-bit input 1D-DWT has to be parallelized. One approach is to process four rows in parallel but we have to take into account the growing chip area (factor 4). Another disadvantage is the fact, that we have to split the RAM into four slices, where each slice corresponds to a w-bit input 1D-DWT. This results in additional data and address lines and slows down the access time to the RAM. As a direct consequence of this, the order of the data transfered to the FPGA has to be adapted accordingly.
To avoid the disadvantages just mentioned we have implemented a filter unit as shown in Figure 6.10, which takes four pixels of the *same* row at a time. Instead of 12 w-bit and 12 $w+1$-bit flip-flops, 4 predict and update units we only need 4 w-bit and 5 $w+1$-bit flip-flops and 2 predict and update components, if the bitwidth of the input coefficients/pixels is w.
We use this unit for both the first and the second level of the transform.

Internal Coefficient Memory

The internal memory for the wavelet coefficients is capable to store 16×16 coefficients. Since the bitwidth of the coefficients differs with their corresponding subbands the memory block consists of 5 slices. In

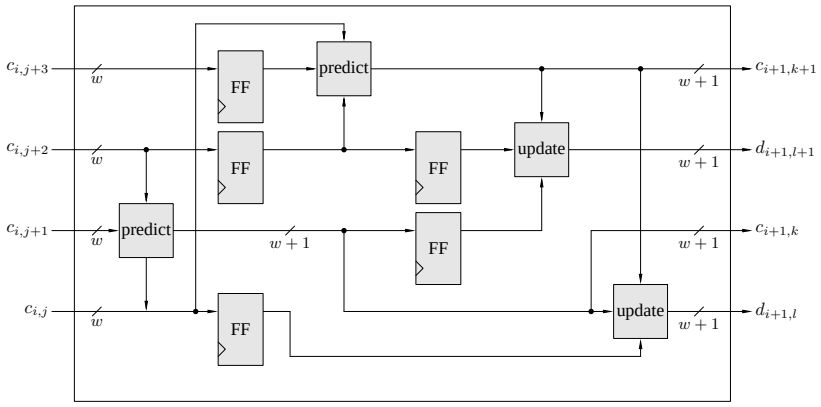
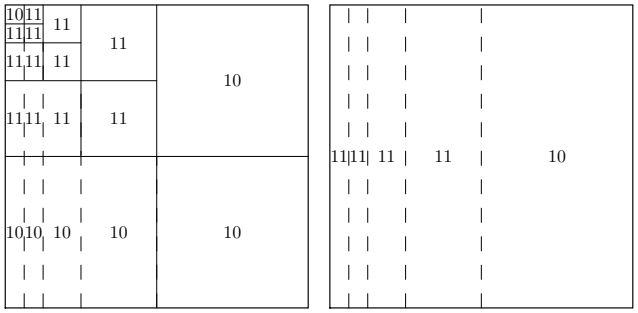


Figure 6.10: $4w$-bit input 1D-DWT unit

Figure 6.11(a) we have shown once again the minimal affordable bitwidth for each subbands as deduced in Section 3.2. The structure of the internal memory is illustrated in Figure 6.11(b).



(a) minimal affordable memory bitwidth

(b) slices of memory

Figure 6.11: structure of the internal memory to store the wavelet coefficients

6.4 FPGA-Implementation of the Modified *SPIHT* Encoder

In this section we give a brief overview to our VHDL implementation of the modified *SPIHT* compressor proposed in Section 5.3. In comparison to the algorithm *SPIHT Image compression without Lists* of Wheeler and Pearlman [WP00] we could reduce the internal memory from $(4+\frac{5}{16}d')N^2$ to

$$\left(\frac{37}{16}+\frac{5}{16}\log_2(d'+1)\right)N^2$$

bits ($N=512$, $d'=11$), which is required in addition to the memory for the wavelet coefficients. The amount of internal memory needed can be deduced from Figure 6.12. There are four RAM blocks besides the memory to store the wavelet coefficients. The modules named LP $:=$ LIP $\bigcup$ LSP and LIS represents the three lists LIP, LSP, and LIS, respectively. The modules named SL and SD store the precomputed significance attributes for all thresholds. Table 6.1 shows the size for each of these modules. It is remarkable, that in conjunction with the partitioned approach we need $(\frac{37}{16}+\frac{5}{16}\cdot 4)\cdot q^2 = 0.111$ kbytes only compared

Table 6.1: size in bits of each RAM block

LIS	$\left(\frac{N}{2}\right)^2 + \left(\frac{N}{4}\right)^2$	$\frac{37}{16}N^2$	$\left(\frac{37}{16} + \frac{5}{16}\log_2(d'+1)\right)N^2$
LP	$2N^2$		
SL	$\left(\frac{N}{4}\right)^2 \cdot \log_2(d'+1)$	$\frac{5}{16}N^2 \cdot \log_2(d'+1)$	
SD	$\left(\frac{N}{2}\right)^2 \cdot \log_2(d'+1)$		

to $(4 + \frac{5}{16} \cdot 11) \cdot N^2 = 238$ kbyte of the algorithm *SPIHT Image compression without Lists* on the whole image ($N = 512$, $d' = 11$, $q = 16$).

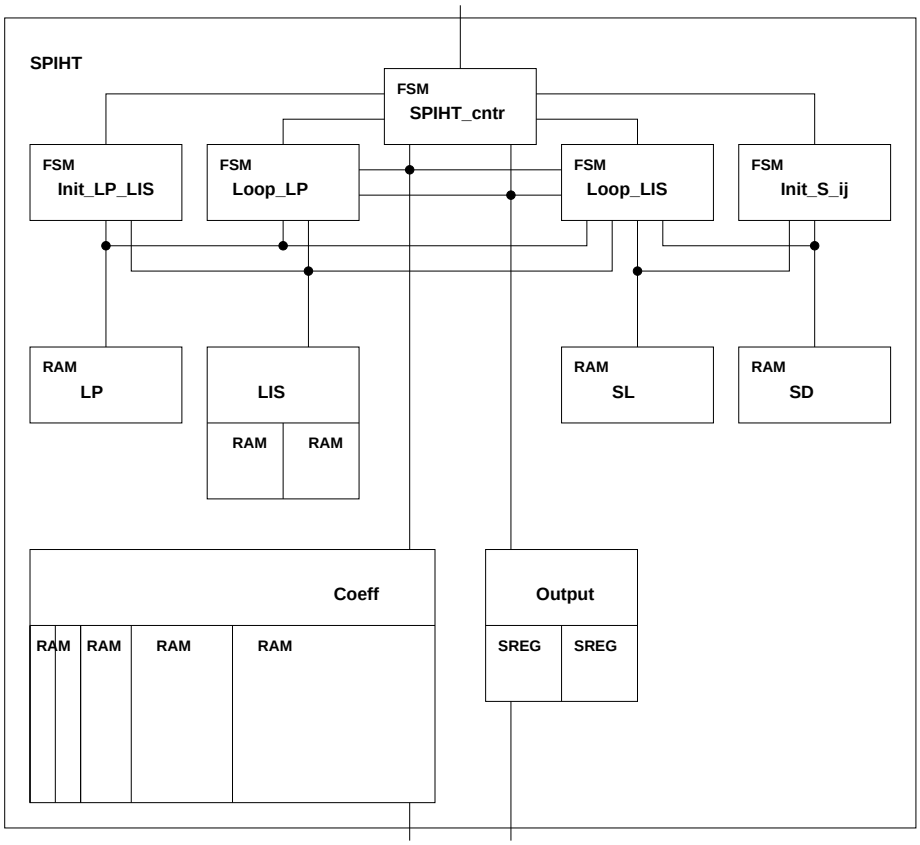


Figure 6.12: block diagram of our modified *SPIHT* compressor

We shortly explain the overall functionality before we present some of the modules in detail. Figure 6.12 shows the block diagram of the whole circuit.
Each subimage of the wavelet transformed image is transfered once to the internal memory module named 'coeff' or is already stored there. At first, the initialization of the modules representing LIP, LSP, and LIS and the computation of the significances is done in parallel. The lists LIP and LSP are managed by the module 'LP', the bitmap of LIS by the module 'LIS'. The significances of sets are computed for all thresholds $th \leq k_{\max}$ at once and are stored in the modules named 'SL' and 'SD', respectively. Here we distinguish between the significances for the sets $\mathcal{L}$ and $\mathcal{D}$. Details of this computation are presented in Section 6.4.2. With this information the compression can be started with bit plane $k_{\max}$. Finite state machines control the overall procedure.

The data to be output is registered in module 'output' from which it is put to the local SRAM on the PCI card on a 32 bit wide data bus. Additionally, an arithmetic coder can be configured into that module. This further reduces the compression ratio (see Section 6.4.3).

6.4.1 Hardware Implementation of the Lists

To reduce the memory requirement for the list data structures in the worst case, we implement the lists as bitmaps. The bitmap of list L represents the *characteristic function* of L.

Definition 5 *Let L be a list of elements out of $\{0,\ldots,N-1\}\times\{0,\ldots,N-1\}$. A map $L\chi:\{0,\ldots,N-1\}\times\{0,\ldots,N-1\}\to\{0,1\}$ with*

$$L\chi(i,j)=\begin{cases}1 & \text{if } (i,j)\in L,\\ 0 & \text{else}\end{cases}.$$

is called characteristic function of the list L.

Note that by this approach we loose the ordering information within the lists LIP, LSP, and LIS. The effect of this reordering with respect to the decoded image was discussed in detail in Section 5.4. The bitmaps themselves are implemented using RAM modules.
We provide two synchronous RAM modules for the three lists. The lists LIP and LSP are realized in one RAM module, the list LIS in the other one. The RAM module which realizes LIP and LSP has a configuration of $q\times q$ entries of bit length 2 as for each pixel of the $q\times q$ subimage either $(i,j)\in$ LIP, $(i,j)\in$ LSP, or $(i,j)\notin$ LIP$\cup$LSP holds.
The second RAM module implements LIS. Since none of the coefficients in the subbands LH$^{(0)}$, HL$^{(0)}$, and HH$^{(0)}$ can be the root of a zerotree, we have to provide a bitmap of size $\left(\frac{q}{2}\right)^2$. Furthermore, in the area which corresponds to LL$^{(0)}$ we have to distinguish between zerotrees of type A and B. Inside this quadrant, only coefficients of the upper left subquadrant can be of type B, the other coefficients are always of type A. Consequently, only for the area which corresponds to LL$^{(1)}$ the type information has to be stored. This results in additional $\left(\frac{q}{4}\right)^2$ bits.
We decided to use these two RAM modules for the representation of the three lists due to the simple control mechanism. Of course, with logarithmic coding we could further reduce the RAM size. There are three possible states for each coordinate in the area of the subband LL$^{(0)}$, which can occur in the combinations given in Table 6.2. The combinations can be found by a detail analyse of the original algorithm. With

Table 6.2: possible configuration states of a coordinate (i,j)

(i,j)	$\notin$ (LIP$\cup$LSP)	$\in$ LIP	$\in$ LSP
$\notin$ LIS	(a)	(b)	(c)
$\in$ LIS, type A		(d)	(e)
$\in$ LIS, type B		(f)	(g)

logarithmic coding we would reduce the RAM size to

$$3\cdot\left(\frac{q}{2}\right)^2+2\cdot\frac{3}{4}q^2=\frac{9}{4}q^2.$$

For $q=16$ this results in a saving of 16 bit only in contrast to the representation without logarithmic coding. This corresponds to only one CLB in the FPGA.

6.4.2 Efficient Computation of Significances

In the following we will concentrate on the module which computes the significances. The significance of an individual coefficient is trivial to compute. Just select the k^{th} bit of $|c_{i,j}|$ in order to obtain $S_k(i,j)$.

This can be realized by using bit masks and a multiplexer. However, it is much more difficult to efficiently compute the significance of sets for all thresholds in parallel. Before we go into details, we have to introduce some notations.
We define $S^*(\mathcal{T})$ as

$$S^*(\mathcal{T}) := \begin{cases} \max\{k | S_k(\mathcal{T}) = 1\} + 1 & \text{if } \mathcal{T} \neq \emptyset \text{ and } \exists k : S_k(\mathcal{T}) = 1 \\ 0 & \text{else} \end{cases}$$

Thus, $S^*(\mathcal{T})$ stands for the maximum threshold k for which some coefficient in $\mathcal{T}$ becomes significant. Once $S^*(\mathcal{T})$ is computed for all sets $\mathcal{L}$ and $\mathcal{D}$, we have preprocessed the significances of sets for all thresholds. In order to do this, we use the two RAM modules SL and SD. They are organized as

$$\frac{N}{4} \times \frac{N}{4} \times 4 \text{ bit}$$

and

$$\frac{N}{2} \times \frac{N}{2} \times 4 \text{ bit}$$

memory, respectively.
The computation is done bottom up in the hierarchy defined by the spatial oriented trees. The entries of both RAMs are initialized with zero. Now, let (e, f) be a coordinate with $0 < e, f < q$ just handled by the bottom up process and let

$$(i, j) = \left(\left\lfloor \frac{e}{2} \right\rfloor, \left\lfloor \frac{f}{2} \right\rfloor \right)$$

be the parent of (e, f) if it exists. Then SD and SL have to be updated by the following process

```
foreach  0 < e, f < 8
    SD(e, j) := 0;
    SL(e, j) := 0;
foreach (e, f)
    if e < N/2  and  f < N/2  then
        SL(i, j) := max{SL(i, j), SD(e, f)}                    (6.1)
    SD(i, j) := max{SD(i, j), S*(e, f)}                        (6.2)
    if e < N/2  and  f < N/2  then
        SD(i, j) := max{SD(i, j), SD(e, f)}                    (6.3)
```

The initialization and the bottom up process are organized by the finite state machine shown in Figure 6.13. After reset we start the computation in state one and initialize $k'_{\max}$ and the row and column indices e and f. At this time $\mathrm{SD}(e, f)$ and $\mathrm{SL}(e, f)$ hold their old values from the last subimage under consideration for all $0 < e, f < q$. If the enable signal becomes active, we proceed in state 2. Here we buffer the present value of $\mathrm{SD}(e, f)$. In the states 3, 4, 5, and 6 we compute line (6.1). The condition

```
e, f  are odd
```

checks, if we visit a 2×2 coefficient block for the first time. The states 2, 5, and 6 are responsible for computing the maximum of $\mathrm{SD}(i, j)$ and $S^*(e, f)$ (line (6.2)), which is buffered in tS. State 8 performs the assignment in line (6.3). Furthermore, this finite state machine updates the value $k'_{\max} = k_{\max} + 1$ for the subimage under consideration. In state 10 the low frequency coefficient at position $(0, 0)$ will be included in this computation, too. The operation in state 9 is done using a simple subtractor and a combined representation with interleaved bitorder of the row and column index e and f, that is

$$f_{n-1}, e_{n-1}, f_{n-2}, e_{n-2}, \ldots, f_1, e_1, f_0, e_0.$$

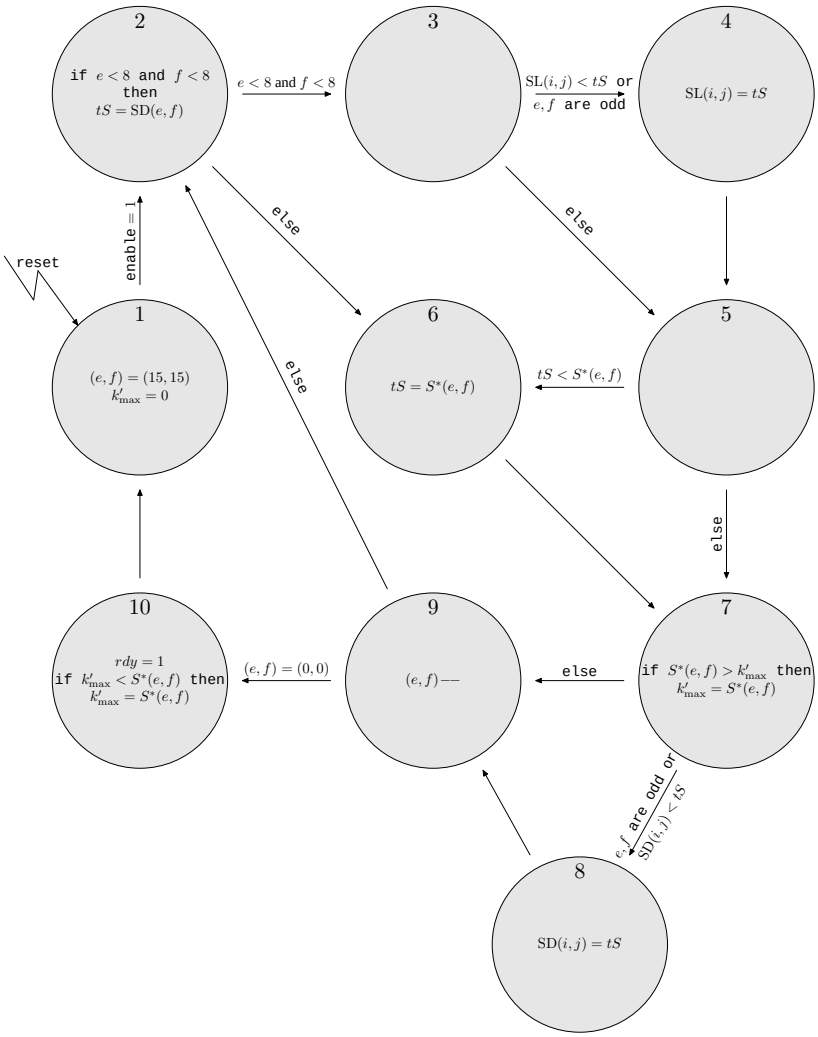


Figure 6.13: FSM to compute $S^*(\mathcal{L})$ and $S^*(\mathcal{D})$

6.4.3 Optional Arithmetic Coder

Our design can optionally be configured with an arithmetic coder that processes the data which is output by the modified *SPIHT* algorithm. We decided to utilize the coder of Feygin, Gulak, and Chow [FGC93]. It is very suitable for FPGA designs because it does not contain multiplication operations. The implementation is straightforward. It needs three clock cycles to code one bit of the modified *SPIHT* output. The overall structure of the arithmetic coder module is shown in Figure 6.14. The compressed stream of the modified *SPIHT* algorithm is first stored in a circular buffer of length 4. The signal $type$ determines the probability model. We have investigated the cumulative percentage of the different output positions in the algorithm in detail. Note that in our modified algorithm (see Figure 5.7) there are exactly eight lines (Line 0, 4, 6, 7, 12, 15, 16, 22) which output some data. For each of these output positions we provide one probability model and call them $\mathcal{A}_1, \ldots \mathcal{A}_8$. The model $\mathcal{A}_4$ for instance describes the properties of the sign output at Line 7. The module named 'carry chain' implements the so called *bit stuffing* for the potential carry overs in the arithmetic coder [RL81]. Furthermore, there exist signals to insert special markers and to stop the compression at the specified bit rate.

The most interesting part was the development of probability models adapted to the modified *SPIHT* compressor.

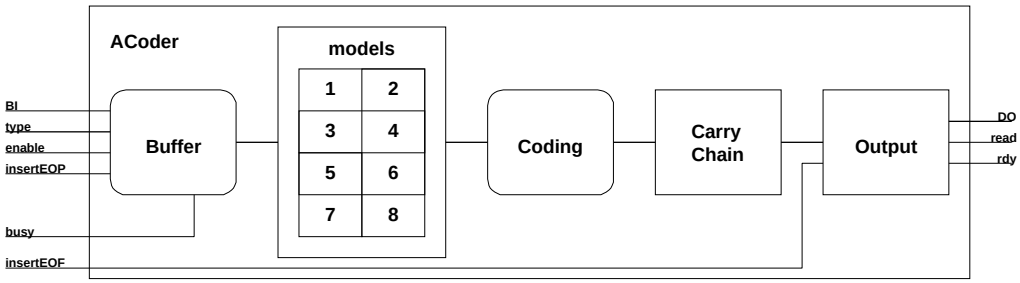


Figure 6.14: overall structure of the arithmetic coder

Probability models

In order to obtain meaningful models, we have counted the number of one's and zeros with respect to their output position while compressing our set of benchmarks images. Table 6.3 shows the cumulative percentage for Line 0 of the algorithm.

Table 6.3: cumulative percentage of $k_{\max}$

$k_{\max}$	cumulative percentage
0-3	0 %
4	1.9 %
5	11.1 %
6	26.7 %
7	42.7 %
8	17.1 %
9	0.4 %
10-15	0 %

The cumulative percentages can be expressed with conditional probabilities $p(0|w)$ with $w \in \{0,1\}^*$ It describes the probability that the next symbol is zero under the condition that w has already been read. We use a history of maximum size four. (Remember that, for subimages of size 16×16, $k_{\max}$ can be at most 11 because of the used wavelet and the number of transform steps.) To simplify matters, the probabilities were rounded to multiples of 5%. In this context, we certainly have to avoid the exceptional cases 0% and 100% because we work with cumulative percentages. Therefore, the values 1% and 99% have been introduced. As example, assume that $k_{\max}$ equals 6, i.e., Line 0 of the modified algorithm outputs the binary string 0110. We have to read four bits in order to obtain

$$\begin{aligned} k_{\max} &= k_3 k_2 k_1 k_0 \\ &= 0110, \end{aligned}$$

starting with the most significant bit k_3. The conditional probability that the first bit is zero is

$$p(0|\epsilon) = 0\% + 1.9\% + 11.1\% + 26.7\% + 42.7\% = 82.4\% \approx 80\%.$$

since in that case $k_{\max}$ has to be smaller than 8. Analogously, we have

$$\begin{aligned} p(0|0) &= \frac{0+0+0+0}{82.4} = 0\% \approx 1\% \\ p(0|01) &= \frac{1.9+11.1}{82.4} = 15.8\% \approx 15\% \\ p(0|011) &= \frac{26.7}{26.7+42.7} = 38.5\% \approx 40\% \end{aligned}$$

All the other probability models $\mathcal{A}_2, \ldots, \mathcal{A}_8$ are obtained in an analogous manner.
The VHDL implementation of these probability models has been specified by finite state machines, too.

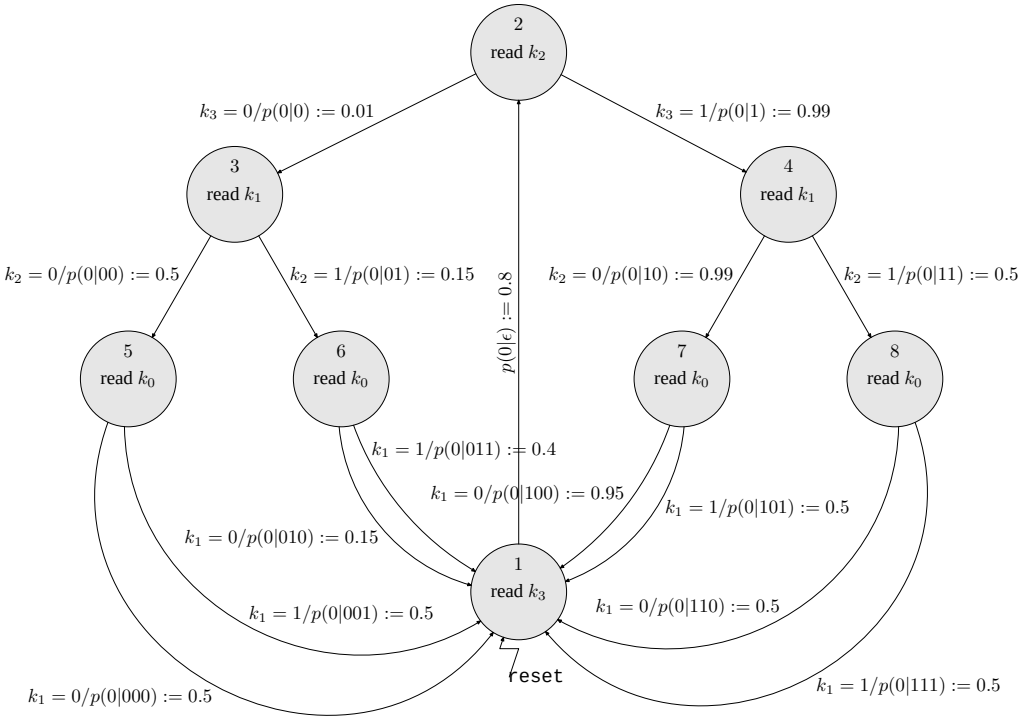


Figure 6.15: finite state machine to represent the model $\mathcal{A}_1$

Figure 6.15 shows the finite state machine of model $\mathcal{A}_1$. We will discuss the functionality of this finite state machine for the input $k_{\max} = 6$. The edges are labeled with pairs

condition / probability of zero to be output.

In each state we start reading one bit from the modified *SPIHT* output stream of Line 0. In state 1 we do not know anything about $k_{\max}$. Therefore the output of this finite state machine is $p(0|\epsilon) = 0.8$, labeled at the edge from state 1 to state 2. Now we know that $k_3 = 0$, start reading k_2, and follow the edge with the condition $k_3 = 0$ leading in state 3. Currently the probability to be output is $p(0|0) = 0.01$. In state 3 we start reading k_1 and have to follow the edge labeled with $k_2 = 1$. The output probability is now $p(0|01) = 0.15$. From state 6 we lead to state 1 and output the probability $p(0|011) = 0.4$. This completes the generation of the probabilities for the $k_{\max}$ of a subimage. Note, that for $k_{\max} < 4$ or $k_{\max} > 9$ (cumulative percentage is zero) we set the conditional probability of the bit k_1 and bit k_0 to 50%. Therefore, e.g. the edges from state 3 to 5 and state 5 to 1 are labeled with the probability 0.5.

Experimental results

We have implemented two versions of the modified *SPIHT* encoder based on the partitioned approach, one without and one with the arithmetic coder. Table 6.4 summerize the runtimes for several images. All compressions were done in a lossless manner. We have achieved clock rates of 40MHz for both implementations. Note that our VHDL designs were synthesized without manual optimizations. All basic memory and arithmetic modules were generated with Xilinx tools. Our implementations take 743 and 1425 logic blocks of the Xilinx XC4085XLA device, respectively.
In order to compare the software implementation with our presented FPGA design we measured the execution time of both. To obtain a faithful measurement of the hardware execution time we have included a

Table 6.4: execution times of hard- and software implementations in the case of lossless compression, 8 bit grey-scale images of size $N = 512$, partition size $q = 16$

image	*ubk-software*	FPGA	
		arithm. coder	
		without	with
baboon	2.3 s	204.0 ms	272.3 ms
barbara	2.0 s	203.1 ms	254.0 ms
goldhill	2.0 s	195.8 ms	246.9 ms
lena	1.9 s	192.5 ms	236.4 ms
peppers	1.9 s	196.7 ms	245.0 ms

counter (resolution 25ns) into the design. We could improve the compression time of $512 \times 512 \times 8$ bit greyscale images by a factor of 10 in comparison to an AMD 1GHz Athlon processor.
The effect of the arithmetic coder upon the compression ratio is shown in Table 6.5. The coder compresses the *SPIHT* output by further 2 to 4 percent. It is remarkable, that the compression ratio is always improved, in the lossless as well as in the lossy case.

Table 6.5: influence of the arithmetic coder

image	baboon	barbara	goldhill	lena	peppers
0.5 bpp	95.9	98.6	97.4	97.3	95.7
lossless	97.8	97.3	96.8	96.0	96.1

Chapter 7

Conclusions and Related Work

In this thesis we have presented a partitioned approach to wavelet based image compression and its application to image encoders using programmable hardware. We have developed an efficient FPGA architecture of the state of the art image codec *SPIHT*, which is comparable to the original software solution in terms of visual quality. We have achieved clock rates of 40MHz for our FPGA implementations. Note that our VHDL designs were synthesized without manual optimizations. All basic memory and arithmetic modules were generated with Xilinx tools. We could improve the compression time of $512 \times 512 \times 8$ bit greyscale images by a factor of 10 in comparison to an AMD 1GHz Athlon processor. The optional arithmetic coder compresses the modified *SPIHT* output by further 2 to 4 percent. It is remarkable, that the compression ratio is always improved, in the lossless as well as in the lossy case.
The main contribution is that we have proposed image encoder suitable for low cost programmable hardware devices with minimal internal memory requirements. This method outperform the recently published algorithm of Wheeler and Pearlman *SPIHT Image Compression without Lists*[WP00] with respect to the affordable memory.

In the following we will introduced the upcoming new JPEG2000 standard, which is also wavelet based, but is not premised on an embedded zerotree wavelet encoder. Afterwards, we will balance the advantages and disadvantages of our approach to the proposed method in the JPEG2000 standard.
It will become apparent, that there are some marked similarities to our partitioned approach. We emphasize that both developments were done independently of each other. Note, that until now to our knowledge no JPEG2000 encoder ASIC (application specific integrated circuit) is available. First intelligent property cores for JPEG2000 codecs are offered by Amphion Semiconductor Ltd. and inSilicon Corporation in 2002 [Amp02], [inS02].

7.1 *EBCOT* and JPEG2000

An algorithm named *EBCOT*, **E**mbedded **B**lock **C**oding with **O**ptimal **T**runcation presented by David Taubman [Tau00] was chosen as the central codec behind the new JPEG2000 Standard [TM02]. Mostly all of the desired features of the new still image compression standard are supported by the *EBCOT* algorithm. The most import ones are

- excellent visual quality of the reconstructed images even at lowest bit rates
- SNR scalability
- resolution scalability
- region of interest coding
- error resilience (noisy channels).

The algorithm itself is wavelet based, one of the basics to support the resolution scalability, which is an inherent feature of wavelet transforms. In contrast to compressors like *SPIHT* they partition the multiresolution scheme into so called *code blocks* of typical size 64 by 64 coefficients. These blocks are compressed independently of each other. The SNR scalability is accomplished by post compression rate distortion optimization and tricky bit stream organization. Errors encountered in code blocks have no influence to the other blocks. Therefore error resilience could be achieved in error prone environments. The region of interest coding could easily be adopted by weighting the distortion contribution of code blocks from inside the given region.
For the sake of completeness we give a short overview of the *EBCOT* algorithm. Since this is the central part of the JPEG2000 standard you will find all of the described techniques in available and upcoming codecs supporting JPEG2000.

7.1.1 The *EBCOT* Algorithm

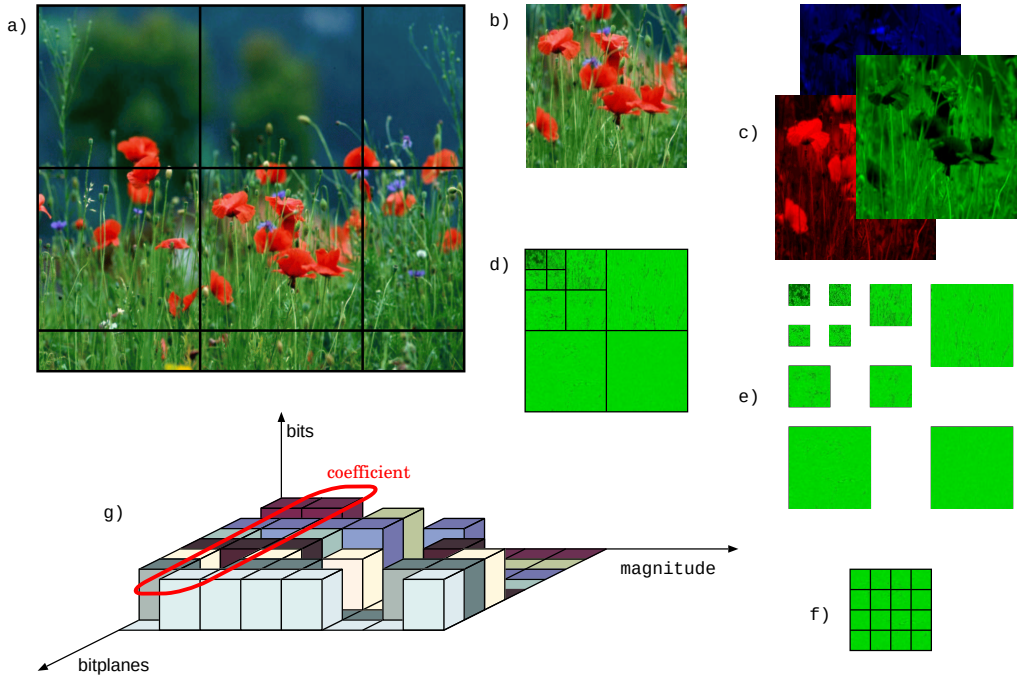


Figure 7.1: overview of the *EBCOT* algorithm

Figure 7.1 gives an overview of the stages of the *EBCOT* compressor. In the first stage the image is divided in *tiles* of manageable size, which is dependent on the type of the encoder. A software encoder on a modern computer can choose huge tiles compared to a hardware encoder implemented on a digital signal processor. Each of these *tiles* is now decomposed in color components which are coded independently of each other (cf. Figure 7.1c)).
At first a discrete wavelet transform is performed on each color component of the tiles. For lossy compression the Daubechies (9,7)-wavelet [Dau92] and for lossless compression the CDF(2,2)-wavelet has to be used, respectively (cf. Figure 7.1d)). Beside the usually wavelet decomposition known from *Mallat* [Mal89] other decompositions like wavelet packets are possible too [VK95].
Each subband (cf. Figure 7.1e)) of the given decomposition is now partitioned into code blocks of size 64 by 64 or 32 by 32 (cf. Figure 7.1f)). With each code block the information of the corresponding subband and the relative offset is stored. These code blocks are compressed in three passes. An adaptive arithmetic coder based on the *MQ-coder* [PM88] is used. The corresponding models are determined using a similar

signification attribute known from zerotree algorithms. As a result each code block is represented by a fine embedded bit stream (cf. Figure 7.1g)).

PCRD-opt

After each of the code blocks are compressed independent of each other a so called **P**ost **C**ompression **R**ate **D**istortion optimization method is applied. Simply spoken, they try to minimize the overall distortion for a given target bitrate. This can be seen as a linear optimization problem if some assumptions are made.
Let $D_i^{t_i^{r_i}}$ be the distortion contribution and $L_i^{t_i^{r_i}}$ the length of the truncated compressed bit stream of code block i at a truncation point $t_i^{r_i}$, respectively, where $0 \leq r_i < n_i$ if code block i has at most n_i truncation points assigned. If the distortion metric is additive, i.e.

$$D = \sum_i D_i^{t_i^{r_i}},$$

they have to solve the resource allocation problem

$$\textbf{minimize } D = \sum_i D_i^{t_i^{r_i}}$$
$$\textbf{subject to } \sum_i L_i^{t_i^{r_i}} <= L^{max}$$

Remark that this is an integer linear program, which means, that in general there is no set $\{t_i^{r_i}\}$ which yields to $\sum_i L_i^{t_i^{r_i}} = L^{max}$.
To reduce the overall solution space it is suggested to restrict the available truncation points to the ones on the convex hull of the rate distortion graph. For illustration see Figure 7.2. Each point corresponds to a pair $(L_k^{t_k^{r_k}}, D_k^{t_k^{r_k}})$. The longer a prefix of a code block, the smaller is the corresponding distortion and vice versa. As you can see, the only rate distortion pairs that make sense are the ones on the convex hull (red polygon). This scales down the search space significantly.

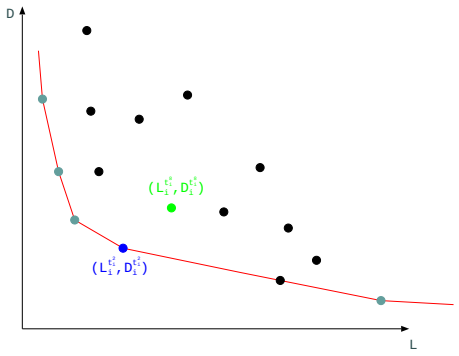


Figure 7.2: rate distortion curve and convex hull

After this optimization problem is solved, one has several possibilities to generate a bit stream.

SNR **scalability** To produce an embedded bitstream, that is a bitstream, where each prefix is a near optimal approximation of the original image in terms of the signal to noise ratio, the fine embedded code blocks are stringed together as illustrated in Figure 7.3. Thus, a prefix of some code block is located at the begin of the bitstream, if a lot of image energy is represented by this prefix. In the JPEG2000 standard this is done using so called *quality layers*. Each quality layer contains incremental contributions from various code blocks. With each received fraction of the overall bitstream the reconstructed image is successively enhanced.
This ordering is useful for image transfer over channels with narrow bandwidth. The decoder can present a preview at low visual quality in full resolution even if only few bytes are available.

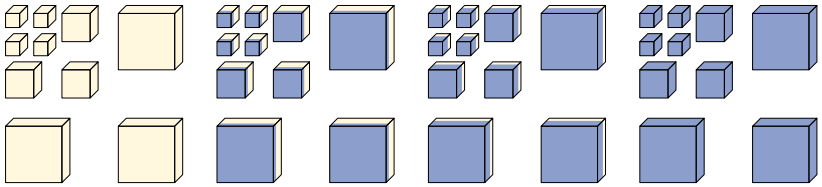


Figure 7.3: subbands after three level of wavelet transforms, in case of SNR scalability incremental code blocks contributions from all subbands ordered by there rate distortion ratio are used to construct a bitstream

Resolution scalability Another interesting question is, whether a small preview of a huge image is required only. Then it would be helpful, if the bitstream is ordered in a way, that the bytes at the beginning of the bitstream represent the original image at lower resolution but with excellent visual quality. The more bytes are available the larger resolution is possible. The multiresolution scheme after the wavelet transform had taken place gives the opportunity to support this resolution scalability.

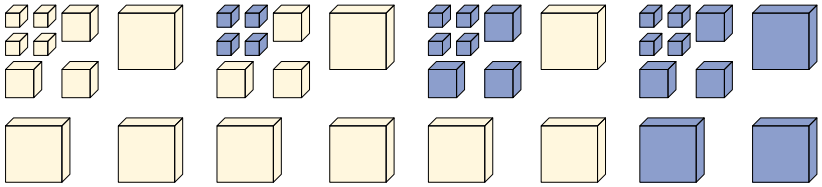


Figure 7.4: in case of resolution scalability code blocks contributions from those subbands are used that allow a reconstructed image of quarter size (second image), half size (third image), and full size (fourth image)

For an illustration see Figure 7.4. The first information in the bitstream is now composed from code blocks of all subbands of a transform level beginning with the last.

7.2 Similarities and Differences of JPEG2000 and our Approach

The *EBCOT* algorithm was chosen to be the central codec behind the new JPEG2000 standard because of the many supported features. Since we have adapted the embedded zerotree wavelet encoder *SPIHT* we will enumerate the advantages and disadvantages of this type of algorithm, included our adapted one, compared to *EBCOT*.

- The excellent visual quality of both codecs are undisputable. The SNR scalability is an inherited feature of codecs which produce embedded bitstreams. Therefore this property is provided by both methods.
- Resolution scalability can not be made available by the original *SPIHT* algorithm. Unfortunately, the partitioned approach does not help to support this feature, too.
- Region of interest coding can be provided by the *SPIHT* codec. The region of interest has to be translated into the wavelet domain and can then be represented by appropriate weighting of the coefficients in that areas. This is strongly simplified due to the partitioned approach, since related techniques as in the *EBCOT* paradigm can be used.
- Error resilience is not feasible by the original *SPIHT* method. Suppose some of the signification bits are toggled in a noise channel, then the decoder can not duplicate the execution path of the encoder anymore. There are no synchronization points in the bitstream available. Thus, even a simple bit

fault can distort the whole decompression procedure. In the case, that bit faults are introduced in a bitstream generated using the modified *SPIHT* encoder based on the partitioned approach, the impact of such an incident is limited to the corresponding subimage. This is similar to the error handling of the *EBCOT* algorithm.

- A feature, that is not provided by the *EBCOT* compressor, is taking advantage of the self similarities between the different subbands in each orientation. The partitioning into code blocks offers the opportunity of parallel processing and efficient hardware architectures. Our partitioned approach is capable to take advantage of the mentioned self similarities as well as the preferences of the local processing of each subimage.

- Besides the excellent techniques to compress the individual code blocks the post compression rate distortion optimization is the key to obtain good performance. The price to pay is the computational complexity in order to implement this optimization. The currently available JPEG2000 codecs suffers especially from the tremendous computation times [Ada01], [Tau02]. However, the post compression rate distortion optimization can be combined with the partitioned approach in order to improve the compression efficiency.

Appendix A

Hard/Software Interface MicroEnable

A.1 Register/DMA on Demand Transfer, C example

```
#include<stdio.h>
#include"menable.h"
int main(int argc, char** argv) {
  microenable* board;
  fpga_design* design;
  volatile unsigned long *reg;
  int error;
  char c;

  // initialize board
  board = initialize_microenable( ) ;
  if( board == NULL ){
    fprintf(stderr,"initialization error %d\n",Me_GetErrCode(board));
    exit(-1);
  }
  // load design
  design = load_design( board,"./example.hap");
  if(design == NULL ){
    fprintf(stderr,"load design error %d\n",Me_GetErrCode(board));
    exit(-1);
  }
  // config FPGA
  if(( error = configure_fpga( board, design )) < 0 ){
    fprintf(stderr,"config error %d\n",error);
    exit(-1);
  }
  // set PLX clock in MHz
  set_plx_clk(board, design,  40 );
  // set design clock in MHz
  set_fpga_clk(board, design,  40 );
  // get access pointer
  reg = GetAccessPointer(board);
  if (reg  == NULL ){
    fprintf(stderr,"Get access Pointer is NULL\n");
    exit(-1);
  }
  // access the design
  // read the fixed register
  printf("Reg 3 : 0x%8X\n\n",reg[3]);
```

```
  // set a register at address 0x0
  reg[0x000000] = 0x00000f0f;
  // read registers
  printf(" Ctr Reg 0 : 0x%8X\n",reg[0x000000]);
  printf(" Ctr Reg 1 : 0x%8X\n",reg[0x000001]);
  printf(" Ctr Reg 3 : 0x%8X\n",reg[0x000003]);

  // wait for user response
  c = getchar() ;
  printf(" DMA Access \n");
  // allocate buffers
  unsigned long writeBuffer[128];
  unsigned long readBuffer[128];
  int i;

  // initialize the buffers
  for( i = 0; i < 128 ; i++ ){
    readBuffer[i] = 0;
    writeBuffer[i] = i;
  }

  // Load the Rotator register
  reg[1] = 0x00000001;

  // start the rotation
  reg[0] = 0x20000000;

  // read data via dma on demand from the design
  error = ReadDmaDemandData(readBuffer,
      32*sizeof(unsigned long),
      1,
      0x04000000,
      1,
      board);
  printf("transfer OK\n");
  // error Check
  if( error < 0 )
    printf(" error %d during data transfer",error );
  // display the result
  for( i = 0; i < 32 ; i++ ){
    printf("%8X\n",readBuffer[i]);
  }
  // return from the programm
  return 0;
}
```

A.2 Register/DMA on Demand Transfer, VHDL example

```
library IEEE;
use IEEE.std_logic_1164.all;

entity registerSpace is
  port(
    origWord   : out std_logic_vector( 31 downto 0 );
    address    : in std_logic_vector( 21 downto 0 );
    dataIn  : in std_logic_vector(31 downto 0 );
    dataOut  : out std_logic_vector(31 downto 0 );
    writeRdy   : in std_logic;
    readRdy  : in std_logic;
    clk    : in std_logic;
    reset     : in std_logic;
    rotateRdy : in std_logic;
    rotateStart : out std_logic;
    regReset  : out std_logic
  );
end registerSpace;

architecture registerSpaceArchitecture of registerSpace is
  signal registerA : std_logic_vector( 31 downto 0 );
  signal registerB : std_logic_vector( 31 downto 0 );
begin
  ------------------------------------------------------------------
  -- Write process
  ------------------------------------------------------------------
  WRITE_REG_PROCESS : process (reset, clk)
  begin
    if reset = '1' then  -- reset active
      registerA <=  ( others => '0' );
    elsif (clk'event and clk = '1') then -- on rising edge
      -- registers set from the PCI side
      if (writeRdy = '1' and address( 3 downto 0 ) = "0000" ) then
        -- register address 0x000000
        registerA( 30 downto 0) <= dataIn ( 30 downto 0 );
        registerA(31) <= rotateRdy;
      elsif (writeRdy = '1' and address( 3 downto 0 ) = "0001" ) then
        -- register address 0x000001
        registerB <= dataIn ( 31 downto 0 );
      else -- no write access
        registerA(30 downto 0 ) <=  registerA(30 downto 0 );
        registerA(31) <= rotateRdy;
        registerB <= registerB;
      end if;
    end if;
  end process WRITE_REG_PROCESS;
  ------------------------------------------------------------------
  -- Read Process
  ------------------------------------------------------------------
  READ_REG_PROCESS : process ( readRdy, address, registerA, registerB )
  begin
    if ( readRdy = '1' ) then
      -- Adr: 0x00000000
      if ( address( 3 downto 0) = "0000" ) then
        dataOut <= registerA;
```

```
      -- Adr: 0x00000001
      elsif ( address( 3 downto 0) = "0001" ) then
        dataOut <= registerB;
      -- Adr: 0x00000003
      elsif ( address( 3 downto 0) = "0011" ) then
        dataOut( 15 downto 0  ) <= "1010101000010010";
        dataOut( 31 downto 16 ) <= "0001001000110100";
      else
        dataOut <= (others=>'0');
      end if;
    else
      dataOut <= (others=>'0');
    end if;
  end process READ_REG_PROCESS;

  ---------------------------------------------------------------------
  -- Set the registers to the output
  ---------------------------------------------------------------------
  regReset <= registerA(30);
  origWord <= registerB;
  rotateStart <= registerA(29);

end registerSpaceArchitecture;
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity rotator is
  port (
    wordIn      : in STD_LOGIC_VECTOR (31 downto 0);
    wordOut         : out STD_LOGIC_VECTOR (31 downto 0);
    rotateEnable    : in std_logic;
    rotateStart     : in std_logic;
    rotateRdy       : out std_logic;
    rotateAlmostRdy : out std_logic;
    clk             : in std_logic;
    reset      : in STD_LOGIC
  );
end rotator;

architecture rotatorBehave of rotator is
  type fsmStateType is ( idle, rotate );
  signal fsmState : fsmStateType;
begin
  FSM_PROCESS : process( clk, fsmState, rotateStart, reset, wordIn )
    variable countValue : integer;
    variable tmpValue : std_logic_vector( 31 downto 0 );
    variable tmpBit : std_logic;
  begin
    if reset = '1' then
      countValue := 0;
      fsmState <= idle;
      wordOut <= wordIn;
      rotateAlmostRdy <= '0';
    elsif clk'event and clk = '1' then
      -- default assignment
      rotateAlmostRdy <= '0';

      case fsmState is
        when idle =>  -- wait for the start signal
          if rotateStart = '1' then
            tmpValue := wordIn;
            countValue := 0;
            fsmState <= rotate;
            wordOut <= tmpValue;
          end if;
          rotateRdy <= '0';
        when rotate  => -- rotate until the 32 round
          if countValue <= 30 then
          -- rotate another bit
            if rotateEnable = '1' then
              tmpBit := tmpValue(31);
              tmpValue(31 downto 1) := tmpValue(30 downto 0);
              tmpValue(0) := tmpBit;
              wordOut <= tmpValue;
              countValue := countValue + 1;
              rotateRdy <= '0';
            end if;
            -- generate the rotateAlmostRdy signal
            if( countValue = 31 ) then
```

```
              rotateAlmostRdy <= '1';
            end if;
          else
            -- finish the rotation
            rotateRdy <= '1';
            if rotateStart = '1' then
              fsmState <= idle;
            end if;
          end if;
      end case;
    end if;
  end process;
end rotatorBehave;
```

A.3 Matlab/METAPOST-Scripts

```
%
% Matlab code to draw the limit functions ψ̃ and ψ̃
%

impulse=[1];

g0=sqrt(2)*[-1/8,1/4,3/4,1/4,-1/8];
h0=1/sqrt(2)*[0,0,-1/2,1,-1/2];
dwtsig=impulse;

for level=1:7
    dwtsig=sqrt(2)*conv(dwtsig,g0);
    g0=upsample(g0);
    h0=upsample(h0);
end

dwtsig2=sqrt(2)*conv(dwtsig,h0);
dwtsig=sqrt(2)*conv(dwtsig,g0);


x=[-2:4/(length(dwtsig)-1):2];
length(x)
y=[-2:4/(length(dwtsig2)-1):2];
length(y)
plot(x,dwtsig,y,dwtsig2); grid

%
% Matlab code to compute the ranges of wavelet coefficients
%

l=[-1/8,1/4,3/4,1/4,-1/8];
h=[-1/2,1,-1/2];

echo on;
cdfRange(6,l,h,127,-128);
cdfRange(1,l,h,255,-255);
cdfRange(2,l,h,319,-319);
cdfRange(3,l,h,351,-351);
cdfRange(4,l,h,358,-358);
cdfRange(4,l,h,216,-217);
```

```
%
% Matlab code -- utility functions
%

function cdfRange(level,l,h,max,min);
maxl=max;
maxh=max;
minl=min;
minh=min;

lup=l;
hup=h;

for i=1:level
  [maxl,minl]=maxmin(l,max,min)
  [maxh,minh]=maxmin(h,max,min)
  lup=upsample(lup);
  hup=upsample(hup);
  h=conv(l,hup);
  l=conv(l,lup);
end

function f = upsample(x)
n=length(x);
for i=1:n
  f(2*i-1)=x(i);
  if i<n
    f(2*i)=0;
  end
end

function [maxf,minf] = minmax(f,max,min);
maxf=0;
minf=0;
for I = 1:length(f);
  if f(I)<0
    maxf=maxf+min*f(I);
    minf=minf+max*f(I);
  else
    maxf=maxf+max*f(I);
    minf=minf+min*f(I);
  end
end
maxf=ceil(maxf);
minf=floor(minf);
end
```

```
%
% METAPOST code to draw the limit function φ̃
%

input mpFunctions

beginfig(1);

e:=10mm;

drawxyaxis(1mm,0e,0e,1e,1e,-1,2,-2,4,1,1,1,2);

numeric g[],dwtsigg[];
g1=sqrt(2)*(-1/8);
g2=sqrt(2)*(1/4);
g3=sqrt(2)*(3/4);
g4=sqrt(2)*(1/4);
g5=sqrt(2)*(-1/8);

dwtsigg[1]:=1;

approx:=7;
i:=1;

for level=1 upto approx:
  conv(dwtsigg,g);
  i:=1;
  forever:
    dwtsigg[i]:=sqrt(2)*dwtsigg[i];
    exitif unknown dwtsigg[i+1];
    i:=i+1;
  endfor;
  if (level<approx):
    upsample(g);
  fi;
endfor;

pair p[];
for k=1 upto i:
  p[k]:=(k*0.1e,dwtsigg[k]*0.5e);
endfor;
draw (
  (for k=1 upto i-1: p[k] -- endfor p[i])
  xscaled (10/((2**approx)-1))
  ) shifted (0,2e);
endfig;
end;
```

Bibliography

[Ada01] Michael D. Adams. *The JasPer Project Home Page*, 2001. http://www.ece.uvic.ca/~mdadams/jasper/.

[Amp02] Amphion Semiconductor Ltd. *JPEG2000 Core Accelerates Next-Generation Image Compression for Digital Cameras*, 2002. http://www.amphion.com.

[And01] Peter Andree. MATH Online, September 2001. http://www.ksk.ch/mathematik/mathonline/.

[CDF92] A. Cohen, I. Daubechies, and J.C. Feauveau. Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics*, 45:485–560, 1992.

[CDSY97] R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo. Lossless image compression using integer to integer wavelet transforms. In *International Conference on Image Processing (ICIP), Vol. I*, pages 596–599. IEEE Press, 1997.

[CFH96] H. Chao, P. Fisher, and Z. Hua. An approach to integer wavelet transformations for lossless image compression. *Technical Paper, University of North Texas, Denton, TX 76208*, 1996.

[CMQW94] R. Coifman, Y. Meyer, St. R. Quake, and M. V. Wickerhauser. Signal processing and compression with wavelet packets. In *Wavelets and Their Applications*, volume 442 of *NATO ASI Series C: Mathematical and Physical Sciences*, pages 363–379. Kluwer Academic Publishers, Dordrecht/Boston/London, 1994.

[Dau92] I. Daubechies. *Ten Lectures on Wavelets*. Number 61 in CBMS/NSF Series in Applied Math. SIAM, 1992.

[Fey01] G. Fey. Set Partitioning in Hierarchical Trees: A FPGA - implementation. *Master Thesis (in german), Martin-Luther-University Halle, Germany*, 2001.

[FGC93] G. Feygins, P. G. Gulak, and P. Chow. Minimizing error and vlsi complexity in the multiplication free approximation of arithmetic coding. *IEEE Transactions on Signal Processing*, 1993.

[For83] O. Forster. *Analysis 1,2,3*. Vieweg, Braunschweig, Germany, 1983.

[Fow02] James E. Fowler. *The QccPack*, 2002. http://qccpack.sourceforge.net.

[Gmb99] Silicon Software GmbH. microEnable - A FPGA prototyping platform. *Data sheet and User manual*, 1999.

[Haa10] A. Haar. Zur Theorie der orthogonalen Funktionensysteme. *Math. Annal.*, 69:331–371, 1910.

[inS02] inSilicon Corporation. *JPEG2000 Encoder*, 2002. http://www.inSilicon.com.

[Mal89] S. Mallat. A theory for multiresolution signal decomposition: the wavelet representation. *IEEE Transactions on Pattern Recognition and Mach. Intell.*, 11(3):674–693, 1989.

[New96] Bernie New. Using the Dedicated Carry Logic in XC4000E. Technical report, Xilinx Inc., July 1996.

[OS89] A.V. Oppenheim and R.W. Shafer. *Discrete-Time Signal Processing*. Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1989.

[PM88] W. Pennebaker and J. Mitchell. An overview of the basic principles of the q-coder adaptive binary arithmetic coder. In *IBM J. Res. Develop*, volume 32(6), pages 717–726, 1988.

[RFM02a] J. Ritter, G. Fey, and P. Molitor. An efficient FPGA implementation of the SPIHT algorithm. In *Tenth International Symposium on Field Programmable Gate Arrays, Poster session*. ACM, March 2002.

[RFM02b] J. Ritter, G. Fey, and P. Molitor. SPIHT implemented in a XC4000 device. In *Proceedings of the Midwest Symposium on Circuit and Systems*. IEEE, 2002.

[RL81] J. Rissanen and G. G. Langdon. Universal modeling and coding. *IEEE Transactions on Information Theory*, 2:27, 1981.

[RM00] J. Ritter and P. Molitor. A partitioned wavelet-based approach for image compression using FPGA's. In *Proceedings of the 2000 Custom Integrated Circuits Conference*, pages 547–550. IEEE, 2000.

[RM01] J. Ritter and P. Molitor. A pipelined architecture for partitioned DWT based lossy image compression using FPGAs. In *Ninth International Symposium on Field Programmable Gate Arrays*. ACM, February 2001.

[Say96] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, Inc., San Francisco, CA 94104-3205, 1996.

[Sha] C. E. Shannon. A Mathematical Theory of Communication. *Bell Systems Technical Journal*, 27:379–423,623–656.

[Sha93] J.M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. In *Transactions on Signal Processing*, volume 11, pages 3115–3162. IEEE, 1993.

[SP96] A. Said and W. A. Pearlman. A New Fast and Efficient Image Codec Based on Set Partitioning in Hierarchical Trees. *IEEE Transactions on Circuit and Systems for Video Technology*, 6, 1996.

[SS96] W. Sweldens and P. Schröder. Building your own wavelets at home. In *Wavelets in Computer Graphics*, pages 15–87. ACM SIGGRAPH Course notes, 1996.

[Sut99] St. Sutter. FPGA-architectures for partitioned wavelet transformations on images. *Master thesis (in German), Martin-Luther-University Halle-Wittenberg, D-06099 Halle, Germany*, 1999.

[Swe96] W. Sweldens. The lifting scheme: A custom-design construction of biorthogonal wavelets. *Appl. Comput. Harmon. Anal.*, 3(2):186–200, 1996. http://cm.bell-labs.com/who/wim/papers/papers.html#lift2.

[Tau00] D. Taubman. High performance scalable image compression with EBCOT. In *Transactions on Image Processing*, volume 9(7), pages 1158–1170. IEEE, 2000.

[Tau02] D. Taubman. *Kakadu Software - A Comprehensive Framework for JPEG2000*, 2002. http://www.kakadusoftware.com.

[Thi01] H. Thielemann. Adaptive construction of wavelets for image compression. *Master Thesis, Martin-Luther-University Halle, Germany*, 2001.

[TM02] D. Taubman and M. Marcellin. *JPEG 2000: Image compression fundamentals, standards and practice*. Kluwer Academic Publishers, Norwell, Massachusetts 02061, 2002.

[VK95] M. Vetterli and J. Kovacevic. *Wavelets and Subband Coding*. Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1995.

[Wal91] G.K. Wallace. The JPEG still picture compression standard. *Comm. ACM, vol. 34*, 34:30–44, 1991.

[WP00] F. W. Wheeler and W. A. Pearlman. SPIHT Image Compression without Lists. *IEEE International Conference on Acoustics, Speech and Signal Processing ICASSP*, 2000.

[Xil97] Xilinx Inc. *Xilinx XC4000E and XC4000X Series FPGA Data Sheet*, November 1997.

Jörg Ritter

ritter@informatik.uni-halle.de

Persönliche Angaben

Familienstand: ledig

Staatsangehörigkeit: deutsch

Geburtstag: 12.04.1971

Geburtsort: Greiz

Ausbildung

1987	Polytechnische Oberschule Greiz-Pohlitz Abschlußzeugnis (Gesamtnote: sehr gut)
1989	Erweiterte Oberschule *Theodor Neubauer* Greiz Abitur (Gesamtnote: sehr gut)
1997	Universität des Saarlandes Diplom-Hauptprüfung für Informatiker, Nebenfach Wirtschaftswissenschaften (Gesamtnote gut)
September 2002	Martin-Luther-Universität Halle-Wittenberg Einreichung der Promotionsarbeit

Wehrdienst

1989-1990	Grundwehrdienst in der Nationalen Volksarmee der ehemaligen DDR

Beruflicher Werdegang

1994-1997	Universität des Saarlandes hilfswissenschaftlicher Mitarbeiter im Sonderforschungsbereich 124 *VLSI Entwurfsmethoden und Parallelität* der Deutschen Forschungsgemeinschaft (DFG)
1997-1998	Universität des Saarlandes wissenschaftlicher Mitarbeiter am Lehrstuhl für Angewandte Mathematik und Theoretische Informatik (Prof. Dr. Günter Hotz)
1998-2002	Martin-Luther-Universität Halle-Wittenberg wissenschaftlicher Mitarbeiter am Lehrstuhl für Technische Informatik (Prof. Dr. Paul Molitor)

Promotionsverfahren Jörg Ritter

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, daß ich diese Arbeit selbständig und ohne fremde Hilfe verfaßt, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Um einen Doktorgrad habe ich mich bisher nicht beworben.

Jörg Ritter

Halle/Saale, den 06.12.2002