NewWave
A hardware implementation of wavelet decompression of digital video images

Tomas Möller and Jacob Ström

Supervisor: Björn Breidegard

Lund Institute of Technology
Sweden

Email: T.Möller: d91tm@efd.lth.se, J.Ström: d91js@efd.lth.se
B. Breidegard: bjorn@dit.lth.se

March 1995

Abstract

New Wave is a hardware implementation of real time digital video decompression. The algorithm used is a wavelet transform coding which is a lossy algorithm that achieves very high compression factors with low degradation of the video image. The New Wave chip is capable of decompressing 25 video frames per second. Each frame consists of 512×512 pixels in 256 shades of gray. The solution could easily be generalized to colour images.

Contents

1 Introduction

It is a well known fact that computer images are very bulky - they require lots of memory when stored and much bandwidth when transmitted. With digital video this problem becomes even more appearent. The solution to this is to compress the image before transmission and decompress it afterwards. This can be done in two ways, using either lossless or lossy coding.

Lossless coding

Lossless coding reconstructs the compressed data excactly, with every single byte correct. It is applicable on every type of data: text information, code as well as computer images. The drawback is that the compression factor [1], which depends on the amount of redundancy in the data, is rather low, seldom more than 2 to 3.

Lossy coding

Lossy coding means that some information is lost during the encoding. The decompressed data is not an excact copy of the original, but pretty close. The introduced errors makes it easier to compress the data. For example, the sequence 255, 1, 0, 0, 2 ,0 ,0, 0, 1, 0, 0 1 could be exchanged with 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 which is much easier to compress. Of course this could not be done if the data represents executable code or text information, but if it is part of an image, the difference may not even be visible to the human eye.

Much work has already been done in the area of lossy coding. For a thorough understanding of the advantages with New Wave, we have to get deep into existing standards and investigate their drawbacks.

1.1 The MPEG standard

One decompression algorithm that is being used more and more frequentely is the MPEG algorithm, which is a standard for lossy compression of moving pictures[2]. In its simplest form, an MPEG sequence is just a couple of JPEG[3] images stored together in a file. Smart MPEG encoders are able to use features like differential coding, i.e. storing only the difference from the previous frame. This tends to reduce the amount of data with a factor of three. We will, however, only consider the simpler MPEG files which contains JPEG-frames only. This will make it easier to compare the formats, as we can look at still frames (JPEG-images) rather than moving sequences.

1.1.1 The JPEG algorithm

For simplicity, forget about colors and suppose that the images are greyscale only. In the JPEG standard the image is subdivided into blocks of 8×8 pixels. A discrete cosine transform is applied on each block. The result is a 8×8 matrix with frequency coefficients. (An inverse cosine transform of this matrix would bring back the 8×8 pixel block.) The high frequency coefficients will be situated in the lower right part of the matrix, and the lower frequency coefficients in the top left. The JPEG standard takes advantage of the fact that the human eye is less sensitive to higher frequency components of the image. Therefore, the plan is to encode the lower frequency more accurately than the higher ones. This is done by dividing the matrix with a quantisation matrix where the lower right part has higher numbers than the upper left. This will bring many of the

[1] compression factor $= \frac{\text{original size}}{\text{compressed size}}$

[2] The ISO/IEC 11172-2.

[3] JPEG is a standard for lossy compression of static images

high frequency coefficients to zero or almost zero. To efficiently code this the coefficients are coded with symbols of variable length, where each symbol stands for a number of zeroes followed by the first non-zero coefficient. Symbols that are more frequent are coded with a shorter number of bits according to a table. The JPEG standard contains a standard table but custom tables can be used. MPEG data is always coded using a standard table, i.e. custom tables are not allowed.

1.1.2 Disadvantages of the JPEG/MPEG algorithm

The JPEG and MPEG algorithm share two main disadvantages. The first is the limitation to 8×8 pixel blocks. At high or even at moderate compression levels they become visible. The second disadvantage is that JPEG/MPEG images often makes the impression of being blurry. This is an effect of the strong surpression of high frequency components - the algorithm acts like a low-pass filter on the image, which reminds of the effect of wearing a pair of dimmed glasses.

2 The Wavelet algorithm

Why is wavelet coding so efficient? Here we will try to give you a mathematical as well as an intuitive answer to that question. To make things easier for us, we first consider one dimensional functions (such as sound) and move on later to two dimensional functions (such as images).

If you are not interested in why wavelets are so efficient but want to know how it is done anyway, you can go directly to the subsection called "The two dimensional wavelet transform".

Wavelet coding belongs to the family of transform coding algorithms. The very idea with transform coding is that the values of the transformed function are more gathered – not so smeared out – as the original function. If you look at figure 1, the transformed function is zero almost everywhere except for some

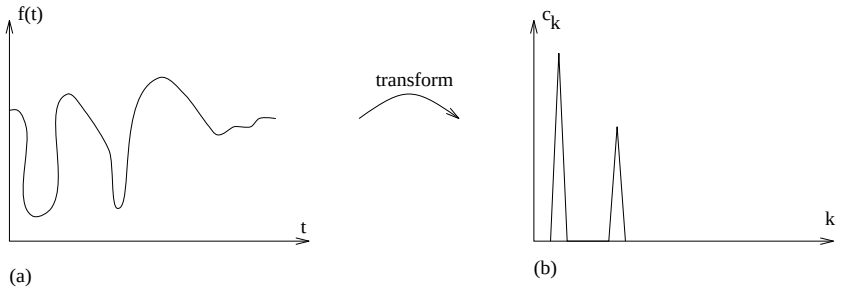

Figure 1: The dream transform.

short intervals. If you apply run length coding[4] on the transformed image, the resulting amount of data would be very small. The trick is therefore to find a transform that brings most of the values in the transformed function (i.e. the coeffcients) to zero. To do a transform you find a number of basis functions,

[4] In run length coding you simply replace consecutive zeroes with the number of zeroes. See Appendix B.2

$\varphi_k(t)$ and you then describe your function $f(t)$ with a summation

$$f(t) = \sum_{k=0}^{\infty} c_k \varphi_k(t). \tag{1}$$

No we hope that the coefficients c_k will be more like figure 1b than $f(t)$ and hence contain large segments of zeroes.

Take for example the cosine transform. Here the basis functions are $\varphi_k(t) = cos(\pi k t)$. We transform the function:

$$f(t) = \begin{cases} 0, & t \leq \frac{1}{2} \\ 1, & t > \frac{1}{2} \end{cases} \tag{2}$$

With some calculations, we get $c_k = \frac{1}{2} \frac{sin(\pi \frac{k}{2})}{\pi \frac{k}{2}}$ The result can be found in figure 3 where both the originalfunction $f(t)$ and the resulting coefficients c_k are showed.

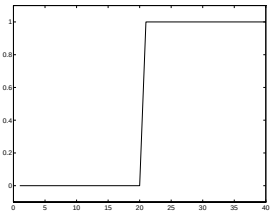

Figure 2: The original function $f(t)$.

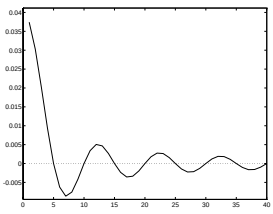

Figure 3: The cosine transform of $f(t)$.

We now start investigating various bases $\varphi_k(t)$. Is for example the cosine transform a good one? Well, it has one disadvantage, and that is that the jump of $f(t)$ in $t = \frac{1}{2}$ in the example influences all of the coefficients c_k. It would have been much better if it had only influenced a limited number of c_k, say all c_k that have k in some interval [a,b]. Then all c_k for which k not belonged to [a,b] would have been zero with a good compression as a result. This feature is called *time locality* and means that the behaviour of the function $f(t)$ for a specific time t_0 (for example $t_0 = \frac{1}{2}$) does only influence a limited number of the coefficients c_k in the transform.

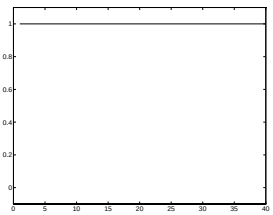

Figure 4: Constant function $f(t) = 1$.

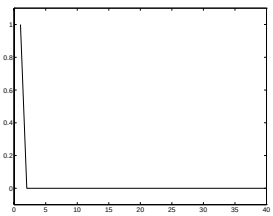

Figure 5: After cosine transform.

A good transform should therefore have time locality. However, to get as many zeroes as possible among the transformed coefficeints c_k we also want the transform to have good frequency behaviour. Imagine a constant function $f(t) = 1$ as

showed in figure 4. If we use the cosine transform (which has not only good, but optimal frequency behaviour) on $f(t)$ the result will be $c_0 = 1$ and all $c_{k\neq 0} = 0$ (see fig 5). This feature is called *frequency locality* and means that behaviour at a specific frequency f_0 in the frequency spectrum of $f(t)$ only influences a limited number of the coefficients c_k.

The optimal transform should have both time and frequency locality in order to bring as many as possible of the coefficients c_k to zero. However, this means that both our basis functions $\varphi_k(t)$ and their frequency spectra must be zero outside a finite region[5]. Such functions does not exist, as Heisenbergs relation of incertainty shows that the variance of a function times the variance of its frequency spectrum always exceeds or equals $\frac{1}{2}$. (In our case both the variances are zero).

What basis functions are we then to choose? JPEG chooses the cosine transform which has optimal frequency locality but no time locality whatsoever. Approximately 15 years ago[6], functions were found that had time locality but yet reasonable frequency behaviour, and at the same time were possible to use as basis functions. These were called wavelet functions. We are now going to look at a basis of wavelet functions. They can look like shown in figure 6:

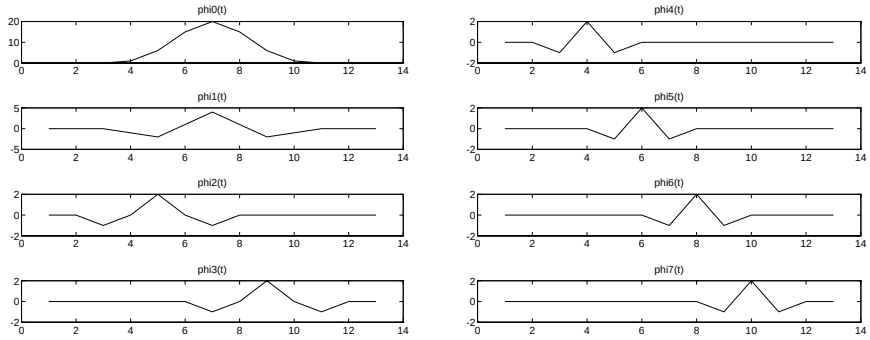

Figure 6: A wavelet base. Note that $\varphi_k(t)$ is written as phik(t).

Note here that the $\varphi_k(t)$ have more detail with increasing k. This means that they contain higher frequencies and this is an indication of the good frequency behaviour of the base. It is also easy to see that the higher k, the more local is $\varphi_k(t)$ in time (the smaller is the nonzero part of $\varphi_k(t)$). In figure 7 a function $f(t)$ is transformed using a base similar to this base[7].

We have seen that wavelets combine good frequency beaviour with time locality, but it does not end there. As we soon will be able to see, we do not need k different basis-functions to perform the wavelet transform. We just need the two wavelets in figure 8 - one that has lowpass and another with highpass characteristics. By doing repeated transformations with translated versions of these two wavelets, we can get the same endresult as if we had used the basis in figure 6 once. The first transformation uses translated versions of the lowpass wavelet for the first $\frac{k}{2}$ basisfunctions, and the last $\frac{k}{2}$ basis functions are translated versions of the highpass wavelet as in rightmost part of figure 6. In figure 9 we can

[5] We leave this statement unproven

[6] A brand new discovery, at least in the mathematical perspective of time.

[7] The only difference is that the base used includes a larger number of basis functions than would be proporiate to depict here

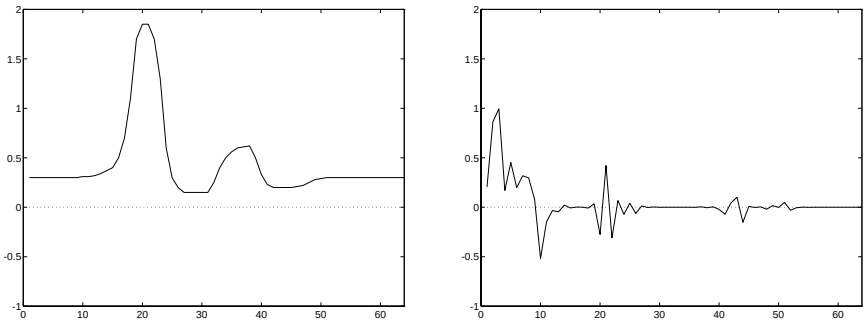

Figure 7: A function $f(t)$ is transformed using a base similar to the one in the previous fiugre.

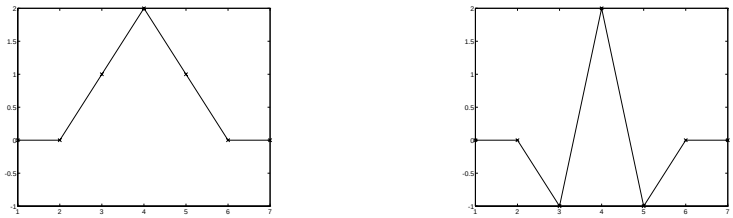

Figure 8: The whole wavelet transform could be done using only these two wavelets as basis functions.

see the impact of this first transfomation of our function $f(t)$ from figure 7a. Notice that the first half of the result ressembles the original $f(t)$ very much, but it is half as wide. In the next step in the iteration, we only transform this first half. The second half of the coefficients are left untouched. Once again the first half of the basis functions are lowpass wavelets, and the other half is made from the highpass wavelet. The result will now look like figure 10. After a third iteration we have the same result as in fig 7b. This iterated form of the wavelet transform is very good for computer implementations. We need not spend all that memory to store the basis functions, as there are only two of them now. When it comes to hardware implementation, the gain is even bigger; we have a limited number of filter coefficients and can custiomize the multipliers for them.

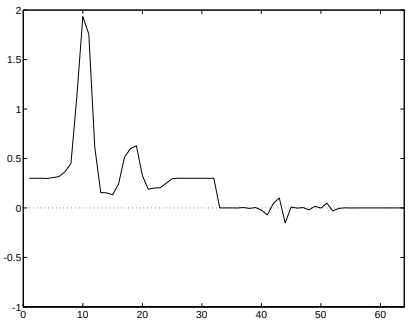

Figure 9: After one iteration.

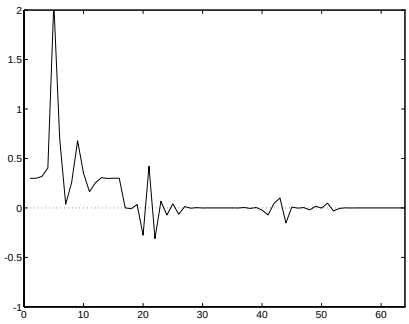

Figure 10: After two iterations.

2.1 The two dimensional wavelet transform

Until now we have only considered one dimensional functions. A grey scale image is best repesented as a two dimensional function $I(x, y)$ where I is the intensity in the point (x, y). This means that we need a two dimensional wavelet transform. However, a two dimensional transform can be obtained by taking the one-dimensional transform first along the x-axis and then along the y-axis. If we use the iterated wavelet transform described above, it can look like in figure 12.

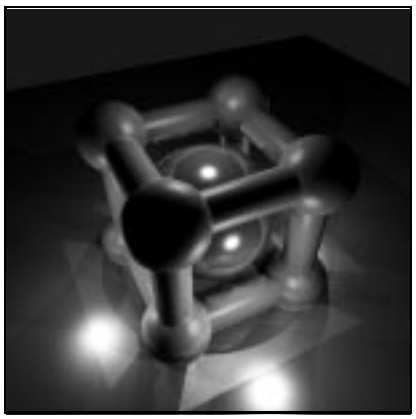

Figure 11: Original image.

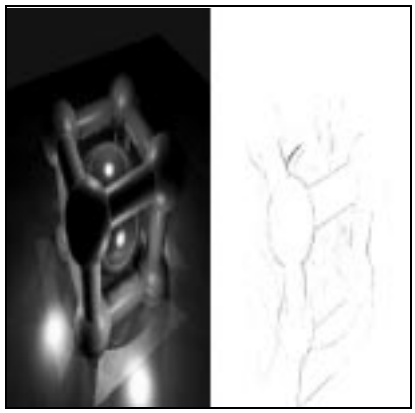

Figure 12: Transformed along x-axis.

The original image (figure 11) is transformed along the x-axis, and we notice that the left part of the transformed image (figure 12) ressembles the original but is narrower. The rightmost part is a highpass version of the original which means that vertical edges in the image are amplified. After this comes the transform along the y-axis (figure 13).

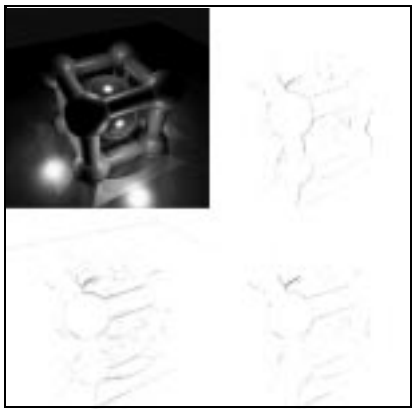

Figure 13: Transformed along both axis.

Figure 14: Transform repeated five times.

We now get one quarter with lowpass information (the upper left), and three quarters with highpass information. The lowpass part still resembles the original image, but the other three contain lots of white area, which means zero

coefficients[8]. As we can see, large parts of the image contains zero or near zero coefficients. To bring even more coefficients to zero, we redo the transformation on the lowpass part. After five iterations, the result looks like in figure 14. The lowpass area is now so small that it is not neccesary to do another iteration.

2.2 The encoding

To compress the image, we save the lowpass data as it is and does a run-length encoding of the rest. Before entering the run-length encoding phase, we need to bring all the near zero coefficients to exactly zero. Otherwise the run-length encoding will not be very successful. By dividing the coefficients with some number, all the near zero coefficents turn into true zeroes, and the run length encoding can start. Each coefficient is translated into a symbol whose msb[9] describes what type of symbol it is. If the bit is 0, the rest of the bits in the symbol describes the coefficient value. If the bit is 1, the rest of the bits describes the number of zeroes before the next coefficient. To encode these symbols as good as possible, Huffman encoding is used. To introduce as few new symbols as possible, a zero symbol (msb=1) contains not the number of zeroes but the log_2 of the number of zeroes. This means that to code for example five zeroes, you need two run-length symbols with msb=1: One with value 2 and the other with value 0, meaning a run of $2^2 + 2^0 = 5$ zeroes. These symbols can be reused, next time you want a run length of four zeroes you just need the one with value 2. After the Huffman encoding you get the small Huffman code. This is saved to a file together with the Huffman lookup table and the lowpass data.

2.3 Reconstruction

To reconstruct an image the whole procedure is made backwards: First the Huffman code is decoded with help of the Huffman lookup-table. Then the runlength symbols are decoded to get the transformed image (figure 14). A multiplication with the division constant from above. The wavelet coefficients will not be the same as before the division and multiplication, and this is what makes the wavelet algorithm lossy. However, as the wavelet transform is energy preserving[10], we know that the changes in the reconstructed image will not be larger than the changes in the wavelet coefficients. The inverse wavelet transformed is then performed. If we choose our wavelets right, we can use the same two filters from the transform to do the inverse transform as well! Even though this may seem fantastic, this is not always the best choice. It is often more important to be able to reconstruct an image fast than to encode it fast. Examples of such applications are video on CD-ROM, video on demand etc. A fast reconstruction is then needed. Shorter wavelets means fewer multiplications and additions and faster reconstruction. We shall therfore see to that the reconstructions filters are as short as possible. In [Adelson90] Adelson and Simoncelli showed that it was possible to use reconstruction filters with a length of three containting only the numbers 1 and 2, which means only shifts and adds. This makes things much easier and faster to implement both in software and in hardware. The two small filters are depicted in figure 8. The penalty is that the construction filters must be of infinite length! However, Adelson and Simoncelli also found that a truncated filter with a length of 15 was satisfactory for all practical matters.

[8] A white pixel means a coefficient value of zero. The more the coefficient value differs from zero, the darker is the pixel.

[9] msb = most significant bit

[10] We leave this statement unproven

3 A Wavelet vs JPEG comparison

Presented in this section is a performance comparison between JPEG and Wavelet compression. Two measures of compression performance are used - *compression ratio* and *peak signal-to-noise ratio (PNSR)*. Even though you consider some images better than others, these ratios are used as objective measures (in cases where it is hard for the human to determine image quality). Assuming P bits in the original image and C bits in the compressed image, the definition of of compression ratio R is

$$R = \frac{P}{C}.$$

The test image is a greyscale image with 8 bits per pixel. According to [Hilton91], the peak signal-to-noise ratio (PNSR), in decibels (dB) is computed as

$$PNSR = 20 \log_{10} \frac{255}{RMSE}$$

where

$$RMSE = \sqrt{\frac{1}{NM} \sum_{i=1}^{N} \sum_{j=1}^{M} (f(i,j) - \hat{f}(i,j))^2}.$$

Here, N is the width and M the height of the image. The pixel value at position (i, j) of the original image is denoted $f(i, j)$ and the decompressed image $\hat{f}(i, j)$.

Figure 15 shows the original image used throughout this comparison. In figure 17 and figure 18, wavelet respectively JPEG compressed images are shown. R is about 34 for both images and if scrutinized, small 8×8 squares can be found in the JPEG image. A PNSR comparison shows that the Wavelet compressed image is better.

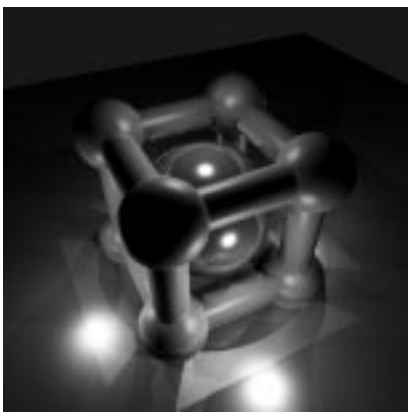

Figure 15: The original test image. File length 262159 bytes.

Figure 16: Image saved with 1 byte per 8×8. File File length 4096 bytes.

The next pair of images, figure 19 and 20, shows the two compression methods with $R \approx 62$. Here the JPEG method suffers greatly from the "square-effect", while the first still has got quite good quality. It is intresting to compare these two images with the image in figure 16, since all three use about $4k$ bytes of storage. The image in figure 16 is the original image with resolution

decreased to 64×64. The quality of the JPEG image is slightly better than this image and the wavelet image really outperforms JPEG here.

Figure 17: Wavelet decompressed picture. Compression ratio 34.7. File length 7754 bytes.

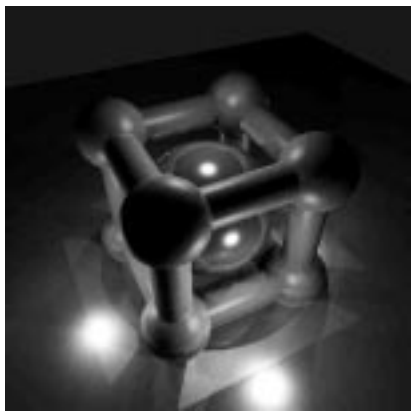

Figure 18: JPEG decompressed picture. Compression ratio 33.7. File length 7788 bytes.

Since JPEG has got a, not neglectable, overhead for each 8×8 square a maximum compression ratio for every picture can easily be found. The result is shown in figure 22, with $R \approx 72$. This phenomenon does not occur for wavelet compression until much later and in figure 21 an image with $R \approx 120$ is shown.

Figure 19: Wavelet decompressed picture. Compression ratio 63.7. File length 4114 bytes.

Figure 20: JPEG decompressed picture. Compression ratio 60.5. File length 4334 bytes.

In figure 23 and 24 wavelet decompressed images with $R = 211.4$ and $R = 303.8$ are shown. The last image has got a compression ratio of more than four times greater than the JPEG image in 22, and still the quality of the wavelet decompressed image outperforms JPEG. Notice that the wavelet method even conserves the highlights pretty well in these last images.

In [Strom94] a more thoroughly comparison is presented (using the famous

Figure 21: Wavelet decompressed picture. Compression ratio 119.8. File length 2189 bytes.

Figure 22: JPEG decompressed picture. Max acheivable compression ratio 71.9. File length 3645 bytes.

Figure 23: Wavelet decompressed picture. Compression ratio 211.4. File length 1240 bytes.

Figure 24: Wavelet decompressed picture. Compression ratio 303.8. File length 863 bytes.

Lena image) and similar results were obtained.

This comparison shows that it is meaningful to use wavelet compression, and that a hardware implementation of wavelet decompression would be very useful.

4 Implementation

When developing a hardware construction, several system models are often created along towards the final product. The first model is usually a program written in a high-level language. The compressing and uncompressing programs were based on *EPIC* and *UNEPIC* developed at MIT by Eero P. Simoncelli and Edward H. Adelson. The next step was to develop a clock cycle true model using BBDS (see A.1 for a description of BBDS).

Most of the important trade-offs were made in the first software model and

these are described in section 4.1. The more detailed construction decisions for the hardware solution are described in section 4.2.

4.1 Software

To be able to implement the decompression algorithm in hardware, the *EPIC* and *UNEPIC* programs had to be partially rewritten. The decompression program, *UNEPIC* first reads the image header of the decompressed file, stores several constants and the Huffman table, then it Huffman decodes and run-length decodes the data following the header. After scaling the decompressed data, the program performs the inverse wavelet transform and then each pixel value is divided with a constant. In this way, the reconstruction of the image is produced.

The goal was to eliminate all floating-point operations and divisions, minimize the number of bits in the integer operations, restrict the amount of on-chip memory, remove all redundant information due to the restriction of 512×512 images and eliminate all recursion. All rewritings of the programs are discussed below.

4.1.1 Recursion

EPIC saves the Huffman tree recursively in the image header and this means that a stack would have to be implemented in hardware. This is unnecessary complicated and in order to eliminate recursion, a new structure, called a Huffman table, was developed to reduce complexity.

Each entry in the Huffman table consists of one bit, called the mode bit, plus a word[11]. If the bit is set to one then that word is a symbol value, that is, the entry is an external node. If the mode bit is zero, then that entry is an internal node. The high order byte of that entry is the entry address of the left child and the low order byte is the entry address of the right child. By using this data structure the Huffman tree could be saved without recursion. The only drawback is that a 16 bit entry restricts the number of entries to 256, which means that the Huffman tree may consist of at most 128 external nodes. For a compression ratio greater than 25, 256 entries sufficed for our test images.

The compression program was rewritten to save a Huffman table in the following way; first one byte indicating the number of words, say n, containg mode bits, is saved, followed by one byte, say m, indicating the number of Huffman entries. Then follows n words of mode bits and m words of Huffman entries.

4.1.2 Arithmetics

To reduce the area of the chip and to maximize the clock frequency, the number of bits in the integer operations in the inverse wavelet transform had to be minimized. Originally, these were 32 bits operations. Without causing any image quality degradation, it was concluded, via software verification, that 16 bits operations sufficed.

The only floating-point operation in the original program was the division performed when scaling the image after the inverse wavelet transform. Both the floating-point operation and the division should be eliminated. By replacing

[11] A word is here 16 bits.

it with a integer shift right operation, the image quality deteriorated approximately 0.6 dB. Several tests showed that the human perception system could not distinguish which of the images was compressed with the original program. The shift operator restricts the compressing program only to use scale factors of type 2^n.

4.1.3 Image header

Since the chip only decompresses 512×512 grey scale images, redundant data in the image header could be removed. The image header in the rewritten programs are described here.

The first word contains the original *EPIC* identification tag (`$ff`) and the number of levels in the inverse transform, which in this case forms `$ff05`. The width and the height of the image were removed, since they both are implicitly 512. Then follows a word containing the scale factor used after the inverse wavelet transform and a word containing the scale factor used after runlength decoding. Originally, a scalefactor was saved for each square in in the transformed image, but since they only differed by a factor 2^n this information was removed and only one word was saved. After that follows 256 words of raw data for the low-pass component of the transformed image. Finally, the Huffman table is read and then the compressed data remains.

4.2 Hardware

The hardware implementation is divided into two parts, namely *Huffman and runlength decoding* and the *wavelet transform*. These two parts are described in detail below.

4.2.1 Huffman and runlength decoding

The main task of this part of the chip is to Huffman and runlength decode, but it also reads and stores parts of the image header. A schematic view of the two-phase hardware implementation is depicted in figure 25. Since the image header, the raw data (low pass) and Huffman table occurs first in the file, these things have to be taken care of first. The compressed image data is read from the memory, called *Compressed Picture Memory*, to the left in the figure. Located above this, is a *Picture Counter (PC)*, which holds the address of the current word to be read from the memory. Below is a box called *Address Logic* which computes a number of control signals, which in turn controls the action of the other boxes.

The control signals computed depend on the value of the PC and some constants read from the memory into the *Constants* box.

They are used to determine when to save constants in latches, when to save the Huffman table, when to write the raw data (low pass) to the *Destination Memory* and when to start Huffman and runlength decode.

After two image constants have been stored in latches, the Huffman table is stored. In order to make the Huffman table work correctly, the first entry is stored on address zero and the following entries on the sequential addresses.

When the Huffman table has been stored, only image data remains to be Huffman and runlength decoded. Since it usually takes several clock cycles to Huff-

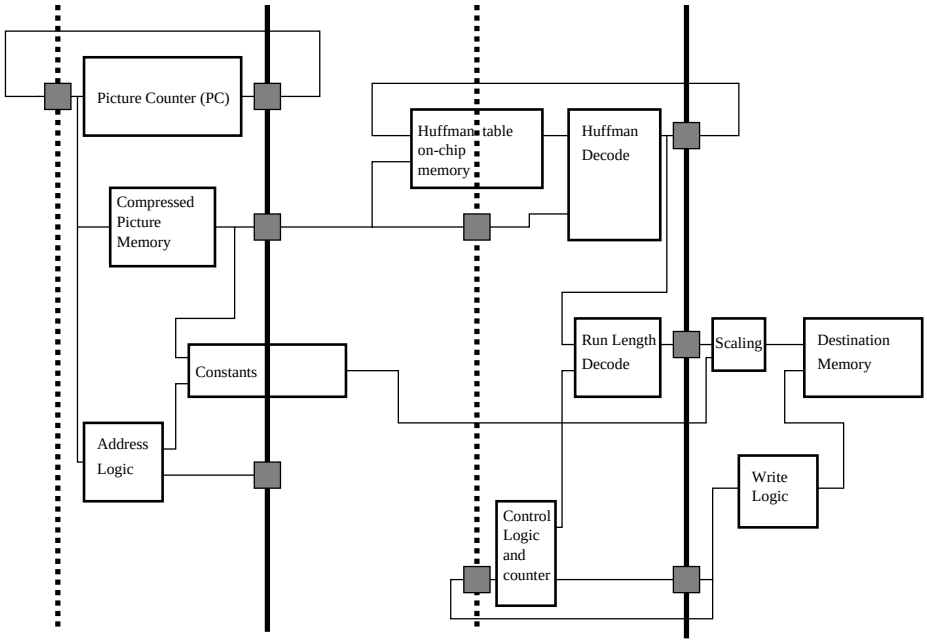

Figure 25: Huffman decoding and runlength decoding.

man decode a symbol, the PC cannot be incremented every clock cycle as before. The problem is solved in the following way. The box called *Control Logic and counter* contains a modulo-16 counter to hold the current bit in the word begin decoded. When this counter reach 15 a control signal is generated, saying that the PC should be incremented.

The box called *Huffman Decode* takes care of all Huffman decoding. A control signal, generated by *Address Logic*, resets this box and the Huffman decoding of the first symbol starts.

The *Control Logic and counter* determines which bit of the word from the memory is to be decoded. If this bit is set, the right child is used as the next entry address and if it is zero, the left child is used. The entry address is feed back into the Huffman table and a new entry is being fetched. This feedback loop continues until the mode bit is set. Then the output from the Huffman table contains the decoded symbol, which is then feed into the *RunLength Decode* box. If the MSB[12] is set, then a runlength symbol has been encountered and all other boxes are being paused. The number of zeros to be written to the destination memory is computed and when these zeros have been written, the paused units start computing again. On the other hand, if the MSB is zero, then an ordinary symbol has been encountered. The SMSB[13] determines the sign of the remaining 14 bits.

Before writing to destination memory the symbol also has to be scaled. The *Scaling* box contains the only multiplier in the construction and depending on the destination address the multiplicand is shifted various steps to the left (see appendix D for VHDL-details).

[12]MSB stands for Most Significant Bit.
[13]Second Most Significant Bit

The *Write Logic* box contains a counter that holds the destination address and this counter is only incremented when the box *RunLength Decode* has a new symbol to be saved.

A real two-phase implementation is shown in appendix C.

4.2.2 Wavelet transform

To understand how to decode a wavelet coded picture, we only have to consider the last step - the other steps are done the same way. The left part of figure 26 consists of four different kinds of coefficients: lowpass coefficients in part L, vertical hightpass in part V, horisontal highpass in part H and diagonal highpass in part D. Every one of these parts contains information for the whole picture. The lowpass coefficients give the basis and the highpass parts add detail to the

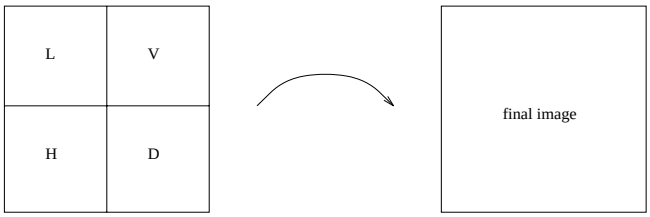

Figure 26: The last iteration in the reconstruction phase.

picture. However, a single coefficient does only affect a small part of the picture, thanks to the time locality of the wavelet transform. In our case with threetap filters, every coefficient affects a three times three square. Squares affected by consecutive coefficients overlap as seen in figure 27. The different kinds of co-

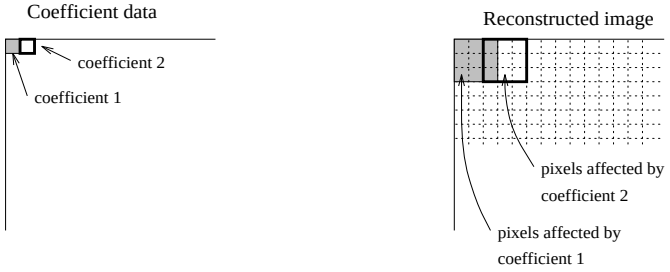

Figure 27: This is the relationship between areas affected by consecutive coefficients.

efficients (L, H, V and D) does not affect the resulting image in the same way. This is for two reasons:
A) Coefficients of different type does not affect the same square. For example, the first L coefficient does not affect the same square as the first D coefficients. The relationship between the two squares are shown in figure 28.

B) The different kinds of coefficients (L, H, V and D) does not affect its square the same way, but according to the following matrices.

$$L = \frac{1}{4}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

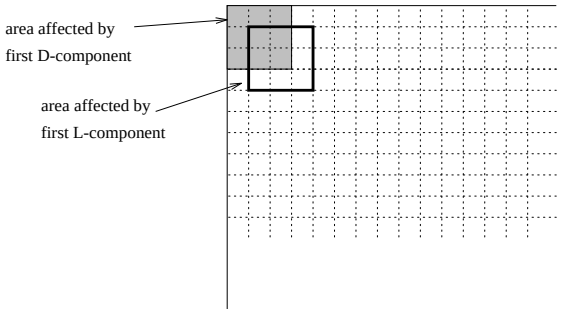

Figure 28: This is the relationship between the areas affected by the first L and D coefficients.

$$H = \frac{1}{4}\begin{pmatrix} -1 & -2 & -1 \\ 2 & 4 & 2 \\ -1 & -2 & -1 \end{pmatrix}$$

$$V = \frac{1}{4}\begin{pmatrix} -1 & 2 & -1 \\ -2 & 4 & -2 \\ -1 & 2 & -1 \end{pmatrix}$$

$$D = \frac{1}{4}\begin{pmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{pmatrix}$$

This means for example that $\frac{1}{4}$ of the value of the L-coefficient is added to the upper left pixel in the L-square, whereas the same pixel in a V-square would get $-\frac{1}{4}$ of the value of the V-coefficient.

In the EPIC program, everything is done sequentially. First the lowpass coefficients are spread out using the L-matrix, then the detail is superimposed using the H-matrix for the H data, V-matrix for the V data etc.

Before starting to develop the hardware algorithm, we had to do a feasibility study. It did not take long before we had understood that the bottelneck was the memory bandwidth. If we strictly followed the algorithm above, the last iteration step (from 256×256 to 512×512) would require a huge memory acceses:

- 512^2 writes in order to zero the resulting image
- 4×256^2 to read the l, h, v and d coefficients
- $9 \times 4 \times 256^2$ writes to the results, because each of the above coefficients spreads its data with the 3×3 spreading matrix.
- $9 \times 4 \times 256^2$ reads from the results to be able to superimpose the data above.

This sums up to 5242880 memory acesses for the last iteration only. To perform that amount of memory operations 25 times a second we would need a memory with a latency time of $\frac{1}{25 \times 5242880} = 7$ns. Even if we had two separate memories,

one for the coefficients and the other for the result, the latter memory would have to do 25×4718592 memory operations a second and thus need a latency time of 8ns. Notice that this is using full pipelining, i.e. with all five iterations being made simontaneously. As a comparison, normal cache memory today have a latency time of about 20ns. Such a construction might be possible to build but very expensive.

The solution to the memory bandwidth problems is to parallellize the problem. By calculating all four spreading matrixes at once it is no longer neccesary to read and write back the results in order to superpose the between the different coefficients L, H, V and D. After new spreading matrix is now used:

$$S = \frac{1}{4}\begin{pmatrix} d & -2d-h & d-2h & -h \\ -2d-v & 4d+l+2h+2v & -2d+2l+4h-v & l+2h \\ d-2v & -2d+2l-h+4v & d+4l-2h-2v & 2l-h \\ -v & l+2v & 2l-v & l \end{pmatrix} + R$$

If you scrutinize it, you will see that it consists of the L, H, V and D matrices superimposed. The D matrix will be found in the upper left part, the H matrix in the upper right, whereas the V and L matrices will be found in the lower left and right part respectively. This matrix is moved in steps of two over the resluting picture, and since the matrix is 4 pixels wide two neigbour matrixes overlap. The information in the overlapping pixels must be superimposed to the next spreading matrix. This comes in as the matrix R in the figure. The nice thing is that the left part of the R matrix (i.e. elements r_{11}, r_{12}, r_{21}, r_{22}, r_{31}, r_{32}, r_{41} and r_{42}) can be received from the previous spreading matrix, and can hence be stored in latches. The upper right four elements (i.e. r_{13}, r_{14}, r_{23}, and r_{24}) derivates from the previous row. A special on-chip memory, the row memory, gives us these values. The remaining part of the R matrix is zero.

After having performed the calculation, we must store the values. The right half of the matrix is stored in latches for use by the next spreading matrix. The lower left part is stored in the on-chip row memory. The 4 upper left elements of the matrix will not need further processing and can be sent to the resulting image. The number of reads and writes for the last iteration (512×512pixels) will be:

- 4×256^2 reads for the l, h, v and d coefficients
- 4×256^2 reads from the row memory
- 4×256^2 writes to the row memory
- 4×256^2 writes to the final image

The coefficient reading and the final image writing does not go to the same memory. If we pipeline the process with the three stages read, calculate and write, the writing from a block two stages earlier can be done at the same time as the read. If we assume the on chip row memory to be at least twice as fast as the off chip memory, the number of sequencial memory accesses become just 4×256^2 for the whole iteration. To save memory, all five iterations is done in sequence. The last iterations is the biggest one, the one before that is only one fourth of that. Together all five iterations require the following memory operations in sequence:

- 4×257^2 For the last iteration

- 4×129^2 for the 4th iteration
- 4×65^2 for the 3rd iteration
- 4×33^2 for the 2nd iteration
- 4×17^2 for the 1st iteration
- 4×128^2 for copying

You may have noticed that we have used 257 instead of 256, 129 instead of 128 and so on. This is because of border phenomena. To get the correct values, the algorithm must pretend the coefficents (l, h, v and d) keeps the same values as before the border. We then have to do one extra matrix spreading operation per row, and one extra row. We also need to copy some values, because the first iteration reads in memory A and stores the reconstructed lowpass coefficients in memory B. In the next iteration the coefficients are read from B and stored in A. Thus we must copy the highpass information from A to B in order to be able to read it from there. All copying is done before the first iteration begin.

The sum of all these memory accesses will be 418708 memory accesses per picture or $25 \times 418708 = 10467700$ memory accesses per second which means a memory latency time of 95ns. As a comparison, ordinary 486 66MHz PC:s are shipped with DRAMS of 70ns.

The wavelet inverse tranform is implemented as a set of address generators that accesses the source-, destination- and rowmemories. After the data has been obtained, it is processed in a component called black box where the calculation takes place. The results is the feed back in latches or stored in row- and destination memories. To control the adress generators there are three counters: One xcounter, one ycounter and one iteration counter. These counters are variable, the xcounter counts modulo 17, 33, 65, 129 and 257 using muxes coupled to the iteration counter output. To get time for interchangeing memory, the ycounter counts modulo 18, 34 etc.

The wavelet inverese transform needs 512Kbytes source memory and an equal amount of destination memory. To allow the Huffman and runlength decoder to run in parallel with the inverse wavelet transform, an additional megabyte of double buffering memory must be used.

5 Results and Conclusions

During a seven-week course, all software have been rewritten and NewWave has been built by the authors. The hardware implementation, that is NewWave, has been built, verified by simulation and synthesized.

We have shown that the wavelet technique is the best compression method, at least to our knowledge, and that it is useable also for making hardware.

Hopefully a standard for compressing images with wavelets will soon be established.

5.1 Technical Information

NewWave was developed as two constructions, the *decoding* part and the *transform* part, running at different clock speeds. The NewWave technical information is presented below.

- NewWave decompresses 25 images per second.
- NewWave only decompresses 512×512 greyscale images.
- The minimum compression ratio is ≈ 25.
- The decoding part runs at 8.6 MHz
- The transform part runs at 10.5 MHz.
- The decoding part occupies 5 mm^2+512 bytes on-chip memory in ams_cyb technology.
- The transform part occupies 28.57 mm^2 + 2048 bytes on-chip memory in ams_cyb technology.
- All memory modules are clocked with 95 ns, which allows us to use DRAM technology.

6 Acknowledgements

The authors want to thank our supervisor Björn Breidegard for support and for believing in us and Sven Spanne och Gunnar Sparr for providing us with litterature on wavelets.

A Tools

The tools used to develop, evaluate and simulate the hardware are described in this section.

A.1 BBDS

BBDS[14] is a design tool for architectural evaluation and rapid prototyping of performance critical systems. Using BBDS and its interactive graphical interface, a clock cycle true model can be developed. A design idea is quickly entered into the tool using a well defined visualization called a Werner diagram[15], and can be verified by simulation, timing analysis, area estimation and synthesis. This facilitates explorative design with fast iteration time, which means that the design space can be explored quickly.

Standard components, such as MUX:es, decoders, adders, comparators, latches, etc are all part of BBDS. Components not included can be written in VHDL, described below, and incorporated into BBDS. Since it is based on automatic synthesis with a user selected target library, BBDS is technology independent.

A.2 VHDL

VHDL is a description lanuguage for digital electronic systems. It stands for VHSIC Hardware Description Language, where VHSIC stands for Very High Speed Integrated Circuits.

B Coding and decoding

EPIC uses Huffman and runlength coding to compress the wavelet transformed image. Huffman and runlength decoding was used in the implementation of the wavelet decompression algorithm. Therefor these techniques are described below.

B.1 Huffman's algorithm

Given a finite set S of symbols, and a finite string of symbols from S, *the optimal encoding problem* is stated as follows:

> *Find an unambiguous binary encoding for S that minimizes the number of bits in the encoded string.*

Assume n symbols is to be encoded and w_i is the number of times symbol i occurs in the string. If d_i is the number of bits for the encoded symbol i, then the total length of the encoded string, L, should be minimized.

$$L = \sum_{i=1}^{n} w_i d_i$$

It can be shown, see [Kingston90], that Huffman's algorithm minimizes L, using a Huffman tree (a kind of binary tree).

[14] Developed at the Department for Computer Engineering at Lund Institute of Technology.
[15] Werner diagram, also a principle developed at the Department of Computer Engineering, Lund Institute of Technology.

Throughout the construction of the tree, a forest (a set of trees) is used. Initially the forest consists entirely of external nodes, one for each symbol with an associated weight. The root weight of each tree is the sum of the external nodes' weights, w_i. Combining two trees of minimum root weight into one until only one tree is left, results in the Huffman tree. The encoded symbols is derived by traversing the tree from the root downto each external node and giving the current symbol a zero for a left turn and a one for a right turn. Huffman coding is obviously a variable length coding algorithm. A demonstration of the construction is shown in figureOB 29.

External nodes are drawn as a boxes and internal nodes as a circles.

To code a string of symbols, first build the Huffman tree and then simply

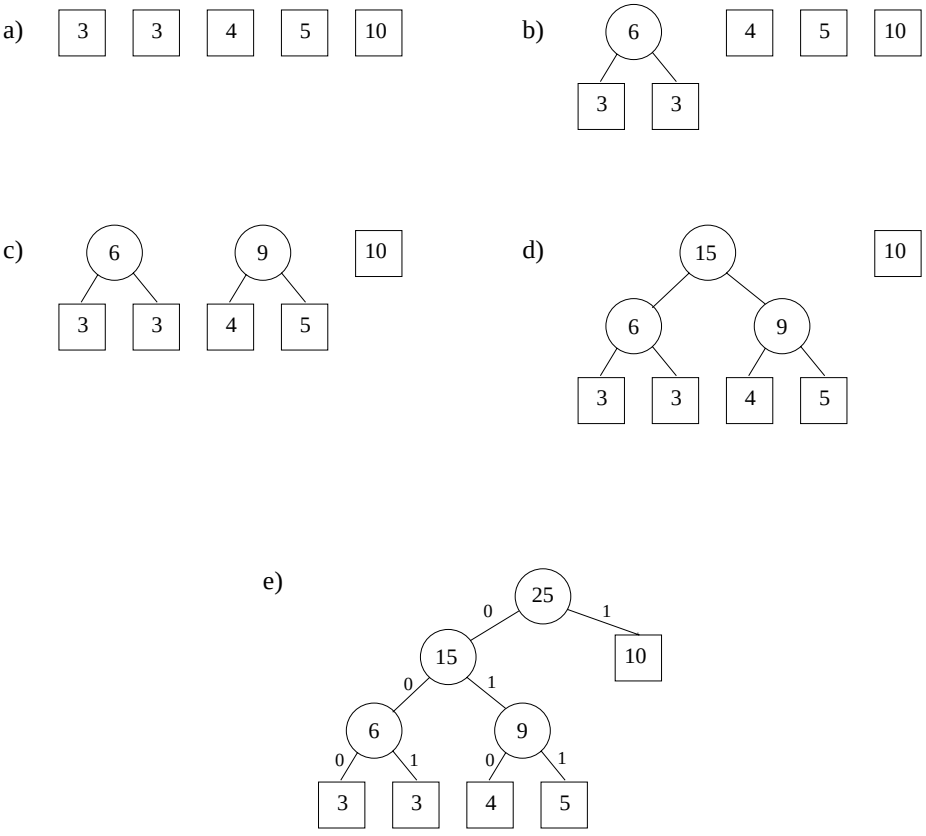

Figure 29: Example of construction of a Huffman tree. External nodes are drawn as boxes and internal nodes as circles. The number in the boxes are the number of occurences of the symbol associated with that box. The symbols are not shown.

replace all the symbols with the encoded symbols. The information needed to reconstruct the data is the Huffman tree (together with the original symbols' values)and the encoded string of symbols. Since the coding is unambiguous (no code can be a prefix of another), decoding of a symbol starts at the root of the tree and takes a right turn for one and a left turn for a zero. When the traversion reaches an external node, the encoded symbol is replaced with the original symbol value at the node.

B.2 Runlength encoding

The main advantage of transforming an image before coding it, is that a lot of image components transforms to zero. As the name implies, runlength encoding takes a run of equal symbols and and encodes them as one copy of the symbol together with a number indicating the length of the run. For example, {0 0 0 0 0 0 0 0 0 0} could be encoded as {0 9}. EPIC only uses runlength encoding for the zeros and thus the first symbol could be eliminated. To distinguish between normal symbols and runlengths, the MSB of a 16 bit word is set to one if the remaining 15 bits indicate the number of zeros of a runlength and it is set to zero if they indicate a normal symbol value.

C Block schemes

Presented below are some of the two-phase block schemes developed for NewWave.

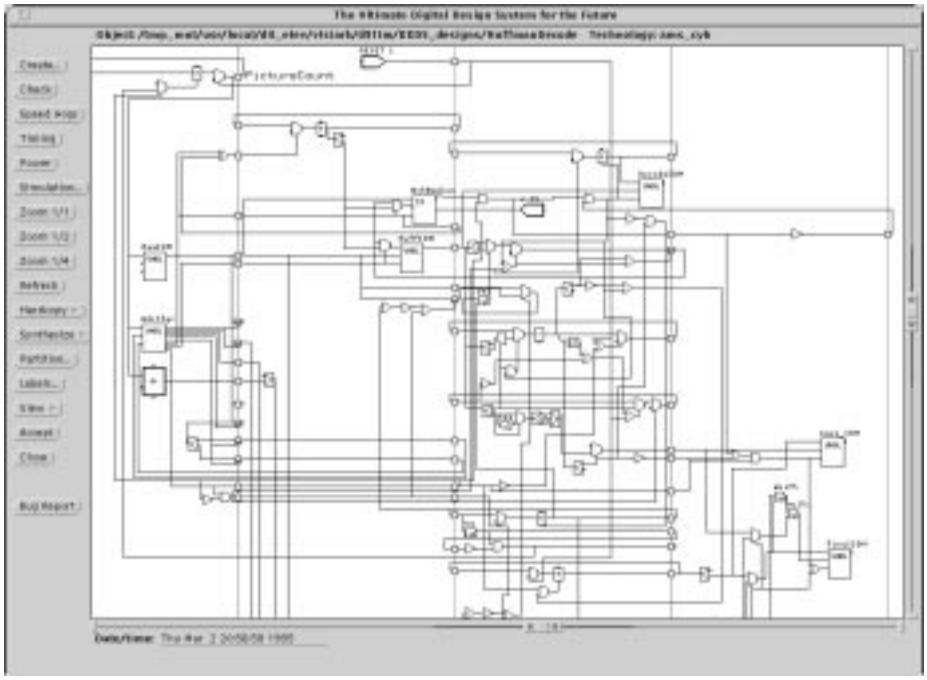

Figure 30: Huffman decoding and runlength decoding block scheme from BBDS

D VHDL-code

In this section VHDL-code for the non standard components are shown.

```
FILE: adrlogik.hdl

LIBRARY IEEE;
USE     IEEE.std_logic_1164.all;
LIBRARY ARITHMETIC;
USE ARITHMETIC.std_logic_arith.all;

entity adrlogik is
```

```
  port(
    addr        : in  std_logic_vector(12 downto 0);
    bitstrlen   : in  std_logic_vector( 7 downto 0);
    huffmanlen  : in  std_logic_vector( 7 downto 0);
    scalefactor : out std_logic_vector( 0 downto 0);
    binsize0    : out std_logic_vector( 0 downto 0);
    rawdata     : out std_logic_vector( 0 downto 0);
    symblen     : out std_logic_vector( 0 downto 0);
    hufflen     : out std_logic_vector( 0 downto 0);
    huffbitsave : out std_logic_vector( 0 downto 0);
    hufftabsave : out std_logic_vector( 0 downto 0);
    huffdecode  : out std_logic_vector( 0 downto 0)
  );
end adrlogik;

architecture BBDS_SIM of adrlogik is
begin
  process(addr,bitstrlen,huffmanlen)
  variable address,bitadd,huffadd:INTEGER;
  begin
    scalefactor<="0";
    binsize0<="0";
    rawdata<="0";
    symblen<="0";
    hufflen<="0";
    huffbitsave<="0";
    hufftabsave<="0";
    huffdecode<="0";
    address:=TO_INTEGER(addr);
    bitadd:=TO_INTEGER(bitstrlen);
    huffadd:=TO_INTEGER(huffmanlen);
    if(address=0) then  -- dummy if
        huffdecode<="0";
    elsif(address=1) then
        scalefactor<="1";
    elsif(address=2) then
        binsize0<="1";
    elsif(address>2) and (address<259) then
        rawdata<="1";
    elsif(address=259) then
symblen<="1";
    elsif(address=260) then
        hufflen<="1";
    elsif(address>260) and (address<261+bitadd) then
        huffbitsave<="1";
    elsif(address>260+bitadd) and (address<261+bitadd+huffadd) then
        hufftabsave<="1";
    else
        huffdecode<="1";
    end if;
  end process;
end BBDS_SIM;



FILE: decoder.hdl

LIBRARY IEEE;
USE     IEEE.std_logic_1164.all;
```

```
LIBRARY ARITHMETIC;
USE ARITHMETIC.std_logic_arith.all;

entity decoder is
  port(
    tal     : in  std_logic_vector(15 downto 0);
    bitnr   : in  std_logic_vector( 3 downto 0);
    bit     : out std_logic_vector( 0 downto 0)
  );
end decoder;

architecture BBDS_SIM of decoder is
begin
  process(tal,bitnr)
  variable index:integer;
  begin
    index:=TO_INTEGER(bitnr);
    if(index<8) then
index:=index+8;
    else
        index:=index-8;
    end if;
    bit(0)<=tal(index);
  end process;
end BBDS_SIM;
```

FILE: multi.hdl

```
LIBRARY IEEE;
USE     IEEE.std_logic_1164.all;
LIBRARY ARITHMETIC;
USE ARITHMETIC.std_logic_arith.all;

entity multi is
  port(
    multtal   : in  std_logic_vector(15 downto 0);
    tal       : in  std_logic_vector(15 downto 0);
    result    : out std_logic_vector(15 downto 0)
      );
end multi;

architecture BBDS_SIM of multi is
begin
  process(multtal,tal)
  variable tmp,ttal:integer RANGE -32768 to 32767;
  begin
     ttal:=TO_INTEGER(tal);
     tmp:=TO_INTEGER(multtal);
     result<=to_stdlogicvector(tmp*ttal,16);
  end process;
end BBDS_SIM;
```

FILE: mux256.hdl

```
LIBRARY IEEE;
```

```
USE     IEEE.std_logic_1164.all;
LIBRARY ARITHMETIC;
USE ARITHMETIC.std_logic_arith.all;
USE     std.textio.all;

entity mux256 is
  port(
    ED15   : in  std_logic_vector(15 downto 0);
    ED14   : in  std_logic_vector(15 downto 0);
    ED13   : in  std_logic_vector(15 downto 0);
    ED12   : in  std_logic_vector(15 downto 0);
    ED11   : in  std_logic_vector(15 downto 0);
    ED10   : in  std_logic_vector(15 downto 0);
    ED9    : in  std_logic_vector(15 downto 0);
    ED8    : in  std_logic_vector(15 downto 0);
    ED7    : in  std_logic_vector(15 downto 0);
    ED6    : in  std_logic_vector(15 downto 0);
    ED5    : in  std_logic_vector(15 downto 0);
    ED4    : in  std_logic_vector(15 downto 0);
    ED3    : in  std_logic_vector(15 downto 0);
    ED2    : in  std_logic_vector(15 downto 0);
    ED1    : in  std_logic_vector(15 downto 0);
    ED0    : in  std_logic_vector(15 downto 0);
    adr   : in  std_logic_vector( 7 downto 0);
    outbit: out std_logic_vector( 0 downto 0);
    word: out std_logic_vector( 15 downto 0)
  );
end mux256;

architecture BBDS_SIM of mux256 is
begin
  process(ED15,ED14,ED13,ED12,ED11,ED10,ED9,ED8,ED7,ED6,ED5,ED4,ED3,ED2,ED1,ED0,adr)
  variable tmp: std_logic_vector(15 downto 0);
  variable tadr,x: integer;
  begin
     outbit<="0";
     tadr:=TO_INTEGER(('0' &adr) and "011110000");
      case tadr is
  when 0 =>
          tmp:=ED0;
  when 16 =>
          tmp:=ED1;
  when 32 =>
          tmp:=ED2;
  when 48 =>
          tmp:=ED3;
  when 64 =>
          tmp:=ED4;
  when 80 =>
          tmp:=ED5;
  when 96 =>
          tmp:=ED6;
  when 112 =>
          tmp:=ED7;
  when 128 =>
          tmp:=ED8;
  when 144 =>
          tmp:=ED9;
```

```
  when 160 =>
          tmp:=ED10;
  when 176 =>
          tmp:=ED11;
  when 192 =>
          tmp:=ED12;
  when 208 =>
          tmp:=ED13;
  when 224 =>
          tmp:=ED14;
  when 240 =>
          tmp:=ED15;
        when others =>
          tmp:="0000000000000000";
     end case;
     x:=TO_INTEGER(adr and "00001111");
     if(x<8) then
x:=x+8;
     else
x:=x-8;
     end if;
     outbit(0)<=tmp(x);
     word<=tmp;
  end process;
end BBDS_SIM;
```

FILE: neg.hdl

```
LIBRARY IEEE;
USE     IEEE.std_logic_1164.all;
LIBRARY ARITHMETIC;
USE ARITHMETIC.std_logic_arith.all;

entity neg is
  port(
    tal      : in  std_logic_vector(13 downto 0);
    negtal   : out std_logic_vector(13  downto 0)
  );
end neg;

architecture BBDS_SIM of neg is
begin
  process(tal)
  variable x:integer;
  begin
    x:=-TO_INTEGER(tal);
    negtal<=to_stdlogicvector(x,14);
  end process;
end BBDS_SIM;
```

FILE: shift.hdl

```
LIBRARY IEEE;
USE     IEEE.std_logic_1164.all;
LIBRARY ARITHMETIC;
```

```
USE ARITHMETIC.std_logic_arith.all;

entity shift is
  port(
    multtal   : in  std_logic_vector(15 downto 0);
    adr       : in  std_logic_vector(17 downto 0);
    result    : out std_logic_vector(15 downto 0)
      );
end shift;

architecture BBDS_SIM of shift is
begin
  process(multtal,adr)
  variable tadr:integer RANGE -2**18 to 2**18-1;
  variable tmp:std_logic_vector(19 downto 0);
  begin
     tadr:=TO_INTEGER(adr);
     if(tadr<1024) then
tmp:="0000" & multtal;
-- behåll mult-faktorn
     elsif(tadr<4096) then
tmp:="000" & multtal & '0';
     elsif(tadr<16384) then
tmp:="00" & multtal & "00";
     elsif(tadr<65536) then
tmp:='0' & multtal & "000";
     else
tmp:=multtal & "0000";
     end if;
     result<=tmp(15 downto 0);
  end process;
end BBDS_SIM;
```

References

[Adelson90] Edward H. Adelson and Eero P. Simoncelli. *Subband Image Coding with Three-tap Pyramids*, MIT Media Laboratory. Cambridge, MA. Picture Coding Symposium 1990

[Fournier94] Alan Fournier. *Wavelets and their Applications in Computer Graphics*, SIGGRAPH'94 Course Notes, 1994

[Fridman94a] Jose Fridman and Elias S. Manolakos. *On the synthesis of regular VLSI architectures for the 1-D discrete wavelet transform*, Proceedings of SPIE Conf. on Mathematical Imaging: Wavelet Applications in Signal and Image Processing II, San Diego CA, July 1994

[Fridman94b] Jose Fridman and Elias S. Manolakos. *Distributed Memory and Control VLSI Architectures for the 1-D Discrete Wavelet Transform*, IEEE VLSI Signal Processing VII, 10/1994

[Hilton91] Michael L. Hilton, Bjorn D. Jawerth and Ayan Sengupta. *Compressing Still and Moving Images with Wavelets*, Multimedia Systems, Vol. 2, No. 3, 1994

[Kingston90] Jeffrey H. Kingston. *Algorithms and Data Structures*, Addison-Wesley publishing company, 1990, Sydney.

[Strom94] Jacob Ström. *Bildkomprimering med JPEG, Fraktaler och Krusningar (Wavelets)*, Technical report, Lund Institute of Technology, 1994, Lund.