Development and Implementation of an *MotionJPEG* Capable *JPEG* Decoder in Hardware

The *JPEG* standard (*ISO/ IEC 10918-1 ITU-T Recommendation T.81*) defines compression techniques for image data. As a consequence, it allows to store and transfer image data with considerably reduced demand for storage space and bandwidth. From the four processes provided in the *JPEG* standard, only one, the *baseline process* is widely used.

In this thesis a hardware based system to decode *JPEG baseline* compressed image data is presented. The different stages of the decoding process are implemented in a pipelined design described in VHDL. Running on a *Virtex-II Pro* FPGA at 100 MHz operation frequency, the performance and accuracy of the hardware decoder is comparable to a software decoding system running on a 1500 MHz commodity PC. The pipelined structure allows for the processing of multiple image blocks simultanously. Thus, the decoder is especially suited to decode *MotionJPEG* movies. Functionality of the system is demonstrated with a hardware *MotionJPEG* video player application.

Entwicklung und Realisierung eines *MotionJPEG* fähigen *JPEG* Dekoders in Hardware

Der *JPEG* Standard (*ISO/IEC 10918-1 ITU-T Recommendation T.81*) definiert Verfahren zur Kompression von Bilddaten. Der *JPEG* Algorithmus ermöglicht es folglich, Bilddaten mit deutlich reduziertem Anspruch an Speicherplatz und Bandbreite zu archvieren und zu übertragen. Der Standard stellt vier verschiedene Prozesse zur Komprimierung von Bilddaten zur Verfügung, von denen sich allerdings nur einer, der *baseline process*, in der Praxis durchgesetzt hat.

Im Rahmen dieser Diplomarbeit wurde ein hardware-basiertes System zur Dekodierung von mit dem *JPEG baseline process* komprimierten Bilddaten entwickelt. Dabei wurde der mehrstufige Prozess der Dekodierung in einem gepipelineten Design realisiert. Auf einem mit 100 MHz getakteten *Virtex-II Pro* FPGA wurden damit vergleichbare Ergebnisse erzielt wie mit einem Softwaredekoder, welcher auf einem mit 1500 MHz getakteten Standardprozessor ausgeführt wurde. Das Design der Pipeline ist speziell dafür ausgelegt, mehrere Bildblöcke gleichzeitig zu bearbeiten. Damit ist der Dekoder bestens zum Dekodieren von *MotionJPEG* Filmen geeignet. Die Funktionstüchtigkeit des System wurde mit einer hardware-basierten *MotionJPEG* Video Player Applikation unter Beweis gestellt.

Contents

1 *JPEG* Compression Overview

The frequently-used term *JPEG* is an acronym for *Joint Photographic Experts Group*[1], a joint committee between ISO [2] and ITU-T [3], who specified the *JPEG* standard in 1992.

The *JPEG* compression standard [Jpe92] defines techniques to compress, decompress, and store image data. Covering a wide field of applications it is a complex venture to implement everything provided in the *JPEG* standard. Therefore the committee defines a set of coding processes, each using their own subset of the *JPEG* technologies: *Baseline process*, *Extended DCT-based process*, *Lossless process* and *Hierarchical process* (cp. table 1.1).

Besides the the *Lossless process* all of the processes implement so called *"lossy compression"*. This means that non-redundant information is removed from the data and the original image cannot be restored exactly form the recorded data. However, done right, the uncompressed image will appear very similar to the original image.

JPEG compression works very well for continuous tone images like photographs of a natural scene. For images with many sharp edges and bigger areas of exactly the same color, for example computer generated diagrams, other compression techniques like those developed for *"GIF"*[4] usually result in better compression rates although being lossless.

The most widely used process is the *Baseline process*. Most modern video compression techniques in common use (like MJPEG, MPEG-1/2) base heavily on the *JPEG baseline process* as well.

The file format widely used - usually with the suffix `.jpg` - has its own standard called *JFIF*[5] [Ham92], which comes with further restrictions. *JFIF* is compatible with the official *JPEG* specification, but not a part of it.

Generally *JPEG* refers to a *JFIF*-file storing image data compressed according to the *JPEG baseline process*. Since the decoder which has been implemented in this thesis decodes these kind of images, this chapter gives an overview of the *JPEG baseline process*.

Figure 1.1 shows and describes the essential steps in baseline *JPEG* encoding, a detailed description of the different steps is given in the following paragraphs.

[1] The official name is *ISO/IEC Joint Technical Committee 1, Subcommittee 29, Working Group 1*
[2] International Organization for Standardization
[3] International Telecommunication Union, formerly CCITT (Comité Consultatif International Téléphonique et Télégraphique)
[4] Graphics Interchange Format
[5] JPEG File Interchange Format

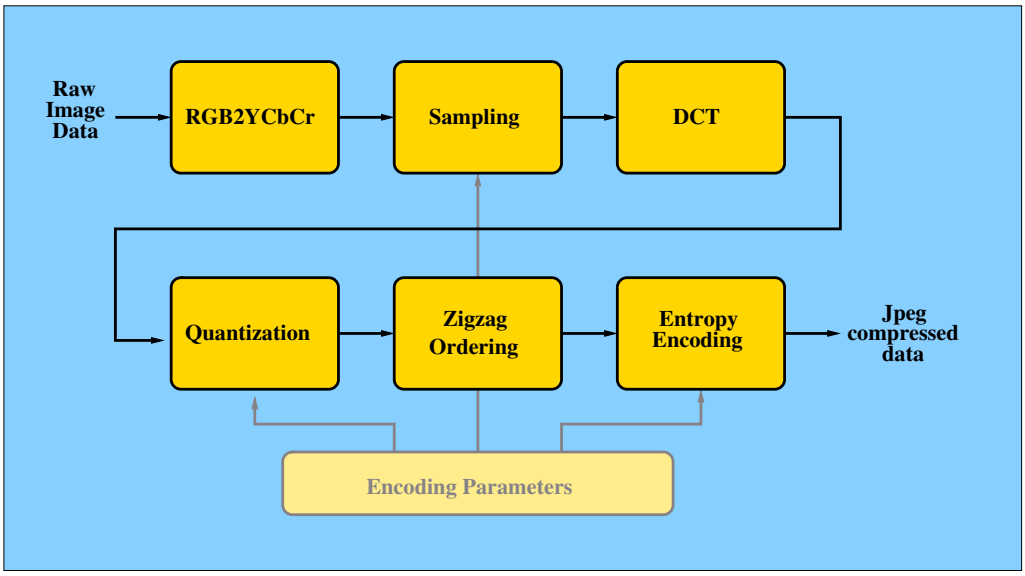


Figure 1.1: The essential steps in *JPEG* encoding (*baseline process*). First the image is transformed to the YCbCr color mode separating luma (Y) from chroma (Cb/Cr) information `[RGB2YCbCr]`. Then the color components are reduced in spatial resolution `[Sampling]`. Applying the *Discrete Cosine Transformation (DCT)* the blocks are mapped to frequency space `[DCT]` where the higher frequencies can now be removed `[Quantization]`. After reordering the remaining coefficients `[Zigzag-Ordering]` the resulting bitstream is then very well prepared for entropy encoding using run length encoding and an Huffman algorithm `[Entropy Encoding]`.

To understand the principles of *JPEG* technologies it is more intuitive to take a look at the steps of **en**coding rather than **de**coding. Therefore, despite the fact that a decoder has been developed, due to better understanding this chapter will explain the steps of encoding. The steps of decoding will be the inverse of the encoding steps but in reverse order.

1.1 *JFIF* - Structure of the Header

The way header information is stored in a *JPEG* file is presented in the *JFIF* standard and the Annex B of the *JPEG* standard. As mentioned before the *JFIF* standard specifies a subset of techniques from the *JPEG* standard and additionally has its own (*JPEG* compatible) restrictions.

To be *JFIF* compatible the image components need to be Y, Cb & Cr for color images and just Y for grayscale images (cp. section 1.2).

Baseline	**Extended DCT-based**
• DCT-based process • Source image: 8-bit samples within each component • Sequential • Huffman coding: 2 AC and 2 DC tables • Decoders shall process scans with 1, 2, 3, and 4 components • Interleaved and non-interleaved scans	• DCT-based process • Source image: 8-bit or 12-bit samples • Sequential or progressive • Huffman or arithmetic coding: 4 AC and 4 DC tables • Decoders shall process scans with 1, 2, 3, and 4 components • Interleaved and non-interleaved scans
Lossless	**Hierarchical**
• Predictive process (not DCT-based) • Source image: P-bit samples ($2 \leq P \leq 16$) • Sequential • Huffman or arithmetic coding: 4 DC tables • Decoders shall process scans with 1, 2, 3, and 4 components • Interleaved and non-interleaved scans	• Multiple frames (non-differential and differential) • Uses extended DCT-based or lossless processes • Decoders shall process scans with 1, 2, 3, and 4 components • Interleaved and non-interleaved scans

Table 1.1: The essential characteristics of the four processes suggested in the *JPEG*-specification [Jpe92]. Only the baseline process is commonly in use.

```
ff d8 ff e0 00 10 4a 46 49 46 00 01 01 01 00 48
00 48 00 00 ff db 00 43 00 0a 07 07 08 07 06 0a
08 08 08 0b 0a 0a 0b 0e 18 10 0e 0d 0d 0e 1d 15
16 11 18 23 1f 25 24 22 1f 22 21 26 2b 37 2f 26
29 34 29 21 22 30 41 31 34 39 3b 3e 3e 3e 25 2e
44 49 43 3c 48 37 3d 3e 3b ff db 00 43 01 0a 0b
0b 0e 0d 0e 1c 10 10 1c 3b 28 22 28 3b 3b 3b 3b
3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b
3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b
3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b 3b ff fe
00 11 08 01 e0 02 80 03 01 22 00 02 11 01 03 11
01 ff c4 00 1a 00 00 03 01 01 01 01 00 00 00 00
00 00 00 00 00 00 00 01 02 03 04 05 06 ff c4 00
18 01 01 01 01 01 01 00 00 00 00 00 00 00 00 00
```

Table 1.2: Part of the header of some *JPEG* file; in hexadecimal representation.

The header is structured using two-byte codes, in *JPEG* terminology called *markers*. A marker starts with the byte *0xFF* followed by an identification byte. A marker may carry some payload; marker and payload together are called *marker segment*. The payload always starts with two bytes giving the length of the payload (excluding the marker but including the two length bytes).

The following list sums up the important information stored in the header.

- Width and height of the image[6]
- Number of components (grayscale or YCbCr color)
- How the components are sampled (gray, 4:2:0, 4:2:2 or 4:4:4).
- Quantization Tables.
- Huffman Tables.

Table 1.2 shows a part of some *JPEG*-file header visualizing the different marker segments by using different colors. A table of important markers is given in the appendix A. For further details about the markers refer to Annex B in the *JPEG* standard [Jpe92].

[6]Note that the number of compressed pixels may be greater than $width \times height$. For technical reasons pixels may be appended at the right and at the bottom side of the image. (cp. section 1.3)

1.2 RGB2YCbCr

Image data is usually represented by giving the red, green and blue component of each pixel (RGB color model). The *JFIF* file format however requires that the data to be compressed is coded in the YCbCr color model and it even specifies how to convert to and from RGB (equations 1.1 and 1.2). This transformation is based on a transformation specified in *CCIR Rec.601* [CCI82] and has been modified slightly to better suit the *JPEG* requirements.

$$\begin{array}{rcrcrcrcl} Y &=& 0.299 \cdot R &+& 0.587 \cdot G &+& 0.114 \cdot B & & \\ Cb &=& -0.1687 \cdot R &-& 0.3313 \cdot G &+& 0.5 \cdot B &+& 128 \\ Cr &=& 0.5 \cdot R &-& 0.4187 \cdot G &-& 0.0813 \cdot B &+& 128 \end{array} \tag{1.1}$$

$$\begin{array}{rcrcrcrcl} R &=& Y & & & & +\ 1.402 \cdot (Cr - 128) & & \\ G &=& Y &-& 0.34414 \cdot (Cb - 128) &-& 0.71414 \cdot (Cr - 128) & & \\ B &=& Y &+& 1.772 \cdot (Cb - 128) & & & & \end{array} \tag{1.2}$$

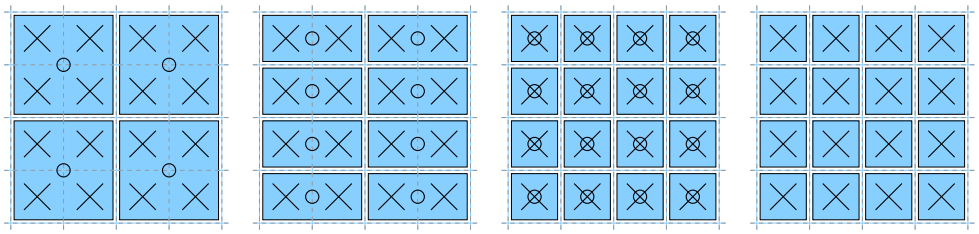
Figure 1.2 illustrates how a picture is composed of its RGB- and its of its YCbCr-components. In the YCbCr color model the appearance of a pixel is given by its *brightness* (Y), its "*blueness*" (Cb) and its "*redness*" (Cr). Green is represented by presence of brightness and absence of red and blue[7] . The Y component (called "luminance") is a weighted sum of the R, G and B components with the weights chosen to represent the intensity of an RGB color. The human eye is most sensitive in the green component, followed by the red component and at last the blue component. The Cb and Cr components (called "chrominance") indicate how much blue and red respectively is in that color.

128 is added to the chrominance components to have a value range from 0 to 255 instead of -128 to 127 (two's complement). Since the Discrete Cosine Transformation needs two's complement signed values as input data, 128 will be subtracted from all values (including Y). This seems quite odd since one could include this step in formula 1.1 in the first place. Most literature explains this in two steps, probably to keep the similarity to the Y, Cb and Cr values as they are defined in *CCIR Rec.601* [CCI82].

The separation of brightness and color makes it possible to handle them differently, as needed by the sampling step.

[7] Be aware that the RGB and the YCbCr color models only define abstract units (R, G, B and Y, Cb, Cr respectively) and have no correlation to actual colors. Making the picture visible as intended, like on a monitor or a printer, additional reference points to actual colors have to be defined. More information about color representation and perception can be found in [Sch03, chapter 2.3] and [Wat99, chapter 5].

Figure 1.2: The famous *Lenna* image decomposed; on the left side in RGB components, on the right side in YCbCr components. Note how the Y component appears to be much richer in detail than the Cb and Cr components. The human eye is more sensitive in perceiving differences in brightness than differences in color.



(a) 4:2:0 (b) 4:2:2 (c) 4:4:4 (d) gray

Figure 1.3: There are several different sampling methods, the most important ones are visualized here. The $\times$ show the position where the luma component is taken, the $\circ$ the position of the chroma components. While the dotted lines outline the pixels of the image, the boxes indicate the size and position of the MCUs.

1.3 Sampling

When examining Figure 1.2 it seems that while R, G, and B components show about the same level of detail, the Y component appears to be much richer in detail than the Cb and Cr components. The human eye is more precise when detecting changes in brightness than changes in color. Storing the luma component in full resolution and the chroma components in reduced resolution may save up to 50% of the data at almost no loss in visual perception quality.

Most *JPEG* files reduce the chrominance components to half of the resolution in both dimensions by taking the mean value of each 2x2 block. This sampling method is called "4:2:0"[8]. Another sampling method evolved form analog television signals is "4:2:2" where chrominance components are reduced only in the horizontal dimension [9]. This sampling method seems obsolete nowadays but it is still of importance since the MPEG-2 standard (as found for example on DVDs) still uses it. For completeness the "4:4:4" method should be mentioned; it does not reduce any component's resolution. For gray-scale images only the Y component is processed. Figure 1.3 illustrates the described sampling methods.

If the "4:2:0" or "4:2:2" sampling method is used this is one of two steps in the compression process were information is lost.

[8] The names of the sampling methods are confusing, "4:2:0" does **not** mean that Y is sampled four times, Cb two times and Cr not at all; it is as described above, for four Y-pixels (a 2x2 block) there is one Cb- and one Cr-pixel sampled. The origin of the name lies in analog television signal transmission.

[9] Analog televisions signals transmitted the image in interlaced mode. First the odd lines of the image, then the even lines. This allowed to double the frame rate of the transmitted video signal but prevent to average over different lines.

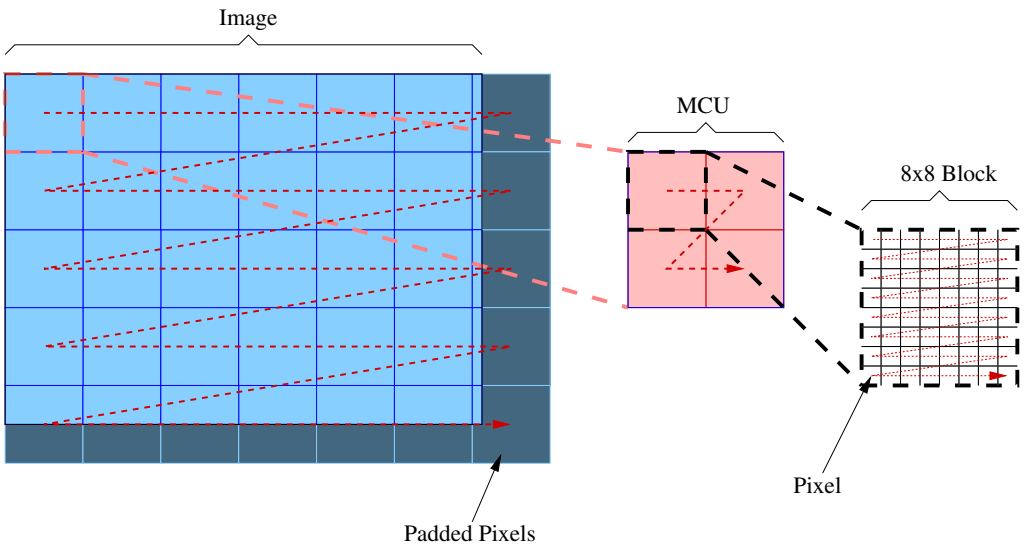


Figure 1.4: A *JPEG* image is composed of smaller units. An image is composed of MCUs which consist of square blocks of 8x8 pixels. It depends on the chosen sampling method how many 8x8 blocks form an MCU (cp. fig 1.3), in this example the 4:2:0 sampling method is used. The order in which the units will be processed is always from left to right and from top to bottom. For the MCUs it is also important to keep the color-decomposition in mind (cp. fig. 1.5).

Since every component is encoded separately in 8x8 blocks and chroma components might be reduced in resolution, for full coverage there might be more than one block in the luma component needed for every block in the chroma components. For "4:2:0" there are four luminance 8x8 blocks needed (arranged in a square), for "4:2:2" only two luminance 8x8 blocks. The required luminance blocks plus one block for each chrominance component is called *Minimum Coded Unit* or *MCU* [10]. The MCUs are illustrated in figure 1.3 as blue squares.

The arrangement of data units will always be from left to right and from top to bottom. This order applies to the pixels inside an 8x8 block as well as for the (luminance) 8x8 blocks in the MCU as well as for the MCUs in the image (illustrated in figure 1.4).

The YCbCr components in an MCU will be in following order: Y-Y-Y-Y-Cb-Cr for "4:2:0", Y-Y-Cb-Cr for "4:2:2", Y-Cb-Cr for "4:4:4" and Y for "gray" (cp. figure 1.5).

Since the image is divided into 8x8 blocks, the encoder pads the image's width and height to be a multiple of 8. Taking sampling into account the image dimensions must even be a multiple of the MCU's dimensions.

[10]Note that in the MPEG compression standard there is a similar, nonetheless different construct of 16x16 blocks called "Makroblock".

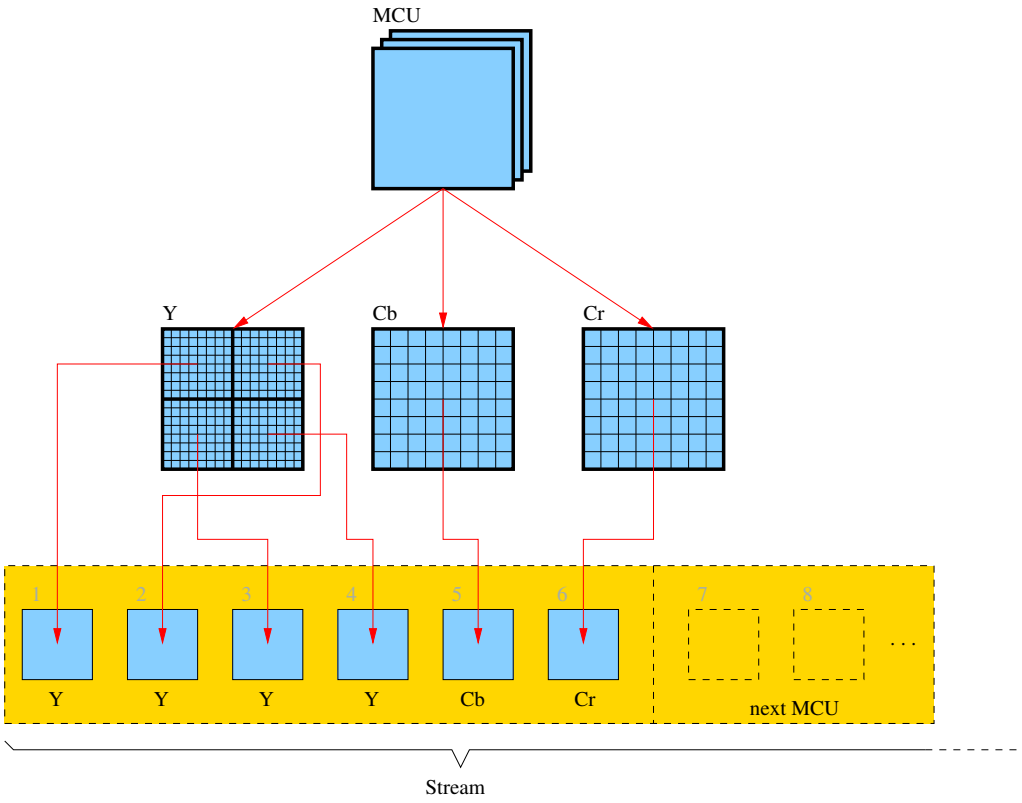


Figure 1.5: A MCU contains three components, from which usually the two color components are reduced to half resolution in both dimensions (4:2:0 sampling). Therefore an MCU consists of 4 luminance and 2 chrominance (one Cb and one Cr) 8x8 blocks. The order in which these blocks are processed is illustrated here.

1.4 Discrete Cosine Transformation (*DCT*)

The *discrete cosine transformation* belongs to a class of linear transforms that are related to *Fourier analysis*. Those transforms map an n-dimensional vector to a set of n coefficients. A linear combination of n known basis vectors weighted with the n coefficients will result in the original vector. So mathematically, it is just a *"change of basis"*. The known basis vectors of transforms from this class are "sinusoidal", which means that they can be represented by sinus shaped waves or, in other words, they are strongly localized in the frequency spectrum. Therefore one speaks about transformation to the frequency domain. The most famous member of this class is the *Discrete Fourier Transformation (DFT)*. The difference between DCT and DFT is that DFT applies on complex numbers while DCT uses just real numbers. For real input data with even symmetry DCT and DFT are equivalent.

The input data to be processed is a two-dimensional 8x8 block, therefore we need a two-dimensional version of the *discrete cosine transformation*. Since each dimension can be handled separately, the two-dimensional DCT follows straightforward form the one-dimensional DCT. A one-dimensional DCT is performed along the rows and then along the columns, or vice versa.

For a more complex mathematical description, the user can refer to [NTR74], [Kha03], [Way99, chapter 9] and [Wat99, chapters 3.5-3.7].

The resulting formula for the two-dimensional *discrete cosine transformation* (DCT) and the corresponding *inverse discrete cosine transformation* (IDCT) is:

$$F(u,v) = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) \cos\left(\frac{2x+1}{16} u\pi\right) \cos\left(\frac{2x+1}{16} v\pi\right) \qquad \text{(DCT)} \qquad (1.3)$$

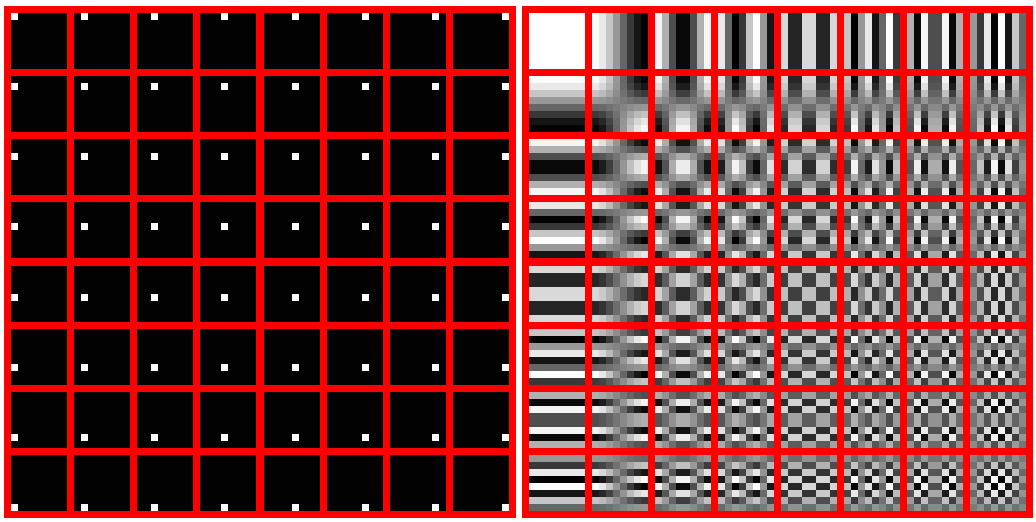
$$f(x,y) = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C_u C_v F(u,v) \cos\left(\frac{2x+1}{16} u\pi\right) \cos\left(\frac{2x+1}{16} v\pi\right) \qquad \text{(IDCT)} \qquad (1.4)$$

$$C_u,\ C_v = \begin{cases} \frac{1}{\sqrt{2}} & for\ u,v = 0 \\ 1 & otherwise \end{cases}$$

where $F(u,v)$ is the DCT coefficient for the 2d-DCT component (u,v) and f(x,y) the pixel value at position (x,v) in the 2d-input-data.

Applying these formulas directly requires much computational resources. Although fast algorithms have been developed [AAN88] it is still computationally expensive. However, the algorithm is suitable for parallelization and therefore an implementation in hardware can be very efficient [Pil07a].

A *JPEG* baseline compression compliant *discrete cosine transformation* takes blocks of 8x8 signed integer values with 8 bit precision as input and produces output blocks of 8x8



(a) Pixel components (b) DCT components

Figure 1.6: Before the *discrete cosine transformation* the 8x8 block is represented by it's 64 discrete pixel values. It can be interpreted as a linear combination of the 64 *basis blocks* on the left, weighted with the corresponding pixel values. After the transformation the block is expressed as a linear combination of the 64 *basis blocks* on the right, weighted with the DCT coefficients. The 8x8 block is still well-defined, but represented in an other basis. Picture 1.6(b) source: [Wik08].

signed integer values with 11 bit precision. [11]

Figure 1.6 illustrates the 64 basis components of the spatial domain (fig. 1.6(a)) and the frequency domain used by the DCT (fig. 1.6(b)) arranged in an two-dimensional 8x8 array.

1.5 Quantization

The *"Quantization"* is a key step in the compression process since less important information is discarded.

The advantage of the representation in the frequency domain is that, unlike in spatial domain before the DCT, not every dimension has the same importance for the visual quality of the image. Removing the higher frequencies components will reduce the level

[11] Note that in theory the transform is a lossless step in the compression chain but information will be lost due to limited machine precision.

of detail but the overall structure remains, since it is dominated by the lower frequency components.

The 64 values of a 8x8 block will be divided according to the 64 values of an 8x8 matrix called the *quantization table*. There is no information lost in the division of the coefficients itself, but the result is then rounded to the next integer afterwards. The higher the divisor, the more information about the coefficient will be positioned after the decimal point hence lost in the rounding operation.

The *JPEG* standard provides an example *quantization table* which has "been used with good results on 8-bit per sample luminance and chrominance images" [Jpe92]. Many encoders simply use this example, but the values are not claimed to be optimal. An encoder may use any other *quantization table*, probably optimized by analyzing the image first. The *quantization table*s are stored in the header of the *JPEG* file in the "DQT" (Define Quantization Table) marker to be available for decoding.

Most *JPEG* encoders allow the user choose a compression level or a quality setting. This parameter specifies the tradeoff between the efficiency of the compression and the associated quality loss. By setting this parameter one actually just specifies what *quantization table* the encoder will use (usually the example *quantization table* will be multiplied by a factor computed from the chosen parameter).

The 64 coefficients of the Discrete Cosine Transformation are, like their spatial equivalents, sorted in two dimensions forming an 8x8 block. Towards the bottom the frequency in vertical direction increases, towards the right the frequency in horizontal direction increases. So the lower frequencies are located in the upper left corner of the block (cp. figure 1.6(b)).

Figure 1.7(a) shows an example 8x8 block of DCT coefficients. Figure 1.7(c) shows the results of this block after quantization and rounding (using the example *chrominance quantization table* (figure 1.7(b)) from [Jpe92]).

1.6 Zigzag-Mapping

The two dimensional order of the DCT coefficients refers to the two dimensions that the 8x8 block had in spatial domain. This order, of course, is not mandatory; one can rearrange the coefficients in any well-defined way. After the quantization step most of the coefficients towards the lower right corner are zero. The *Zigzag-Mapping* - as shown in figure 1.7(d) - rearranges the coefficients in a one dimensional order, so that most of the zeroes will be placed at the end. This array with many consecutive zeroes at the end is now optimized to achieve high compression in entropy encoding.

496	−42	33	17	−20	43	44	−20
160	26	29	7	54	7	12	−18
25	49	−15	32	48	−28	44	20
41	27	−35	20	3	12	−16	9
−39	11	30	−31	26	−9	15	−24
23	−23	23	−9	13	26	12	11
15	17	−13	10	5	−34	5	43
−10	12	8	−22	6	38	−46	8

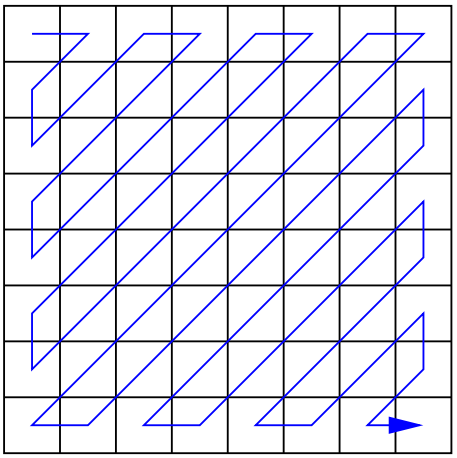
(a) Original DCT Coefficients

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

(b) Quantization Table

29	−2	1	0	0	0	0	0
3	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(c) Quantized DCT Coefficients



(d) Zigzag-Order

29	−2	3	1	1	1	0	0	0	1	0	0	0	0	0	...

(e) After Mapping

Figure 1.7: Quantization will divide the coefficients according to a *quantization table* and round the result to the next integer. The *quantization table* is chosen in a way that most of the lost information belongs to the higher frequencies. This way the block might lose some details but it preserves the global structure. Afterwards most of the remaining coefficients will be located in the upper left corner. Serializing in zigzag-order will result in many consecutive zeroes, thus well suited for entropy encoding.

1.7 Entropy Encoding

The final step is a combination of three techniques: *run length encoding*, *variable length encoding*, and *Huffman encoding*.

The first coefficient is called "DC", all other coefficients are called "AC"[12].

There are several things to keep in mind.

- The first coefficient (DC) is the mean value of the original 8x8 block.
- There is a correlation between the DC coefficients of neighboring blocks.
- It is very likely that the first coefficient has the largest value. This is the most significant coefficient and therefore usually the least reduced one in the quantization step.
- Most zero coefficients appear at the end.
- The chance to find some consecutive zeroes followed by a non-zero component is good as well.
- Most non-zero coefficients have very small values.

The DC coefficient will be decoded slightly different than the AC coefficients. Respecting the correlation to the neighboring blocks, just for the first block the whole DC coefficient is processed. Later blocks will only encode the difference to the preceding block's DC component, this applies for each component separately. AC and DC coefficients have different Huffman tables.

Let's look at an example block of coefficients (the one from figure 1.7(e)):

$$29, -2, 3, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, \ldots$$

Let's assume that the previously decoded block of the same component had the DC coefficient 22, therefore we decode the difference $29 - 22 = 7$.

$$7, -2, 3, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, \ldots$$

[12]This refers to electronics, where "DC" means direct current and "AC" alternating current. The first coefficient is the mean value of the block and the only one contributing equally to all pixels

So now we take care of the zeroes using run length encoding. The tailing zeroes will be combined in one code, called "eob" [13] . To each non-zero code we will stick the information about preceding zeroes, so we can remove the rest of the zeroes. [14] For the DC coefficient there will be no preceding zeroes, however, unlike for the AC coefficients, "zero" is still a valid value that has to be concerned.

$$\begin{array}{cccccccc} 7, & -2, & 3, & 1, & 1, & 1, & \underbrace{0,0,0,1}, & \underbrace{0,0,0,0,\ldots} \\ [\,]\,7, & [0]\,-2, & [0]\,3, & [0]\,1, & [0]\,1, & [0]\,1, & [3]\,1, & [\text{eob}] \end{array}$$

The remaining coefficients will probably be very small so that an variable length approach seams feasible. Therefore we switch to binary representation and add the minimum number of bits needed to represent the coefficients value to the information part. Negative values will be represented by negating every bit (one's complement). This can be done because we have the information about the length, so that every positive value starts with an 1.[15]

$$\begin{array}{cccccccc} 7, & -2, & 3, & 1, & 1, & 1, & \underbrace{0,0,0,1}, & \underbrace{0,0,0,0,\ldots} \\ [\,]\,7, & [0]\,-2, & [0]\,3, & [0]\,1, & [0]\,1, & [0]\,1, & [3]\,1, & [\text{eob}] \\ [\,3]\,111, & [0\;2]\,01, & [0\;2]\,11, & [0\;1]\,1, & [0\;1]\,1, & [0\;1]\,1, & [3\;1]\,1, & [\text{eob}] \end{array}$$

[eob] is coded as [0 0], [zrl] as [15 0]; there is no other code with the structure [X 0].

Since the coefficients are usually very small there is not much gain in compressing them further. But we haven't thought about the information we attached to the coefficients yet. We use 4 bits for the preceding zeroes (see footnote 14 on page 19) and 4 bits for the number of bits used to store the value[16] . These 8 bits are compressed using a Huffman table which maps the frequently occurring values to shorter bit strings and the rarely occurring values to longer bit strings. How to choose the table is left to the encoder[17] .

Let's assume we have build the Huffman table and find the following tables:

[13] For the rare case that the last coefficient in the block is not zero, there is no "eob" appended.

[14] Later we will use just 4 bits for the information about preceding zeroes, so if there are (unlike in this example) more than 15 preceding zeroes we will exchange 16 zeroes with another special code, "zrl" (zero run length). We do this until there are less than 16 zeroes left. Example: 7,4,3, <44 zeroes>, 1, <rest is zero> will be coded as: [0]7,[0]4,[0]3,[zrl],[zrl],[12]1,[eob] .

[15] $[0\;3]\;100 \neq [0\;4]\;0100$ the left value is 4, the right value -11

[16] The Huffman algorithm used is designed for up to 15 bit precision but the baseline process only uses 11 bit precision. More advanced *JPEG* processes may use the full 15 bits.

[17] The official standard comes with example tables for luminance (DC and AC) and chrominance (DC and AC) based on statistics of a large set of images.

DC Table	
Huffman code	Original Value
⋮	⋮
110	0x3 [3]
⋮	⋮

AC Table	
Huffman code	Original Value
00	0x00 [eob]
01	0x01 [0 1]
100	0x02 [0 2]
101	0x11 [1 1]
1100	0x03 [0 3]
⋮	⋮
1111110	0x31 [3 1]
⋮	⋮

Now we can construct the final bit stream:

	$7,$		$-2,$		$3,$		$1,$		$1,$		$1,$	$\underbrace{0,0,0}$	$1,$	$\underbrace{0,0,0,0,\ldots}$
[]	7,	[0]	−2,	[0]	3,	[0]	1,	[0]	1,	[0]	1,	[3]	1,	[eob]
[3]	111,	[0 2]	01,	[0 2]	11,	[0 1]	1,	[0 1]	1,	[0 1]	1,	[3 1]	1,	[eob]
110	111	100	01	100	11	01	1	01	1	01	1	1111110	1	00

The final bit stream:

```
11011110 00110011 01101101 11111110 100
```

So we compressed the 64 bytes of input data down to less than five bytes.

Storing of the Huffman table The Huffman table is stored in the header (DHT marker segment) in a sophisticated way. It will be easier to understand if explained along an example, so let's take a look at the following example DHT table marker:

```
FF C4 00 39 11 00 02 02 02 01 04 02 02 01 04 01
04 01 03 02 07 01 02 03 11 04 12 21 05 13 22 31
00 06 32 41 14 07 23 42 51 15 33 52 61 71 24 16
43 62 08 25 81 91 34 72 63 A1 C3
```

The first two bytes are the DHT[18] marker (`FF C4`), they are followed by two bytes coding the length of the DHT marker segment (`00 39`). Then comes one byte table context (`11`); in this case it's the AC table number one. In the next 16 bytes are the Huffman codes but given in a special way. The first of these 16 bytes stands for the number of Huffman codes with length one, the second byte gives the number of codes with length two and so on. The actual codes have to be rebuild the following way.

[18]For details see appendix A

So in our case we have no code with length one, but two with length two. Taking into account that it will never be prefix of an existing code, a new code will always be the lowest possible number. So we start with the code 00 and then 01.

Length	1	2	
Code	-	00	01
Value	-		

Next we have two codes with the length of three bits, giving:

Length	1	2		3	
Code	-	00	01	100	101
Value	-				

And so on ...

Length	1	2		3		4		5	6
Code	-	00	01	100	101	1100	1101	11100	...
Value	-								

Right after the 16 bytes for the Huffman codes follow the values to be associated with the codes. The number of values depends on the number of codes which have been rebuilt (= sum of the 16 code-bytes).

Length	1	2		3		4		5	6
Code	-	00	01	100	101	1100	1101	11100	...
Value	-	0x01	0x02	0x03	0x11	0x04	0x12	0x21	...

The Huffman tree is not completely utilized, there are no codes that are all 1. Another important thing is that any occurrence of the byte aligned value 0xFF inside the compressed data will be masked as 0xFF00.

2 Basic Approach

This chapter formulates the different considerations for the implementation of the *MotionJPEG* decoder. The layout of the infrastructure around the *JPEG/MotionJPEG* decoder is described and some in-depth thoughts concerning the design of the decoder pipeline are formulated.

2.1 Objectives

The *JPEG* compression algorithm with its sequential decoding steps implies a pipelined design. The pipeline is designed with strictly encapsulated components, interconnected by a two-signal flow control. This way the components may also be used in other projects with minimal effort.

The encapsulated components are also very well suited to be used in a demonstration application in the field of *dynamic partial reconfiguration*. A *hardware scheduler*, as developed by Norbert Abel for his diploma thesis [Abe05], can be used to swap single components in and out, reducing the required chip-resources.

The first step to a video decoder is the implementation of a still image decoder. In doing so the long-term objective, to later do video decoding, has always been kept in mind. The *JPEG* decoder has been developed with regard to be easily upgradeable without performance penalty.

2.2 Basic System Layout

Figure 2.1 outlines the decoding system whose major part is the developed *JPEG* decoder. Besides *JPEG* decoding, the system fetches the image data from the SDRAM, takes care of displaying the decoded data on a VGA monitor, and provides controlling functionality via the PowerPC.

The decoding system is implemented as OPB component ("MyIPIF"[1]) being part of an EDK design. The image data is read from SDRAM and served via the OPB bus to the de-

[1] OPB IPIF (On-Chip Peripheral Bus Intellectual Property Interface) is a standardized interface between the OPB and a user IP core.

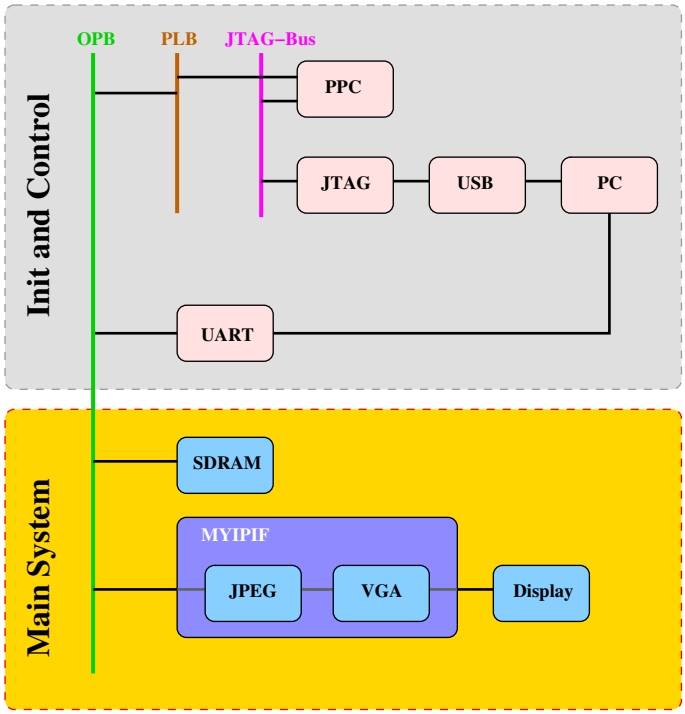


Figure 2.1: The image data is held in the SDRAM. It is presented to the *JPEG* decoder via OPB. The decoded data is then displayed directly on a VGA monitor. Initializing and controlling the entity is done via the PowerPC.

coder entity in 32-bit words. The OPB is configured to operate in burst mode, presenting a new word every clock cycle until the input buffer is completely filled. New data is not requested until the logic of the input buffer requests new data. This way most of the time the OPB will be free to be used by other components which eases the requirements for possible future extensions (like upgrading to *MPEG*, cp. chapter 7.3).

Sending the decoded data over the OPB would as well result in a unnecessary load on the bus. Therefore a basic VGA core has been implemented (cp chapter 5) which is directly instantiated in the "MyIPIF" component and interfaces the *JPEG* decoder to avoid bus traffic.[2]

Since there are insufficient on-chip memory resources available to store a complete decoded image, for a frame buffer the SDRAM needs to be utilized. This would require complex logic for memory management and is not implemented. The decoder has to provide the data synchronous to the to the VGA component displaying it. So the decoder has to (a) be be fast enough to always provide valid data and (b) come with reverse flow control support to not outrun the VGA component. The VGA core operates at 60 Hz. Therefore one image needs to be decoded in less than $\frac{1}{60}s$.

A detailed discussion of the *JPEG* decoder and the VGA entity can be found in chapters 3 and 5 respectively.

[2]Note that the OPB bandwidth is sufficient to handle a stream of decoded data. This is necessary for an *MPEG* decoder design (cp. chapter 7.3).

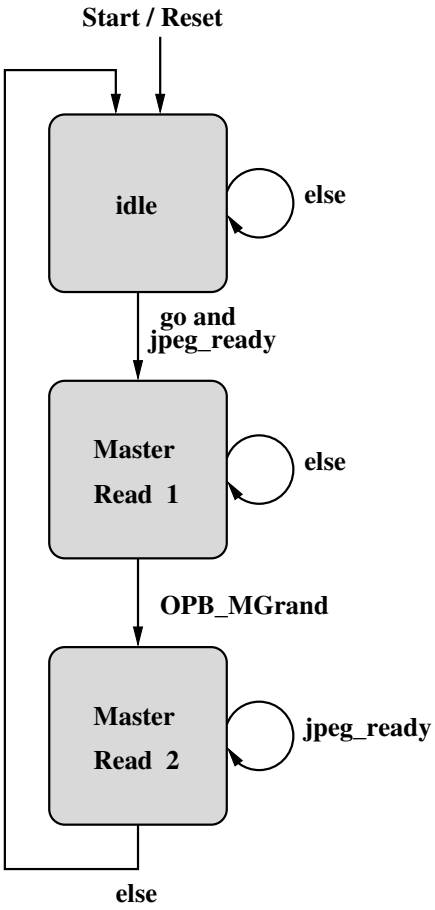


Figure 2.2: The finite state machine to initiate an OPB-master-read cycle. In the `idle` state the entity is listening on the OPB for configuration commands, but as soon as the jpeg decoder requests new data by asserting `jpeg_ready` it will change to the next state. However with the configuration register `go` the user may force the state machine to stay in `idle`. In the `Master Read 1` state the entity is requesting the bus from the OPB arbiter. If the bus has been granted (`OPB_MGrand`) the state machine switches to the state `Master Read 2` where it requests data from the SDRAM until the input buffer is filled and `jpeg_ready` is set to low by the *JPEG* decoder causing the return to the `idle` state. While in state `Master Read 1` the OPB is locked for other components.

2.3 MyIPIF

The *MyIPIF* component is connected to the EDK system as a OPB-master component. It fetches the data from the SDRAM in burst mode (this mode is also known as "*OPB sequential address*"). The finite state machine for this OPB-master-read cycle is outlined in figure 2.2. The OPB provides the data in 32-bit words starting with the first word from the given address, then every clock cycle the next 32-bit word in sequence is provided. When the input buffer of the *JPEG* decoder is filled the state machine memorizes the last read address. The next time data is requested by the *JPEG* decoder the state machine will resume reading from this position. However, if the decoder signals that it received the *end of image* (EOI) marker, the next read cycle starts from the base address again.

When not in the idle state, *MyIPIF* is locking the bus and no other than the addressed OPB component (in this case the SDRAM controller) is allowed to access it. When in the idle state, the component listens on the OPB for configuration commands that then can be issued by the PowerPC (cp. chapter 3.9).

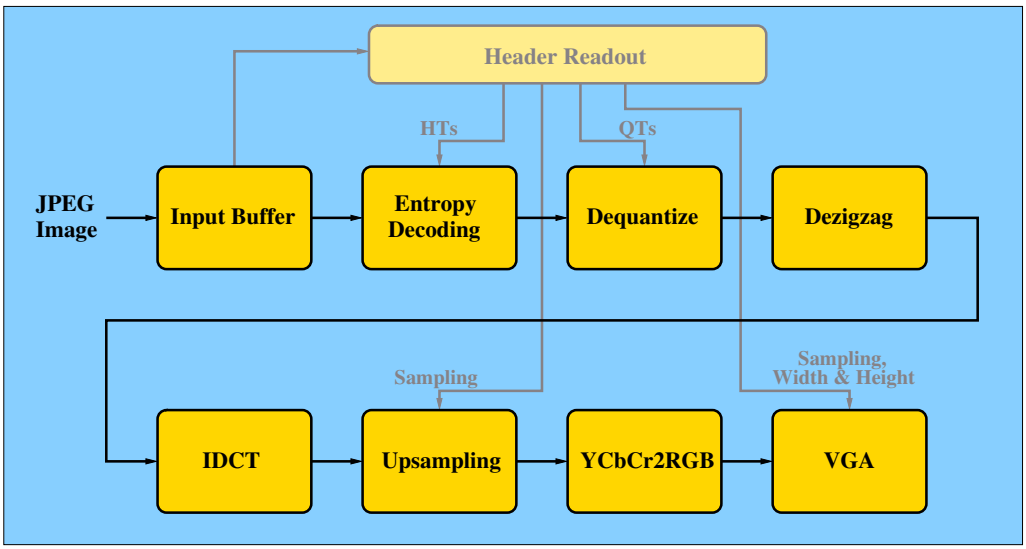


Figure 2.3: The essential steps of the *JPEG* decoder. Except the header readout all components are arranged in a pipeline chain. The header readout is closely connected to the input buffer and presents the information from the header to the specified components.

2.4 Pipelining

Since there are several independent steps in series (cp. figure 2.3), the *JPEG* decoding algorithm is very well suited to be implemented in a pipelined manner. For the pipelining some issues need to be addressed. Insufficient on-chip video memory implies the need to stop and resume decoding on request, so the pipeline has to have a proper backpressure support. Furthermore the context of the data is important. Along with the data, every component in the pipeline needs to know the context information as well. The efficiency of a pipeline is best if it is "balanced", i.e. all components need about the same processing time for their step.

Context The context can be information from the header (e.g. the sampling method used) or something else (e.g. that an EOI marker has been detected). Keeping in mind that the core should be upgradable to *MotionJPEG* this leads to a problem when a new image enters the pipeline and the header related context changes. At that point the pipeline may hold two different images with different header information. The size of the header information (including Huffman- and quantization-tables) is too large to stick it to the data and pass it through the pipeline. Therefore the header readout provides two sets of header information, updated alternately. The (header related) context provided along with the data is just a flag indicating which set to use. This way draining the pipeline on image change can be avoided.

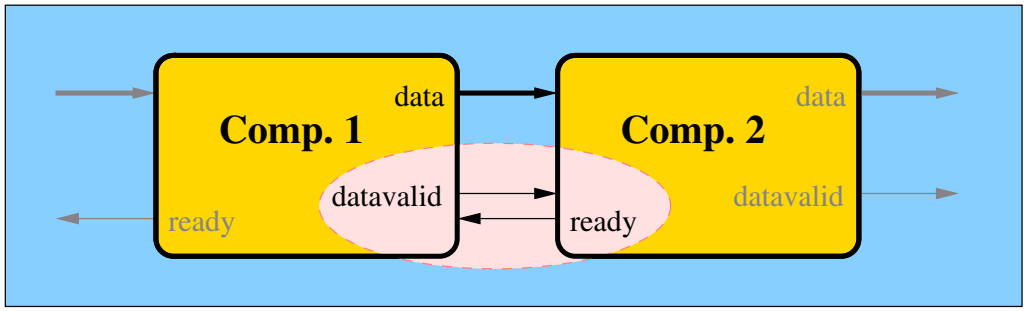


Figure 2.4: The connection of two components in the pipeline is controlled by two signals, `datavalid` and `ready`. When both flow control signals of the connection are asserted, the data is sampled. Both components have to check both signals.

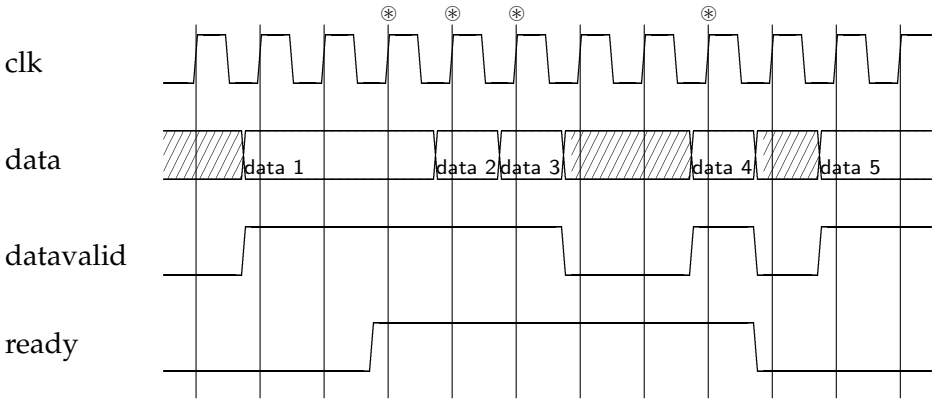


Figure 2.5: Timing diagram for the flow control signals. Data will only be sampled when both signals, `datavalid` and `ready`, are sampled high on the rising edge of the clock. Cycles with valid data handover are marked with ⊛.

Flow Control The flow control ensures that only valid data is processed, no data is lost, and no data is processed multiple times. Each component in the pipeline provides two flow control signals, `datavalid` and `ready`. `datavalid` implies that the component provides valid data and `ready` indicates that the component is ready to take data. The `datavalid` signal is connected forward, the `ready` signal backwards in the chain (see figure 2.4). When the `datavalid` signal of the component providing the data is asserted along with the `ready` signal of the component taking the data, the data is sampled. Both components analyze the flow control signals separately. Figure 2.5 shows a timing diagram illustrating the flow control.

The design of the flow control signals allows to easily separate the pipeline components by connecting fifos in between. This is important for the *dynamic partial reconfiguration* environment, if single components shall be scheduled.

Pipeline Pipelining works best if the pipeline stages are balanced, i.e. the time required for processing is the same for each stage. The decomposition of the image in smaller units may raise the idea of accelerating critical stages by demultiplexing the data stream into more than one path, each path decoding its own data unit. In this case however this would result in no speed gain because the units cannot be identified until after the entropy decoding which is the bottleneck of the decoding process as well.

3 Implementation of the *JPEG* Decoder

This chapter gives in detail how the decoder has been implemented, each of the *JPEG* components are discussed.

For some components it is important to have the theoretical background of the *JPEG* compression, as discussed in chapter 1, in mind. To ease reading, a short description of the theory behind the component will then be given in a blue box at the beginning of the section.

3.1 *JPEG* Top Entity

The JPEG *top entity* encapsulates the connections of the pipeline components. The backward and forward handshaking signals to the outside are those from the first and last pipeline component respectively. The input data is 32-bit wide and the component has three 8-bit wide output data channels (one for each RGB component). Additionally some context information is provided along with the data to the outside, namely the sampling method, the width, the height and an "end of image" flag for synchronization with the VGA entity. A second "end of image" flag is provided backwards to the OPB logic, indicating that all image data has been read. The OPB logic will then start to re-transmit the image data.

3.2 Input Buffer and Header Readout

The input buffer has been designed as a two-stage fifo with logic to pre-analyze the incoming data (cp. figure 3.1). The data coming from the OPB is collected in a first fifo ("Fifo 1") where the 32-bit input data is presented in chunks of 8 bits to the preanalyzer ("check_FF"). Since the data is stored in sequential addresses, it can be fetched in bursts. However, data after the end of the image is not valid and has to be flushed. "check_FF" checks for the EOI marker. If EOI is detected the invalid data will be flushed from "Fifo 1". Furthermore a new decoding cycle is then initiated by signaling the "MyIPIF" component to re-transmit the whole image again and reactivate the header readout.[1].

[1]The header readout state machine is outlined in Appendix B.

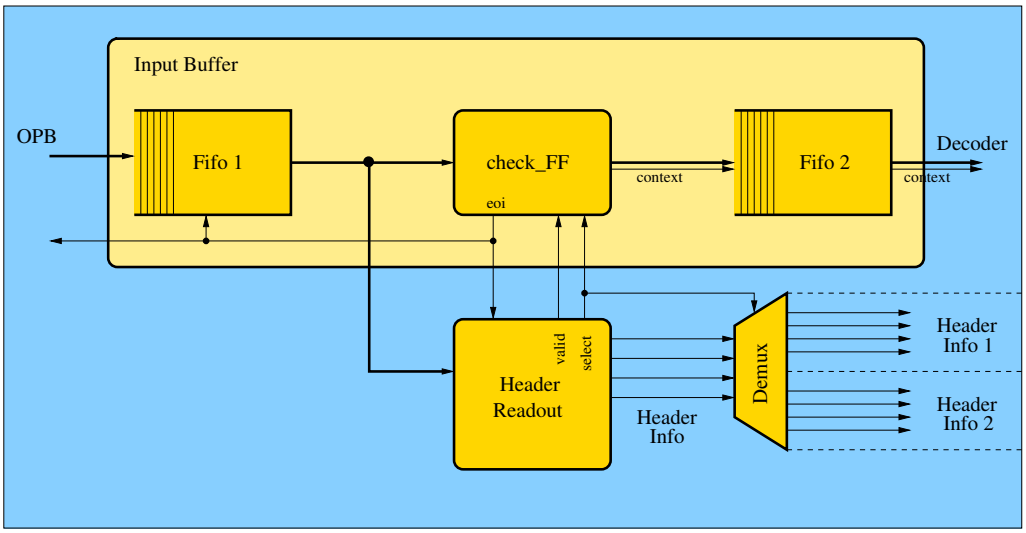


Figure 3.1: The design of the two-staged input buffer is illustrated here. The OPB - operating in burst mode - continues reading even after the end of the image. `check_FF` detects the EOI marker and flushes the invalid data in `Fifo 1`. With regard to *MotionJPEG*, the header readout reads the header information each time the image is decoded, presenting it by alternately updating two different information sets. The presence of `Fifo 2` allows the header readout being done in parallel to the rest of the decoding process.

For decoding the same *JPEG* image over and over it would be sufficient to read out the header just once at the beginning and store the information for the rest of the decoding process. While decoding *MotionJPEG* there are several different images to be sequentially processed and the header of each has to be read. So the header information is not just read out once at the first decoding cycle but for every cycle again. This way the decoder is already prepared for *MotionJPEG*. On the other hand the header information is required by many pipeline stages and cannot be updated while these components are busy. Draining the pipeline before decoding the next image would be very inefficient, so the decoder provides space for two sets of header data used alternately. This way there is no need to drain the pipeline before decoding the next image[2].

The two-staged input buffer combined with the two header-information sets allows to process the header of image `n+1` while the rest of the decoder is still working on image `n`. The multiple header readout itself does not add any delay since it is done in parallel to the rest of the decoding process.

There is another important task that the preanalyzer takes care of. Any occurrence of 0xFF in the compressed bitstream has been escaped to 0xFF00 by the encoder, the preanalyzer replaces the byte sequence 0xFF00 in the compressed data with just 0xFF.

3.3 Entropy Decoding

Theoretical Background:

The last step in the encoding process is entropy encoding, therefore it is the first step in the decoding process. The entropy encoding is a combination of *run length encoding*, *variable length encoding* and a special form of *Huffman encoding*. Entropy decoding is divided in two steps. First, the information needed for the *run length decoding* and the *variable length decoding* needs to be decoded by applying a Huffman algorithm. Then *run length decoding* (used for the zeroes) and *variable length decoding* (for the non-zero values) can be applied (cp. figure 3.2).

Figure 3.3 shows a simplified model of the finite state machine used for the entropy decoding. To keep a clear picture, several status registers and buffers required to perform entropy decoding are not shown. Because this is a very complex pipeline component, this section is divided in several subsections.

[2] A third image has to be stalled, though.

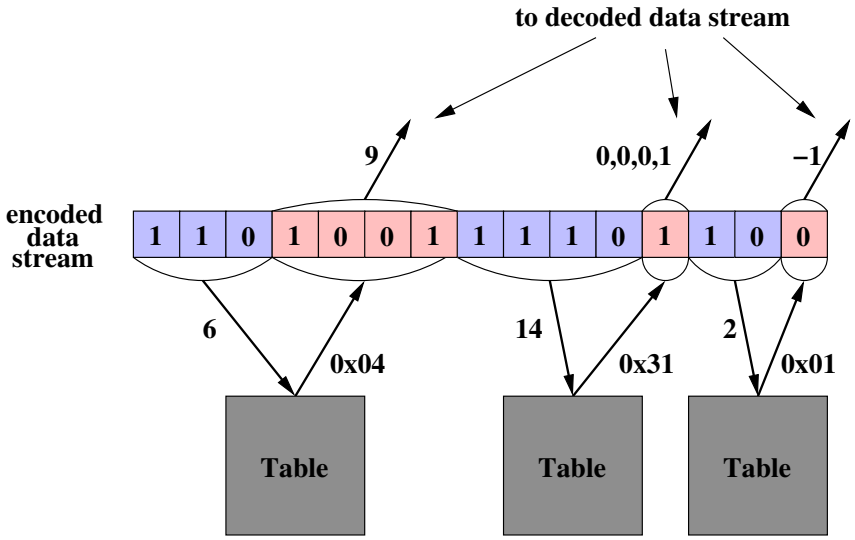


Figure 3.2: The data is read bit by bit until a valid Huffman code (blue) is found. The Huffman table is consulted for the information how many preceding zeroes there are and how many bits the encoded value (red) requires. First the preceding zeroes are written, then the required number of bits is read and the resulting value is written.

3.3.1 Data Input

The decoder needs the data bit by bit, but the input buffer presents the data in 8 bit junks. Changing "Fifo 2" to a 8-to-1 asymmetric fifo would implicate problems with the context that already needs to be presented along with the data. So a 12-bit wide "Fifo 2" is used and the 4-bit context together with the 8-bit data are written as one 12-bit word into this fifo. On the read side of the fifo the data is buffered in an 8-bit shift register and shifted out bit by bit.

A special case may occur if a *JPEG* marker (RST[3], DNL[3] or EOI) has been inserted into the compressed stream. The markers are byte-aligned and therefore it may have been necessary to pad (invalid) bits before the marker. These padded bits are all '1' and therefore are no valid Huffman code according to the specifications. If encountered, the finite state machine continues to decode until the last stuff bit is processed is then reset as soon as the marker is read (the marker is detected by the input buffer, which then sets a flag in the context). This way no valid code is missed.

[3]Not yet supported. The RST (Restart) and DNL (Define Number of Lines) markers are usually not used.

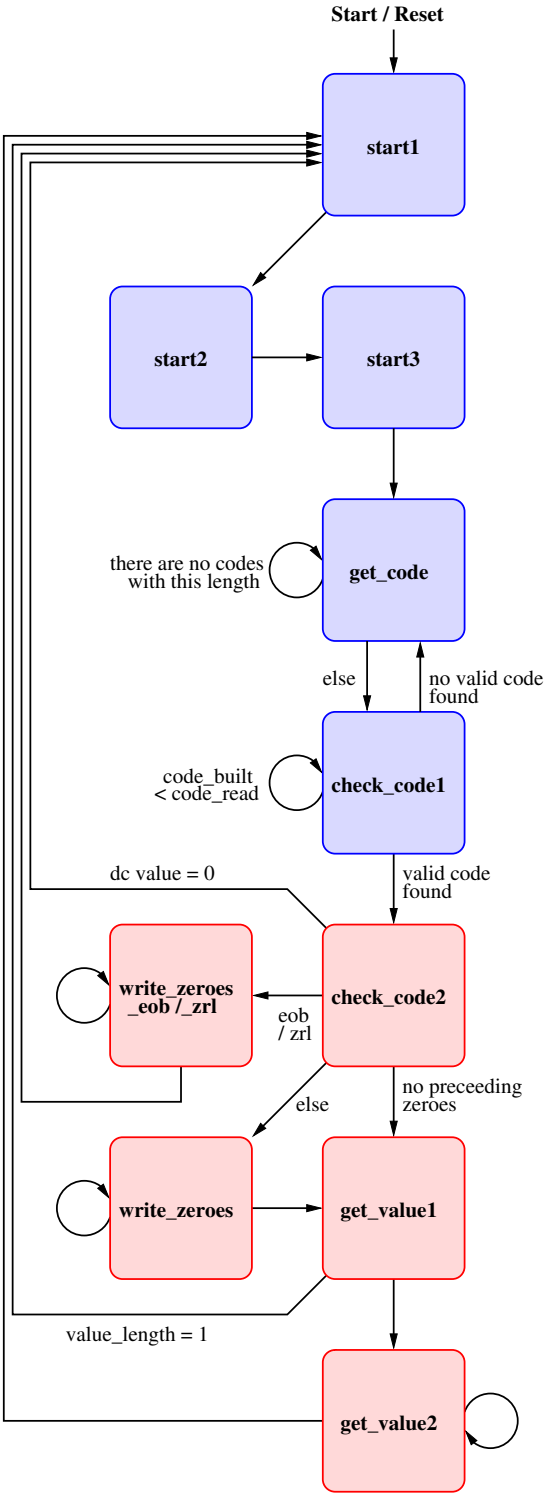


Figure 3.3: The finite state machine for entropy decoding. States for the Huffman decoding are colored blue, states for variable length decoding and run length decoding are colored red. For a detailed description consult the chapters 3.3.2 and 3.3.3.

3.3.2 Huffman Decoding

Theoretical Background:

There are four different kinds of Huffman tables, a different one for DC and AC coefficient and for luminance (Y) and chrominance (Cb or Cr) component. The tables are stored in the header in a sophisticated way. 16 specific bytes hold the information about the codes. The first byte is the number of codes with length 1, the second byte the number of codes with length 2 and so on. They are then followed by the values that are represented by the codes. The codes are not stored in cleartext but have to be build by the decoder. Starting with the lowest length the next valid code is always the lowest possible value that has not a previous code as prefix.

The following is a description of the (blue) states in state machine shown in figure 3.3. Two 16-bit signals are used in this part of the finite state machine, `code_read` and `code_build`. They are compared and if they match a valid code has been detected. `Code_read` is appended by one bit of input data, if necessary, and `code_build` is successively build according to the rules. To distinguish between DC tables and AC tables a counter called `ac_dc_counter` is used, to distinguish luminance from chrominance components a separate, sampling dependent, finite state machine was implemented.

`start1-3:` In the first state the registers storing the table addresses and the codes are reset to zero. The next two states are needed to wait for the BRAM storing the tables to hold valid data.

`get_code:` The length of the potential Huffman code is increased by one bit. A bit is read from the input shift register and appended to the register `code_read`. A zero is appended to the register `code_build`. The number of Huffman codes with the current length is read from the relevant Huffman table (AC or DC, luminance or chrominance) and memorized in `symbols`. If this value is zero, the state machine stays in this state for an additional cycle to append another bit.

`check_code1:` In this state the previously read code is checked for validity by comparing `code_ read` with `code_build`. `code_build` is incremented until either a valid code is found (→ `check_code2`) or the number of tested codes equals the previously memorized value `symbols` (→ `get_code`).

3.3.3 Variable Length Decoding and Run Length Decoding

Theoretical Background:

The byte decoded by the Huffman part of the state machine consists of two nibbles with different meaning. The first nibble represents the information of how many zeroes there are prior to the value to decode, the second nibble is the number of bits required to represent the value to decode.
The Huffman table holds two special bytes (`0x00`: `eob` and `0xF0`: `zrl`) that are treated differently. `zrl` represents 16 zeroes, `eob` indicates that the rest of the block are zeroes.
It must be noted that for the DC coefficients only the difference to the DC coefficient of the previous block has been encoded. This applies for each color component separately.

The following is a description of the (red) states shown in figure 3.3. Two counters are used in this part of the state machine, one for the zeroes to be written and one for the bits to be read for the non-zero value.

Since for the decoding of the DC coefficient, the DC coefficient of the previous block is needed, three register arrays are used to store the DC coefficients, one for each color component. The selection of the correct component register is done by a separate finite state machine.

`check_code2`: The valid code is interpreted by consulting the appropriate Huffman table (AC or DC, luminance or chrominance). It may be one of the special codes `eob` or `zrl` (→ `write_zeroes_eob/_zrl`) or a combination of (a) the number of preceding zeroes (if zero: → `get_value`, else: → `write_zeroes`) and (b) the number of bits required for the value to decode.

`write_zeroes_eob / write_zeroes_zrl`: For `zrl` 16 zeroes are written, for `eob` the remaining bytes of the current 8x8 block are written as zero. Then the state machine will start from the beginning.[4]

`write_zeroes`: The number of preceding zeroes is written.[4]

[4] These states can be separated from the state machine, continuing decoding while the zeroes are written in parallel. Significant performance-enhancing can be achieved (cp. chapter 7.1).

`get_value1:` One bit is read. If it is zero then the value to decode is negative and this bit as well as all following bits need to be inverted.

`get_value_2:` The remaining number of required bits of the value to decode are read and the value is written out.

3.4 Dequantize

Theoretical Background:

In the encoding process the coefficients representing the image in the frequency domain are divided according to the corresponding entry of a quantization table. Two tables are used, one for the luminance and one for the two chrominance components. The result is then rounded to the next integer value, information is lost. The higher frequencies are divided by higher values and therefore loose more information. The division needs to be reversed, this is done in the *dequantize* step.

Since the quantization tables are stored in the header in zig-zag order, dequantize is applied before the *dezigzag* step. This way the tables can be read and used without reordering. They are stored in circular shift registers where the output register is fed back to the input register. The shift registers have to be 8 bit wide and 64 bit deep to hold one table. For the implementation discussed here, they are instantiated using a *coregen* generated component to take advantage of the `SRL16` primitives.

For multiplication a (*coregen*) generated multiplier component is used. On the *Virtex-II Pro* FPGA a `MULT18x18` primitive is used, on other Xilinx FPGAs corresponding primitives would be instantiated (e.g. `DSP48` on a *Virtex 4*).

Information about the used sampling method is needed to choose the correct table. The multiplication of one byte is done in one cycle.

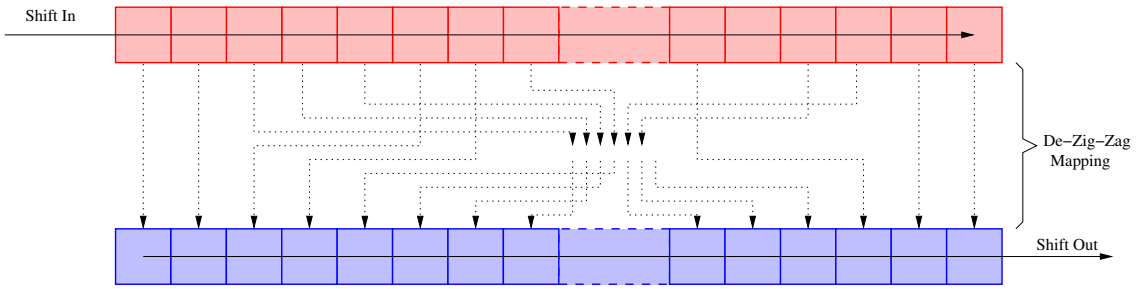


Figure 3.4: If the input shift register (red) is filled and the output shift register (blue) is empty, the data is mapped in the inverse zigzag order. Then new data is accepted again while simultaneously the previously mapped data is shifted out.

3.5 Dezigzag

Theoretical Background:

To optimize the data for entropy encoding the 8x8 blocks are reordered into an array in a zig-zag order so the lower frequencies appear at the beginning and the higher frequencies (that are usually zero after quantization) at the end. The *IDCT* step needs the coefficients in the original order. The reverse mapping is done in the *dezigzag* step.

Two shift registers are used for this step. The first shift register buffers the incoming data. As soon as one complete block is read, it is mapped to the other shift register in the inverse zig-zag order. Then the second shift register shifts out the data while the first begins to process the next data block (cp. figure 3.4).

Shift registers are preferred over *distributed ram*, because *distributed ram* wastes too much fabric resources for the - unneeded - random access logic. Block Memory cores use random access logic as well, but can compensate this by storing the data in dedicated integrated memory blocks instead of logic cells. However those memory blocks are only available in large units[5], a too valuable resource to waste here for just 128 byte data.

The context needs special consideration since the component is internally designed as a two-stage pipeline. The input shift register is filled while simultaneously the output shift register shifts out the previously read data. So the component holds two sets of data with eventually different context information. Therefore two sets of context are needed as well. Context set one is copied to set two at the same time as the data is mapped.

[5] 18 Kbit on the *Virtex-II Pro*

3.6 IDCT

Theoretical Background:

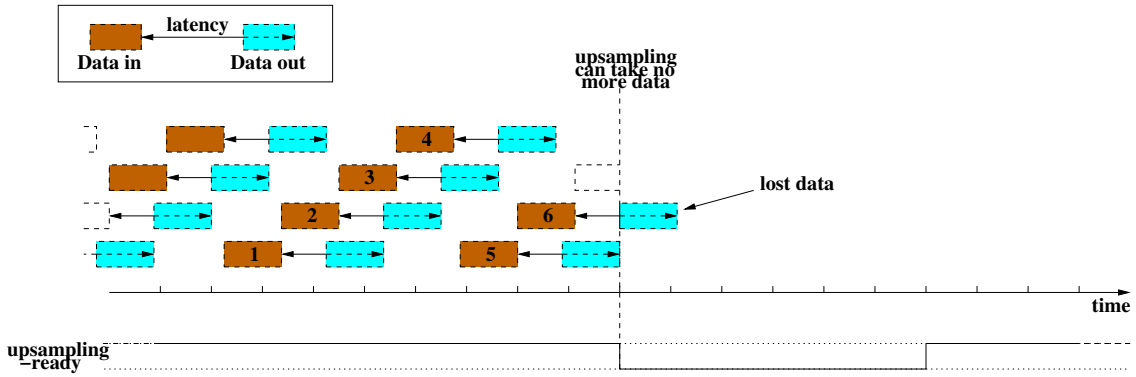
The image has been transformed to frequency space by applying the *discrete cosine transformation*. There the less important[6] , higher frequencies could be compressed in a stronger way than the more important, lower frequencies. In this step the ***inverse*** *discrete cosine transformation* (IDCT) transforms the image back to the spatial domain.

Xilinx provides an *IDCT* core via coregen [Pil07b] which is used because it takes advantage of the FPGAs primitives[7] for the multiplications required. This core is implemented as a pipelined design, but it does not provide reverse flow control.Once one data block is read, data will be shifted out after the calculation delay. If the next entity is then not ready to take the data, it is lost. To ensure proper backpressure, a wrapper has been implemented for the *IDCT* core and the flow control signal "`ready`" of the next component in the chain (*upsampling*) has been adapted. The i"*upsampling*-`ready`" signal is de-asserted when less than one block can be buffered. The "*upsampling*-`ready`" signal is used directly as *IDCT*-`ready` signal, so the second block is not fully read thus indirectly stalling the IDCT core. This can only be done because the IDCT core starts to shift out the first block before the second one is completely read, even if there is always valid input data available. The Xilinx IDCT core for a *Virtex-II Pro FPGA* reads one complete 8x8 block in 92 cycles[8] and needs the same time for writing one 8x8 block. The latency of the core[9] is always 144 cycles (>92 cycles). The time between the last input and the first output data is 144-92=52 cycles (<92 cycles). Setting the *upsampling*-`ready` signal low 64 bytes too early results in a functional reverse flow control while still taking some advantage of the internal pipelined design of the Xilinx-IDCT core (cp. figure 3.5).

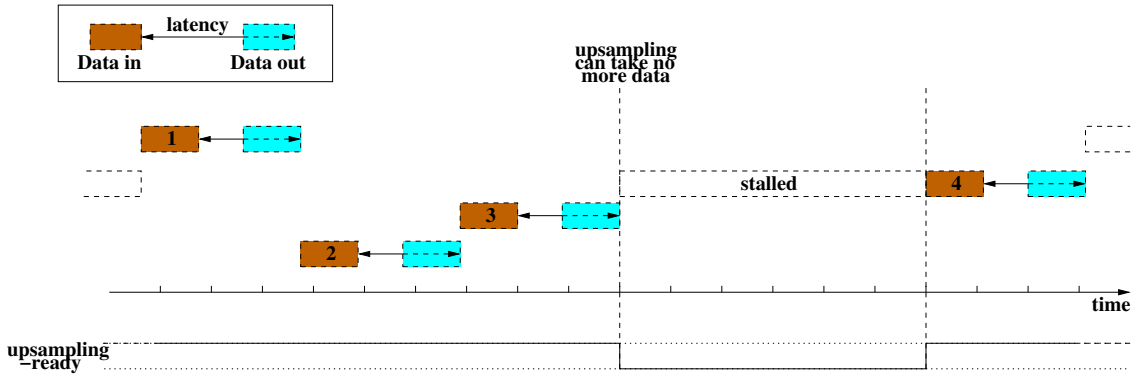
The input coefficients are 12-bit signals, as specified by the *JPEG* standard. Internally, the data may be represented by more bits to reduce rounding errors but requiring more fabric resources. In the case of this decoder 15-bit accuracy is used internally, thus being compliant with the `IEEE 1180-1990` specifications[10][IEE91].

Since the IDCT core may hold more than one (a maximum of three) different sets of data, the correct context must be selected. There are two 8 bit wide counters (`in_counter` and `out_counter`) and four context sets. The `in_counter` increments if data is read, the `out_counter` if data is shifted out. Both counters are two bits wider than needed for one

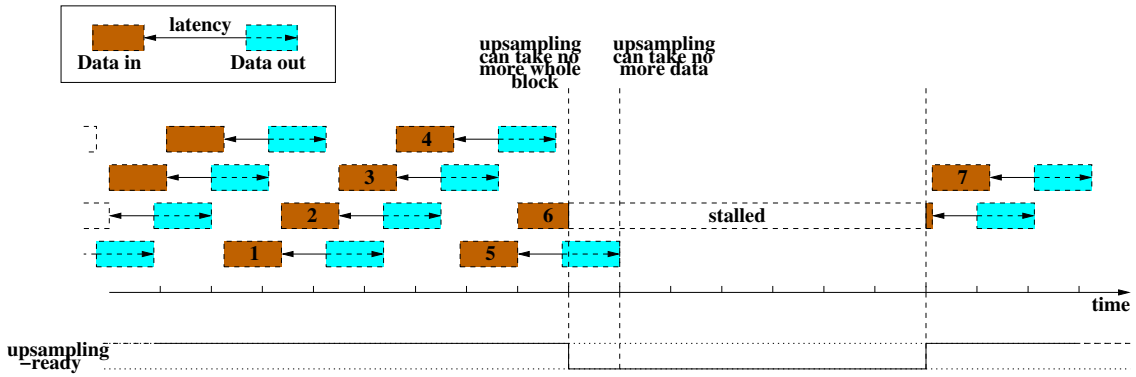
[6] for the viewed optical impression
[7] `MULT18x18` for the *Virtex-II Pro*, `DSP48` for *Virtex-4*
[8] if valid data is always available
[9] last input data to last output data
[10] The IEEE-1180-1990 standard specifies an accuracy test for IDCT transformations.



(a) Flow control as described in section 2.4 results in loss of data because the Xilinx IDCT core cannot be stalled. Once the whole 8x8 block is read, the core will shift out the data after the calculation latency. In this case block number 5 is the last one that can be buffered, but block number 6 is already fully read when the last data of block number 5 is shifted out. Block number 6 is lost.



(b) Manually preventing pipelining in the Xilinx IDCT core results in seriously reduced performance.



(c) A whole block is never fully read before the Xilinx IDCT core **starts** to shift out the previous block. The trick is to set the *upsampling*-`ready` signal to low when the buffer space becomes less than needed for one complete 8x8 block. The *upsampling*-`ready` signal is directly used as *IDCT*-`ready` signal, therefore no new data will be sampled by IDCT. The component is stalled and data loss is prevented.

Figure 3.5: Flowcontrol and pipelining in the *IDCT* component.

block, so the two most significant bits can be interpreted as a two bit block counter. The incoming data is stored in the context set indicated by the two most significant bits of the `in_counter`, the context provided along the output data is taken according to the two most significant bits of the `out_counter`.

Due to the algorithm used the decoded data is shifted out column-wise instead of the required row-wise order. Since in the next entity (*upsampling*) the data is reordered anyways, this does not present a problem. The 8x8 matrix is transposed there in one step with the native reordering.

3.7 Upsampling

(The gray text refers to a previous version.)

Theoretical Background:

The human eye is more sensitive in perceiving differences in brightness than in color. Therefore the color components (in the YCbCr color model there are two color components and one brightness component) are stored in reduced resolution. The upsampling component interpolates the color components to its original resolutions.

Similar to the *dezigzag* component, *the upsampling* component rearranges the data in a different order and therefore has to buffer a whole block of data. It is implemented the same way as the *dezigzag* component by using two shift registers. The internal two-step pipelined design is implicating the above mentioned problem with the context. Two sets of context are needed. Set one copied to set two at the same time as the data is copied. Because the context information (the sampling method) influences size of the buffers and the order in which they are mapped, special care must be taken in the copying process.

The input order is one 8x8 block after another until one complete MCU is received, the output of the component will be a whole MCU in the usual "from left to right and top to bottom"-order. Additionally to the mapping from read-in to read-out order, the data is rearranged to compensate the column wise output order of the *IDCT* component.

Contrary to *dezigzag* which has to buffer only one 8x8 block, this component has to buffer a whole MCU. Depending on the sampling method used, a MCU can be up to six 8x8 blocks for the input buffer and twelve 8x8 blocks on the output buffer. The initial implementation with shift registers is inefficient in terms of device usage. The buffer needs to hold $(6 + 12) \cdot 64$ bytes $= 1152$ bytes. The dual port BRAM primitive is used, the data is

mapped by different addressing for reading and writing, comparable to how the data is rearranged in the VGA component.

As mentioned in the previous chapter, flow control has to be slightly different than in the other components, otherwise advantages of the internal pipelined design of the *IDCT* component are lost. The `ready` signal has to be set to low when the available buffer space falls below 64 bytes (less than one complete 8x8 block).

3.8 YCbCr2RGB

Theoretical Background:

JPEG compression works with the YCbCr color model, while VGA uses the RGB color model (RGB is the most common color model). This component transforms the data from YCbCr to RGB. The formula for the transformation is:

$$\begin{aligned} R &= Y + 1.402 \cdot (Cr - 128) \\ G &= Y - 0.34414 \cdot (Cb - 128) - 0.71414 \cdot (Cr - 128) \\ B &= Y + 1.772 \cdot (Cb - 128) \end{aligned}$$

The *IDCT* core outputs data that is level shifted, this means that 128 is subtracted from the values. So the 8 bit values have a range from -128 to 127 (two's complement representation) instead of the range 0 to 255. According to the transformation formula this is what needs to be done for the color components anyways, so just the Y component is corrected by simply adding 128.

To apply the formula, the Y component is multiplied by 1024 (10 bit shift right) and the same is done for the factors that the color components are multiplied with. The factors were rounded to the next integer[11] leading to the following formula with just integer arithmetic:

$$\begin{aligned} R &= 1024 \cdot (Y_{\mathrm{IDCT}} + 128) + 1436 \cdot (Cr_{\mathrm{IDCT}}) \\ G &= 1024 \cdot (Y_{\mathrm{IDCT}} + 128) - 352 \cdot (Cb_{\mathrm{IDCT}}) - 731 \cdot (Cr_{\mathrm{IDCT}}) \\ B &= 1024 \cdot (Y_{\mathrm{IDCT}} + 128) + 1815 \cdot (Cb_{\mathrm{IDCT}}) \end{aligned}$$

This is the formula that is currently implemented.

However after the color transformation the RGB values are not certainly within the range 0 to 255. Due to rounding errors in the *IDCT* step the resulting RGB values of this step may be less than 0 or higher than 255. The *JPEG* standard is aware of this problem and

[11] The rounding errors induced are minimal, one pixel value in rare cases.

requires the RGB values to be cropped to the proper range of 0 to 255 by setting all values higher than 255 to 255 and all negative values to 0. Being just some comparisons, this boundary check was not very difficult to implement.

Unfortunately the design tool was not capable to route the synthesized logic within the timing constraints for the *Virtex-II Pro* FPGA. Basically the multiplier primitives used for the multiplication in the transformation formula are adding too much latency. So this step is divided into two clock cycles. In the first cycle the formula is applied. In the second cycle the boundaries are checked and the results are divided by 1024 (10 bit shift left) which was needed for integer arithmetic.

Flow control and context handling of this component can be held very simple. The `ready` signal coming from outside of the decoder (this is the last component in the pipeline) is fed directly to the *upsampling* component and additionally connected to the *clock enable* pin of the used flip-flops. The `datavalid` signal and the context from the *upsampling* component are delayed by two cycles, keeping them synchronized with the data.

3.9 Slow Control

The the system can be controlled via the embedded PowerPC. The user can connect the board via an UART to a personal computer where a terminal program (like "minicom") is running. The terminal will show the following output:

```
Go: ............ 1
No Go: ......... 2
Reset: ......... 5
```

The user may start and stop the system ("Go"/ "No Go") and initiate a reset ("Reset") by sending the corresponding number (as ASCII character).[12] The usefulness of the slow control will become more evident when upgrading to *MotionJPEG* (cp. chapter 4.3).

[12]Take care not to send special characters like "Page Up" or "Backspace". This may confuse the PowerPC.

4 Upgrade to *MotionJPEG (MJPEG)*

The *JPEG* decoder is designed to decode multiple images in sequence without performance penalty, so the decoder is well suited for the upgrade to *MotionJPEG*. Even so, two matters have to be addressed. First, it is important how the movie is stored, so that the *JPEG* images can be extracted. Second, to display the decoded data on a VGA monitor some images need to be decoded multiple times.

4.1 Container Formats

A container file combines the different data streams of a movie. This can be the video and audio of course, but also additional information like subtitles or a second audio track (e.g. in a different language) are common. To have the all simultaneously needed information stored closely localized in the file, the different data streams are interleaved. Additional non-stream information, like the frame rate of the video or the video and audio codec used, are usually stored in a header and sometimes in a trailer. Figure 4.1 illustrates a container file.

There are many different video container formats, the most popular ones are listed below:

- Audio Video Interleave (`.avi`): A still well-established format, despite being technologically outdated.
- MPEG-4 File Format (`.mp4`): The standard format for MPEG-4 video streams.
- Matroska (`.mkv`): An open source container format.
- Ogg (`.ogg`): The standard format for Ogg streams.

To implement a video decoder at least one container format has to be supported. The decoder has to parse the header and extract the relevant information. This can be the frame rate and duration of the stream, codecs used, etc. The video stream has to be extracted and then decoded according to the parameters defined in the header.

However, there was not enough time to implement a container format interpreter so the input stream has to fulfill some additional restrictions to be interpretable by the decoder developed. Extracting the video stream from the "container" is done by checking for the *JPEG*-markers `EOI` and `SOI`, so the other data (the blue chunks in figure 4.1) must not

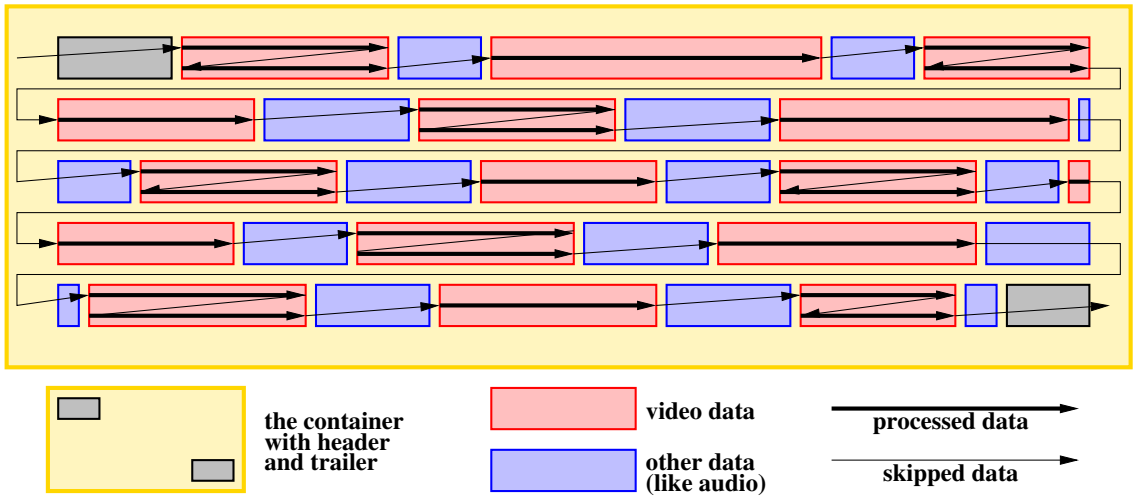


Figure 4.1: A container file is a data structure with a header (and perhaps a trailer), and some data streams - like video and audio data - that are stored in an interleaved manner. Some video frames are repeated to achieve the required framerate.

contain these markers. Since the chance to find such a marker in an audio stream is very good, the file must not include any audio data.

The video is displayed on the monitor with a frame rate of 60 Hz and it can only be adjusted how many times each frame is repeated. So the framerate of the video stream has to be a $\frac{60}{n}$ fps[1] with $n \in \{1, 2, 3, \ldots, 16\}$ and has to be manually adjusted. Otherwise the video would run too fast or too slow.

4.2 Fetching the Stream

The *JPEG* decoder can be used for *MotionJPEG* decoding when fed with a sequence of *JPEG* images. However two issues need to be addressed. First, the framerate of the recorded video may not be the same as the framerate required by the monitor. Second, it has to be assured that no data is lost due to the branch prediction logic of the *JPEG*-decoders input buffer.

The VGA monitor displays images with 60 Hz refresh rate and in the test setup there is not sufficient on-chip memory to store a complete frame, so the decoder has to provide a new image every $\frac{1}{60}$ s. If the video frame rate is different from 60 Hz, some frames have to be decoded multiple times. Usually the resulting repetition rate is not the same for each frame (cp. figure 4.1). To be played accurately at 60 Hz, a stream recorded with

[1] fps: frames per second

24 fps needs to repeat the every second frame three times and the other frames two times. However other frame rates are harder to map to 60 Hz.

To repeat a frame the SDRAM-address where the image starts is stored, so the frame can be fetched as often as the frame rate requires. This implementation supports videos with a framerate of $\frac{60\,\text{Hz}}{n}$ with $n \in \mathbb{N}$, i.e. the repetition rate for each frame must be equal. To adjust the frame rate two 4 bit counters are used, a reference counter set by user and a status counter counting how often the image has already been read. If they are equal the decoder advances to the next frame.

Figure 4.2 shows the basic components and connections needed to fetch the stream and adjust the frame rate. The `JPEG_EOI` signal is controlled by the input buffer of the *JPEG*-decoder component. This signal simultaneously resets the `Fifo 1` in the input buffer to flush invalid data (cp. chapter 3.2). The next image cannot be fetched until after the *JPEG*-decoder signals `JPEG_EOI`, part of the image would be flushed as well. An `EOI` and `SOI` detection logic searches the incoming data for these markers. The logic for the address calculation may proceed in the following two ways after an `EOI` marker has been detected, depending if the frame shall be repeated or not:

1) If the frame is not going to be repeated, the stream is analyzed further until `SOI` is detected. Then the `OPB_address` of the word containing this `SOI` marker is stored. If the `JPEG_EOI` of the previous frame has not yet been received the address will be reset to the stored value at the time when `JPEG_EOI` is received. Otherwise data is lost due to the reset of the input buffer.
2) If the frame is to be be repeated, the decoder waits for the `JPEG_EOI` and then resets `OPB_address` to the previously stored value.

Note: Since the data is read from SDRAM aligned to 32-bit words it may contain both valid and invalid bytes, feeding the *JPEG*-decoder with only valid data and omitting the then unnecessary reset of the input buffer is not possible without extra effort.

4.3 Slow Control

Video decoding needs some additional parameters to be adjusted. Besides the three options already explained in section 3.9 (“`Go`”, “`No Go`” and “`Reset`”), there are now some (self-explaining) video specific options (`Pause On/Off`, `Next Frame` (while paused), `Faster`, and `Slower`). The following control screen will be accessible via UART:

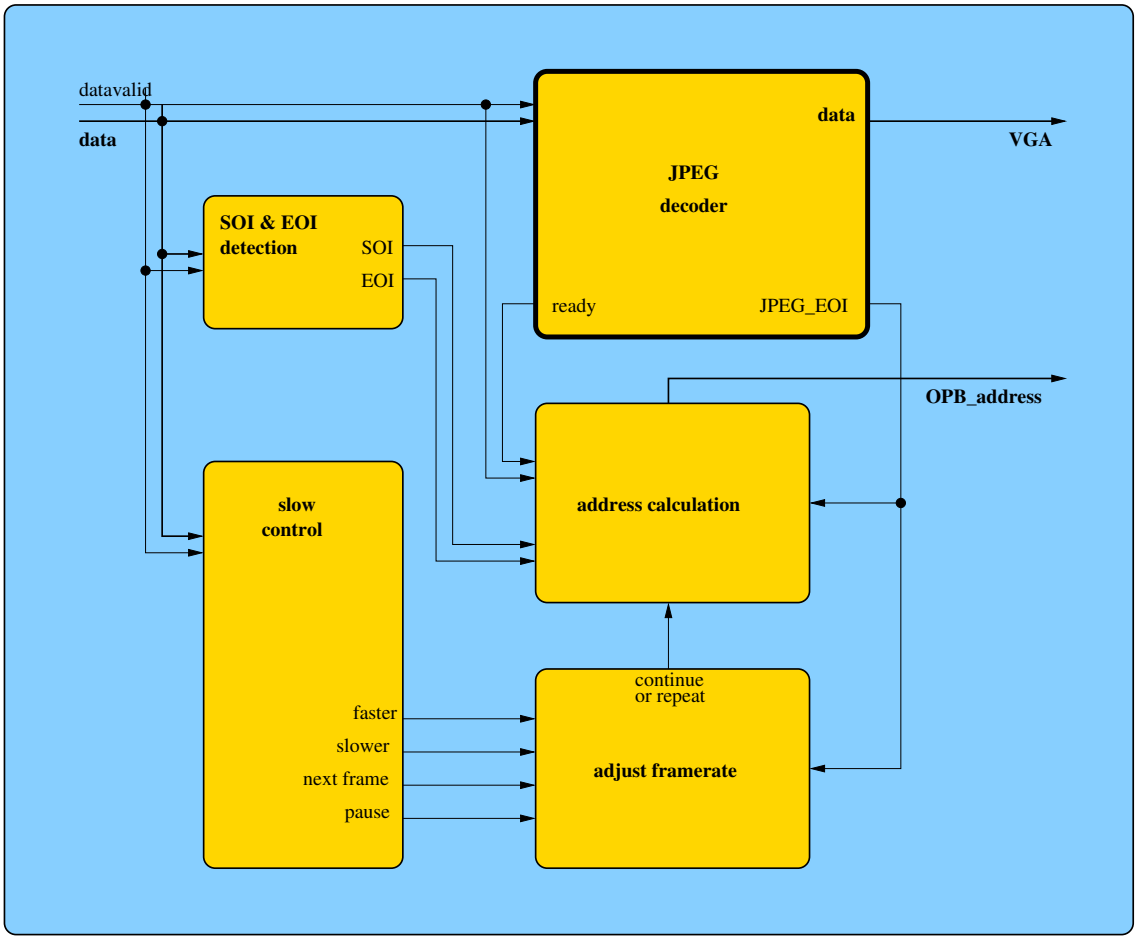


Figure 4.2: The framerate can be adjusted using the slow control. In case the frame is repeated, the OPB address is set back to the start address of the image. In case the frame is not repeated, the data stream is received further until a new image starts. In this case the start address of the new image is stored for later frame repetition.

```
Go: ............ 1
No Go: ......... 2
Reset: ......... 5
Pause On: ..... 14
Pause Off: .... 15
Next Frame: ... 16
Faster: ....... 17
Slower: ....... 18
```

Commands may be issued by sending the referring number.

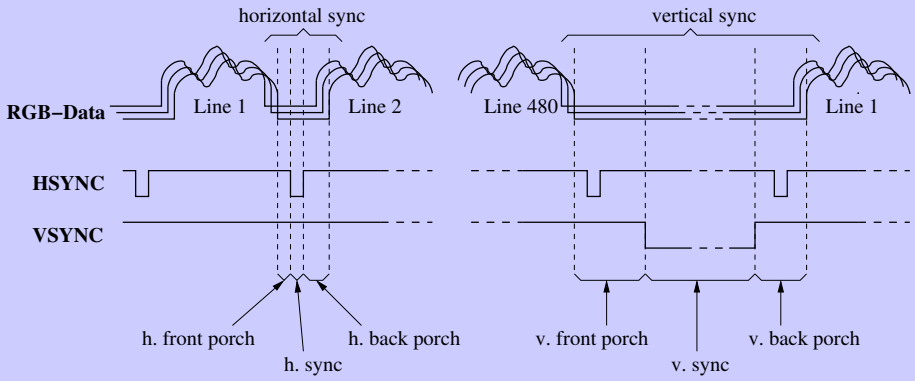
5 Simple VGA Component

To avoid unnecessary traffic on the bus, a basic VGA component directly interfacing the *JPEG* decoder has been implemented. Its implementation is discussed in this chapter.

5.1 Generating the VGA Signals

Theoretical Background:

There are five analog signals needed to operate a VGA monitor, the three RGB color components and two sync signals. The timing is illustrated below.



VGA is an analog standard therefore analog signals have to be generated. This is done by a *digital to analog converter* (DAC) installed on the development board. The DAC requires the following input signals: the three RGB color components (each 8 bits), the horizontal and the vertical sync signals (1 bit), a blank signal (1 bit, overriding the RGB input) and a clock signal (operating at the frequency of pixel change). It outputs the five analog VGA signals.

To operate the DAC at *industry standard* (a resolution of 640x480 pixels and a refresh rate of 60Hz) a new clock domain is introduced. A new pixel is displayed every $\frac{1}{25}\mu$s, therefore the DAC clock is operating at 25 MHz. The clock is generated by a component (`vga_signals`) that has been simplified and ported to VHDL from an existing Verilog

design [GS06]. This component also generates the horizontal and vertical sync signals, and the blank signal. Additional to the VGA signals a line counter and a pixel counter are provided to indicate the position of the current pixel on screen. As all those signals are generated independently from the rest of the system, the data to be displayed has to be synchronized with those signals. This is done by using the true-dualport Block RAM primitives that are also used to buffer and rearrange the data.

Figure 5.1 illustrates the VGA component where the data is passed to the new clock domain (25 MHz) via a dualport Block RAM. Some signals are generated in the 25 MHz clock domain and needed in the 100 MHz domain as well and vice versa. So they have to be synchronized into the other clock domain. The signals from the 100 MHz domain that are needed in the 25 MHz domain are critical since here the transition from a higher frequency to a lower frequency domain may lead to missed signals due to undersampling. However these signals (`width`, `height`, and `sampling`) are not very volatile, so this is not a real problem.

5.2 Buffering and Rearranging of Data

As mentioned before, there is not enough on-chip memory available to buffer a decoded image completely, so the decoder has to decode the data synchronous to the display. Although the decoder has proper backpressure support being able to stop pixel value output at every time, it can not be guaranteed that there is always valid data present. To accomplish reliability it is still necessary to buffer some decoded data in the VGA component.

Yet there is another compelling reason for a buffer between the *JPEG* and the VGA component: the order the pixels are decoded from a *JPEG* file is not the same as the pixels are needed by the display. The *JPEG* decoder will output the pixels MCU-wise, while the VGA core needs them line-wise. The largest supported MCU is a square of 16x16 pixels (cp. chapter 1.3). Therefore a minimum buffer needs to be able to hold up to 32 lines of the image, enough to write 16 lines (one "line of MCUs", cp. figure 5.2) while reading the previous 16 lines, in a different order. This is very BRAM consuming[1] and cannot easily be relocated to SDRAM because of the different read and write order. The OPB bus could not be used in burst mode for reading **and** writing from SDRAM.

The different read and write addresses are calculated separately. To define strictly separated address spaces for reading and writing the signal (`memory_select`) is used. This signal is used as most significant bit (MSB) of the read address and its negation as MSB

[1] 32 rows with a maximum of 640 pixels, 3 components each 1 byte per pixel: $32\text{rows} \cdot 640\frac{\text{pixel}}{\text{row}} \cdot 3\frac{\text{byte}}{\text{pixel}} = 61\,440\text{byte} = 61.44\text{kB}$ (20% of all BRAM available on the *Virtex-II Pro* FPGA).

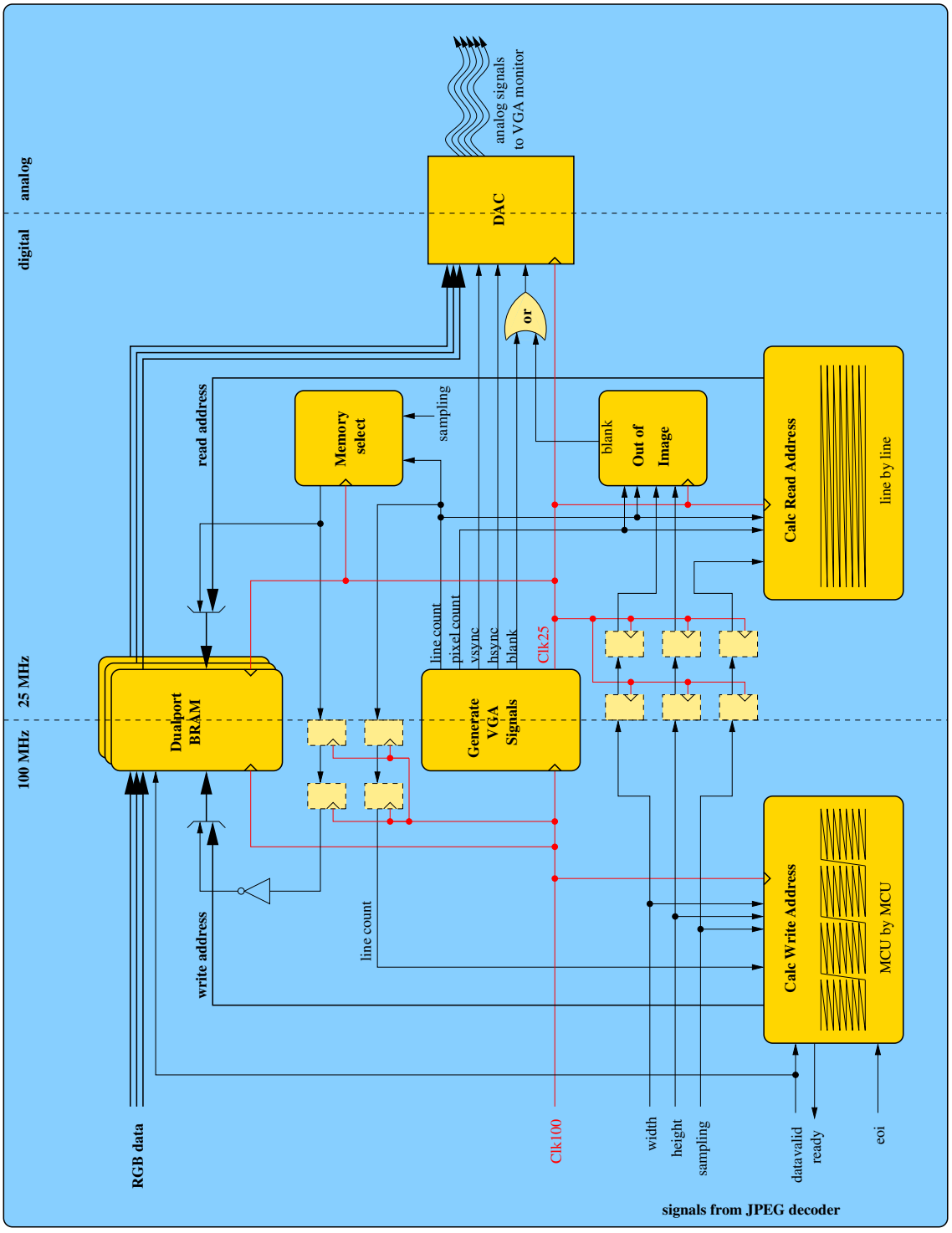


Figure 5.1: The VGA component with the two clock domains and the digital to analog conversion.

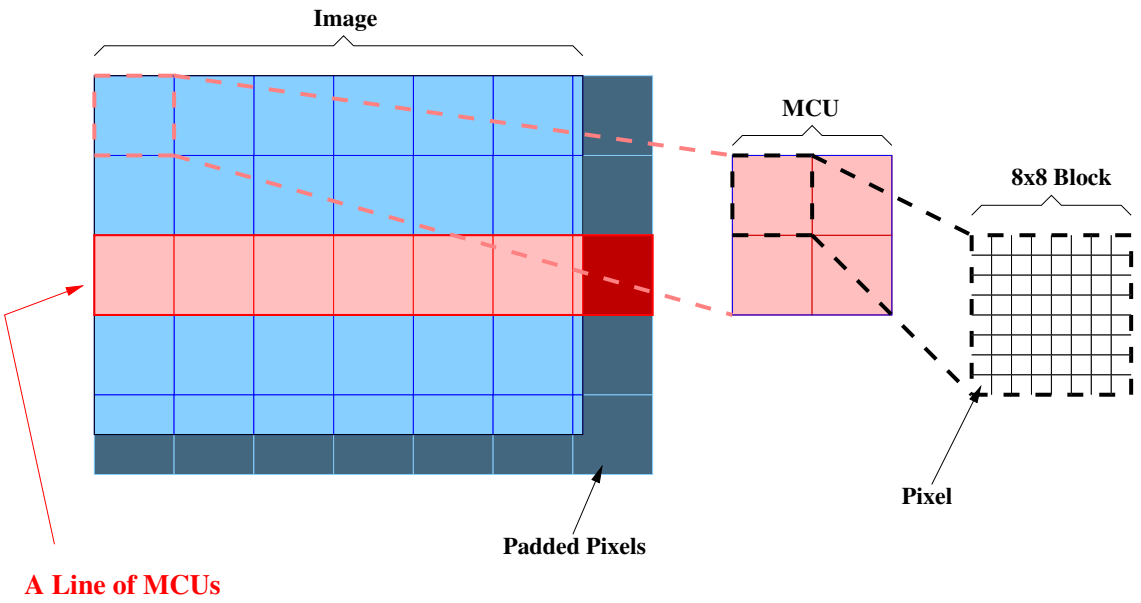


Figure 5.2: Illustration of a "line of MCUs". The VGA component provides buffer space for two lines of MCUs. One for writing the decoded image data and one for reading, used in turns.

of the write address. The signal toggles after every 16^{th} line when the 4:2:0 sampling method is used, and after every 8^{th} line when an other sampling method is used[2].

Write Address Calculation The decoded data has to be presented in a defined order, lost data results in a misaligned image. To assure that not more MCUs than the one "line of MCUs" is written to the buffer, `sampling` and `width` are needed. From these signals the number of MCUs in a line is calculated and the flowcontrol `ready` signal is de-asserted when the line is full. When the complete image has been decoded, writing has to stop as well, waiting for a new VGA frame. Therefore the signals `eoi`, `line_count` and `height` are used. The use of `eoi` additionally synchronizes the data stream in case the image unexpectedly misalignes after all. The write address is incremented when the flow-control signals indicate valid data handover.

Read Address Calculation The data is stored MCU-wise to the Block RAM, not pixel-line after pixel-line. The data of one pixel-line of the image is stored in scattered address spaces. However the scattering is well defined. The correct read address can be calculated from `pixel_count` and `line_count` when considering the sampling method.
If the displayed pixel on the VGA monitor is not inside the area covered by the image, the RGB output values are set to zero by asserting the `blank` signal to the DAC.

[2]The height of a MCU at 4:2:0 sampling is 16 lines, for the other sampling methods it is 8 lines.

6 Results

To analyze the quality and the performance of the developed *JPEG* decoder a whole set of test images with different characteristics and encoding parameters has been created. The images chosen are a picture of some clouds with only continuous variations in color and brightness, on which the *JPEG* algorithm is known to work very well. Furthermore the famous "Lena" image is used, a very popular test image in the field of image processing containing both, smooth areas and sharp edges. The last image is a composition of text with many sharp edges, on which the *JPEG* algorithm is known to work very poorly.
Those images are encoded in all different sampling methods, 4:4:4, 4:2:2, 4:2:0, and gray-scale and with the different quality settings, 20, 40, 60, 80, 90, and 100.

The 72 images created were then used to analyze the developed *JPEG* decoder. This has been done by performing a behavior analysis with *Modelsim*. A testbench has been written that reads a *JPEG* file from the hard disk and provides it to the *JPEG* decoder. The output of the decoder, along with a timestamp, is dumped to a logfile. The testbench permanently asserts its `ready` signal, so the maximum performance of the decoder is measured. The results of the analysis are presented in the next two sections.

6.1 Quality of Decoding

The implementation of the *JPEG* decoder comprises a tradeoff between decoding accuracy and usage of silicon resources. The *IDCT* (cp. chapter 3.6) core internally uses 15-bit wide signals, being compliant with the `IEEE 1180-1990` specifications[IEE91]. However it is still possible to be more precise by using wider internal signals.

It is expected that, although being comparable, the developed hardware decoder does not provide the full accuracy of software decoders using the well-established `libjpeg` [Ind98]. `libjpeg`, executed on a 32-bit processor, is used as a reference implementation for quality analysis.

Each of the figures 6.1, 6.2 and 6.3 shows a) an image decoded by a software decoder using `libjpeg`[1], b) the same image decoded (in a Modelsim simulation) by the developed

[1]The `CImg` library [Tsc] has been used to do the analysis, it uses the `libjpeg` to import *JPEG* images. The C++ code implemented for the verification is available on the CD (cp. Appendix **??**).

hardware *JPEG* decoder, c) an image showing the differences of software and hardware decoder and d) a table with some statistics.

Each of these figures is dealing with an image showing a different theme. The themes are chosen to point out how the decoder is dependent on the content of the image. The images are originally compressed with the same parameters, the quality is set to 80 and the sampling method is 4:2:0. The parameters are chosen to represent commonly used values.

In all three cases the differences between hardware and software decoder, caused by the tradeoff mentioned above, are barely noticeable by comparing the images with the human eye. Only the difference-image and the histogram reveals the minor deficiency of the hardware decoder.

Statistics The following statistics are presented in the histogram:

- **Max:** The maximum value of the difference image.
- **MSE:** The mean squared error between the software and the hardware decoded image.
- **PSNR:** The peak signal to noise ratio between the software and the hardware decoded image. The PSNR is the ratio between the maximum possible value of a signal and the noise. It is commonly used to quantify the quality of image compression algorithms, where the compressed and reconstructed image is considered a noisy approximation of the original image. In this chapter the hardware decoded image reconstruction is considered a noisy approximation of the software decoded image reconstruction.

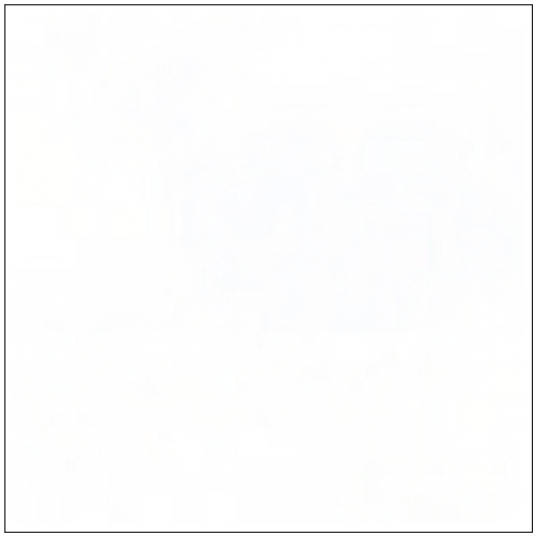
6.2 Decoding Performance

Several parameters have an impact on the decoding time. A set of images has been produced and simulated to point out the different effects of the parameters. Width and height are important later in combination with the VGA component (cp. section 6.3), but their influence on the decoding time is trivial. The dimension of all test images are set to the same value: 256x256 pixels. The varied parameters are:

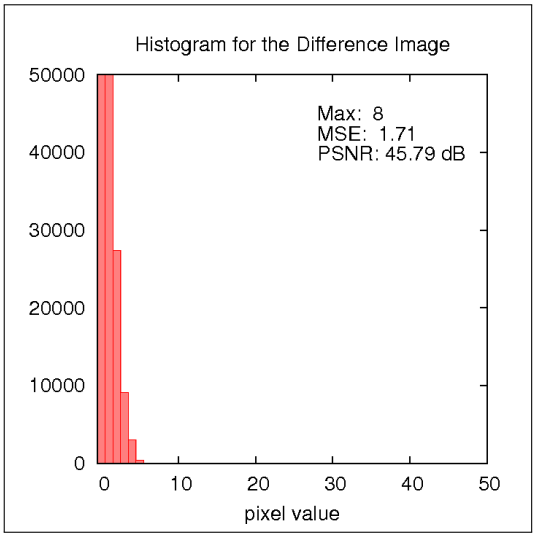
- The quality setting which the image has been compressed with.
- The sampling method: 4:4:4, 4:2:2, 4:2:0 or grayscale.
- The structure of the scene that the image displays.

(a) Decoded by well-established software decoder (libjpeg).

(b) Decoded by developed hardware decoder (simulated with Modelsim).



(c) Difference of software and hardware decoder.



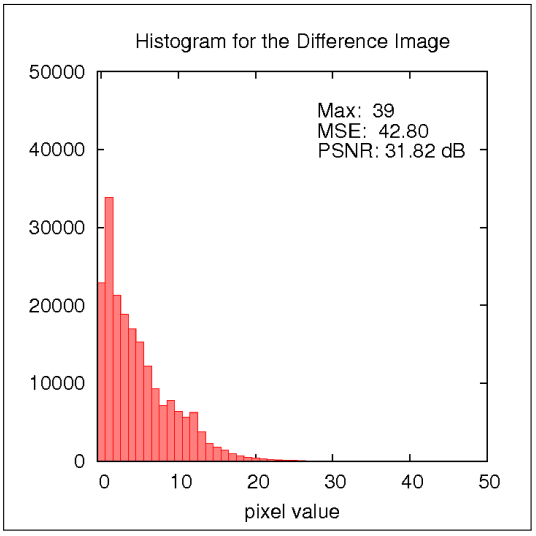
(d) A histogram and some statistics of the differences. The first two channels are cropped, channel 0 has 70845 counts, channel 1 has 85779 counts.

Figure 6.1: `clouds_q80_420.jpg`: An image of some clouds with no sharp edges and only smooth variations in color and brightness, originally compressed with the *JPEG* quality setting 80 and the 4:2:0 sampling method.
The decoder works very well for this kind of image. There is no difference noticeable between the hardware decoder and the reference software decoder, even when the two pictures are subtracted from each other (pixel by pixel for each component separately).

(a) Decoded by well-established software decoder (libjpeg).

(b) Decoded by developed hardware decoder (simulated with Modelsim).

(c) Difference of software and hardware decoder.

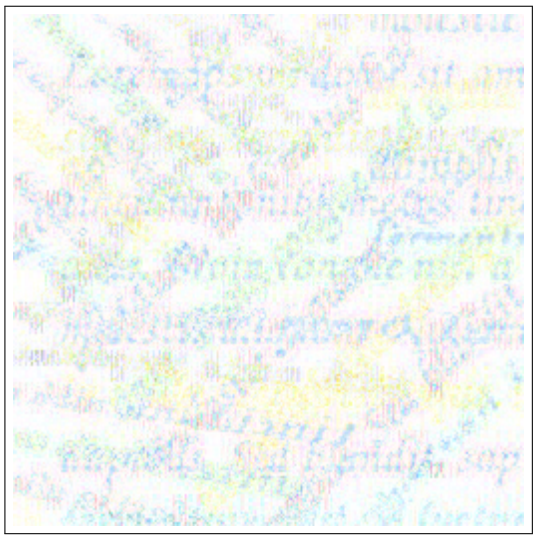


(d) A histogram and some statistics of the differences.

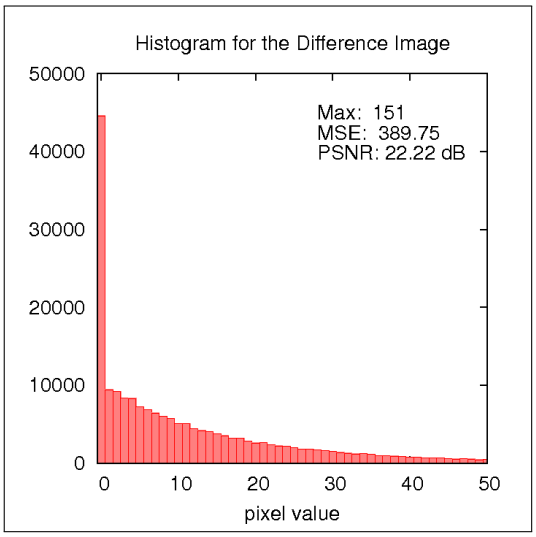
Figure 6.2: `lena_q80_420.jpg`: A photograph of a girl with some continuous tone areas and some areas with more detailed structure, originally compressed with the *JPEG* quality setting 80 and the 4:2:0 sampling method.
The decoder performs well for this kind of image. There is no difference noticeable between the hardware decoder and the reference software decoder. When the two pictures are subtracted from each other (pixel by pixel for each component separately) a close look is needed to spot the differences.

(a) Decoded by well-established software decoder (libjpeg).

(b) Decoded by developed hardware decoder (simulated with Modelsim).



(c) Difference of software and hardware decoder.



(d) A histogram and some statistics of the differences.

Figure 6.3: `lorem_q80_420.jpg`: An image of some printed text with many sharp edges, originally compressed with the *JPEG* quality setting 80 and the 4:2:0 sampling method.
The decoder works satisfyingly for this kind of image. There is hardly any difference noticeable between the hardware decoder and the reference software decoder. However when the two pictures are subtracted from each other (pixel by pixel for each component separately) some noticeable differences are revealed.

Figures 6.4 to 6.6 show the measured decoding times (for a frame) in tabular and plotted representation. It can be observed that a quality setting of 100 leads to a significant loss of performance. This quality setting sets all entries of the quantization tables to one, which essentially eliminates the quantization step. Not much zero-coefficients are to be decoded, thus eliminating the effect of the part of the entropy decoding that performs relatively well. With a lower quality setting, the compression rate and the performance of the entropy decoder increases. Since the entropy decoder is the bottleneck, this has a direct effect on the overall decoding time.

The second issue to be pointed out is the sampling method. Grayscale works best since there is less data to be dealt with, in the whole decoding process. The 4:2:0 sampling method needs nearly half the time of the 4:4:4 sampling method, and 4:2:2 nearly two thirds of 4:4:4. These times are proportional to the amount of data after the entropy decoding and before the `upsampling` component. This effect again uncovers the entropy decoder as bottleneck.

Another mentionable effect is that the structure of the image content has a direct influence on the decoding performance. This is not surprising, the information of sharp edges is held in grand portions in the high frequency coefficients. With high values in those coefficients, they are not completely removed in the `quantization` step and the compression algorithm does not work very well. More data needs to be processed by the entropy decoder.

6.3 Performance Requirements for VGA

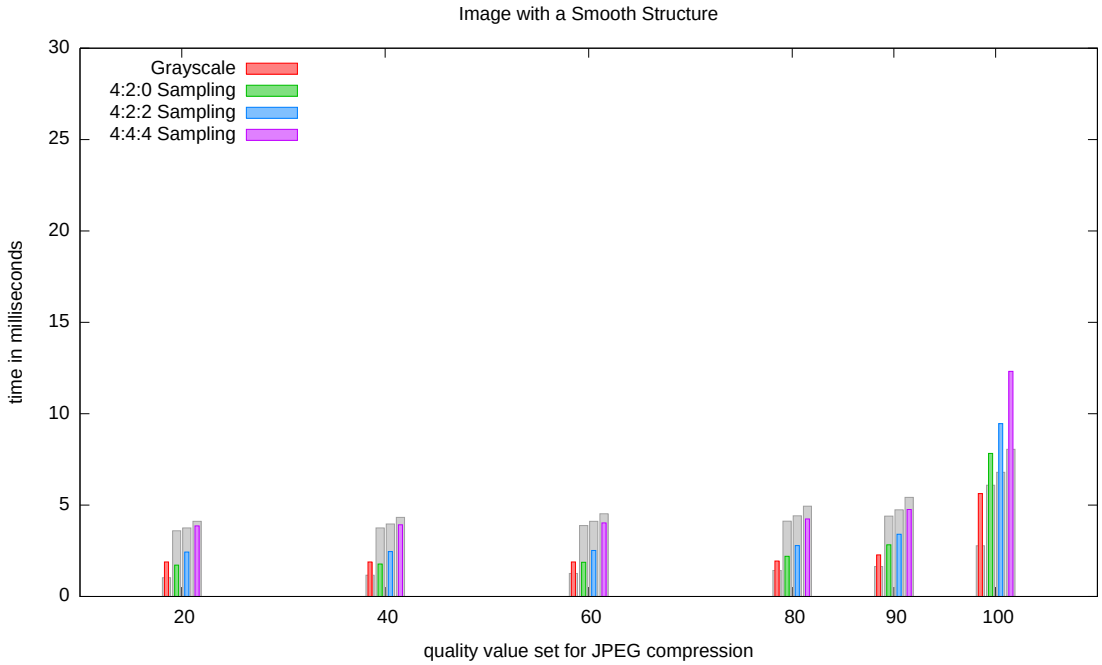
To display the decoded image on a VGA monitor some additional requirements need to be fulfilled. The VGA monitor works at a refresh rate of 60 Hz, so an image has to be decoded in less then a $\frac{1}{60}\ s = 16\frac{2}{3}\ ms$. Since there is not enough memory available to store a whole frame, only a line of MCUs is buffered. There are 524 pixel lines total[2] (including the horizontal and vertical synchronization signals), resulting in

$$\frac{1}{60}\frac{\text{s}}{\text{frame}} \cdot \frac{1}{524\frac{\text{lines}}{\text{frame}}} = \frac{1}{31440}\frac{\text{s}}{\text{line}} = 31.807\frac{\mu\text{s}}{\text{line}}$$

Since the buffer may hold just one line of MCUs, there is no performance gain in reducing the height dimension of the image, only reduction of the width dimension helps. As noticed in chapter 6.2, it also helps to encode the movie with reduced quality or a better sampling rate.

If there are some lines with many sharp edges (for example when subtitles are part of the image), the decoder may not keep up with the VGAs need in that part of the image. It

[2]This value has been ported from [GS06] and proven to work. However other sources suggest 525 lines in total.



(a) The gray boxes refer to the reference measurements using a software decoder (`libjpeg`), executed on a standard 1.5 GHz PC. The colored boxes refer to the hardware decoder.

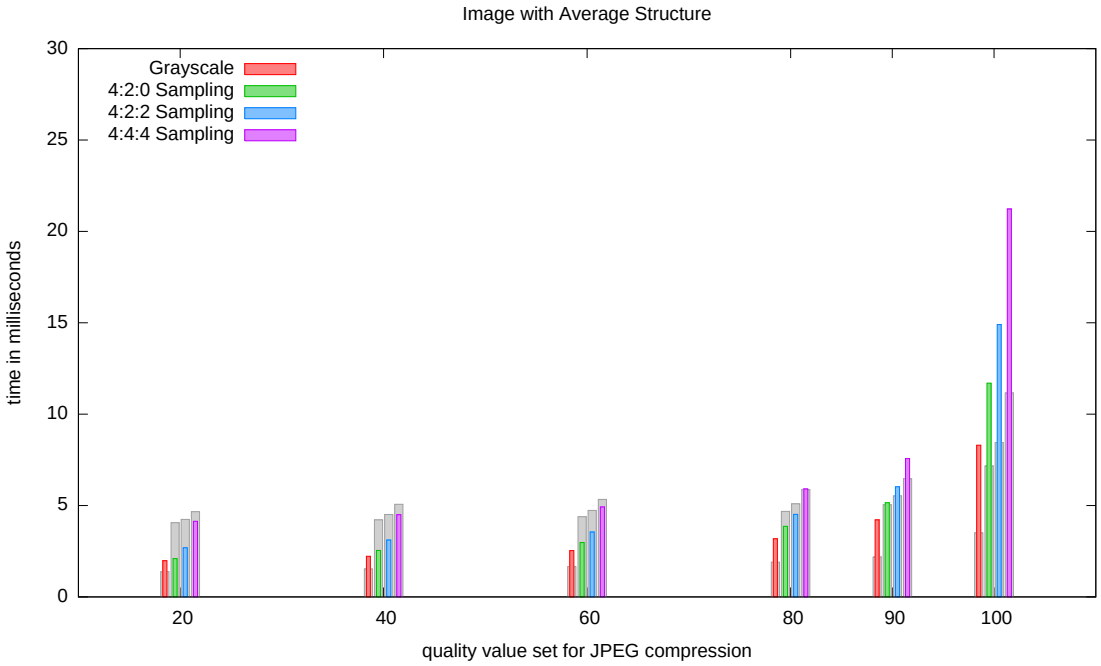
Quality	Gray	4:2:0	4:2:2	4:4:4
20	1.89	1.72	2.43	3.86
40	1.89	1.77	2.46	3.92
60	1.89	1.87	2.52	4.03
80	1.94	2.20	2.78	4.24
90	2.28	2.83	3.41	4.76
100	5.63	7.82	9.46	12.32

(b) Decoding times of the hardware decoder, in milliseconds.

Quality	Gray	4:2:0	4:2:2	4:4:4
20	1.03	3.60	3.75	4.11
40	1.16	3.75	3.96	4.33
60	1.26	3.88	4.11	4.52
80	1.42	4.12	4.41	4.94
90	1.64	4.40	4.74	5.43
100	2.77	6.09	6.79	8.05

(c) Decoding times of a reference software decoder, in milliseconds. (Average value over 1000 measurements.)

Figure 6.4: Decoding times at different compression levels and different sampling rates for an image with very smooth structure(`clouds*.jpg`). Reference data has been measured by a software decoder executed on a standard 1.5 GHz PC.



(a) The gray boxes refer to the reference measurements using a software decoder (`libjpeg`), executed on a standard 1.5 GHz PC. The colored boxes refer to the hardware decoder.

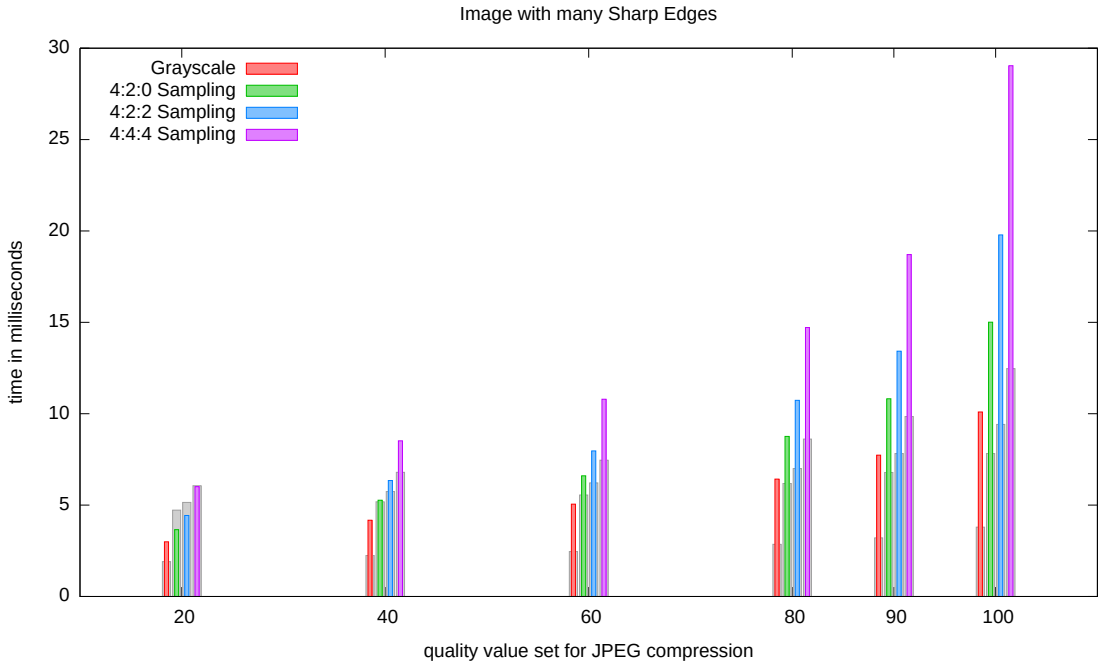
Quality	Gray	4:2:0	4:2:2	4:4:4
20	1.98	2.09	2.69	4.14
40	2.23	2.54	3.11	4.50
60	2.53	2.97	3.55	4.92
80	3.18	3.86	4.52	5.90
90	4.21	5.15	6.02	7.57
100	8.30	11.70	14.90	21.23

(b) Decoding times of the hardware decoder, in milliseconds.

Quality	Gray	4:2:0	4:2:2	4:4:4
20	1.38	4.06	4.24	4.66
40	1.53	4.22	4.51	5.07
60	1.65	4.39	4.73	5.33
80	1.90	4.69	5.10	5.87
90	2.18	5.05	5.53	6.47
100	3.50	7.17	8.44	11.15

(c) Decoding times of a reference software decoder, in milliseconds. (Average value over 1000 measurements.)

Figure 6.5: Decoding times at different compression levels and different sampling rates for an image with average structure (`lena*.jpg`). Reference data has been measured by a software decoder executed on a standard 1.5 GHz PC.



(a) The gray boxes refer to the reference measurements using a software decoder (`libjpeg`), executed on a standard 1.5 GHz PC. The colored boxes refer to the hardware decoder.

Quality	Gray	4:2:0	4:2:2	4:4:4
20	2.99	3.66	4.43	6.01
40	4.17	5.27	6.35	8.51
60	5.05	6.61	7.97	10.79
80	6.42	8.76	10.73	14.72
90	7.73	10.82	13.42	18.71
100	10.09	15.01	19.78	29.04

(b) Decoding times of the hardware decoder, in milliseconds.

Quality	Gray	4:2:0	4:2:2	4:4:4
20	1.90	4.73	5.15	6.06
40	2.24	5.18	5.74	6.80
60	2.45	5.55	6.22	7.46
80	2.85	6.18	7.00	8.62
90	3.21	6.78	7.81	9.84
100	3.80	7.82	9.41	12.46

(c) Decoding times of a reference software decoder, in milliseconds. (Average value over 1000 measurements.)

Figure 6.6: Decoding times at different compression levels and different sampling rates for an image with very sharp structure (`lorem*.jpg`). Reference data has been measured by a software decoder executed on a standard 1.5 GHz PC.

does not matter how fast the rest of the image is decoded. Such an image will flicker and the lines that are hard to decode appear out of order. However the rest of the image will not be affected, if the decoder can catch up to the VGA component again.

6.4 Comparison to a Software Decoder

Figures 6.4 to 6.6 additionally contain measurements of a software decoder (using `libjpeg`).

The software was executed on a standard CPU (*Intel Pentium M*), running at 1.5 GHz, requiring 24.5 W power [Int04]. For comparison, the *Virtex-II Pro* FPGA runs at just 100 MHz and requires only between 0.5 W and 3 W power [Tel05].

Measuring a software implementation is not as accurate as the measurements of the hardware implementation. The standard library for time measurement (`time.h`) does not provide a finer granularity than the operating system time slice for scheduling. Such a time slice is usually about 10 ms and therefore not exact enough for this measurement. However there is a way to measure the CPU cycles used by directly reading out the TSC[3] of the x86-processor using the assembly instruction RDTSC[4]. The measured value may contain some scheduling cycles of the operating system, which is fair since this is the environment a software implementation usually works in. 1000 measurements per image have been made and the average value, including an average part of scheduling cycles, have been recorded.

The hardware and the software decoder timewise perform in the same order of magnitude. It is noticeable that the software decoder outruns the hardware decoder when dealing with grayscale images. The pipelined design of the hardware decoder does not benefit from the fact that for grayscale images the `upsampling` and the `YCbCr2RGB` components are not necessary.
High quality images also perform better on the software side, obviously entropy decoding is not the heavy part for the software decoding.
For 4:2:0 and 4:2:2 sampled images however the hardware decoder does better. The MCUs are spread over a wide range of addresses. This may lead to more cache misses in the CPU that executes the software decoder.

6.5 Proof of Operation

The *JPEG* decoder is successfully put into operation, figure 6.7 shows the development board displaying a *MotionJPEG* movie on a connected VGA monitor. The movie is played

[3] TSC: time stamp counter.
[4] The patch applied on the `libjpeg` source code can be found on the CD provided (cp. Appendix **??**)

as intended, without any flickering or other disturbances and can be controlled via the serial port.

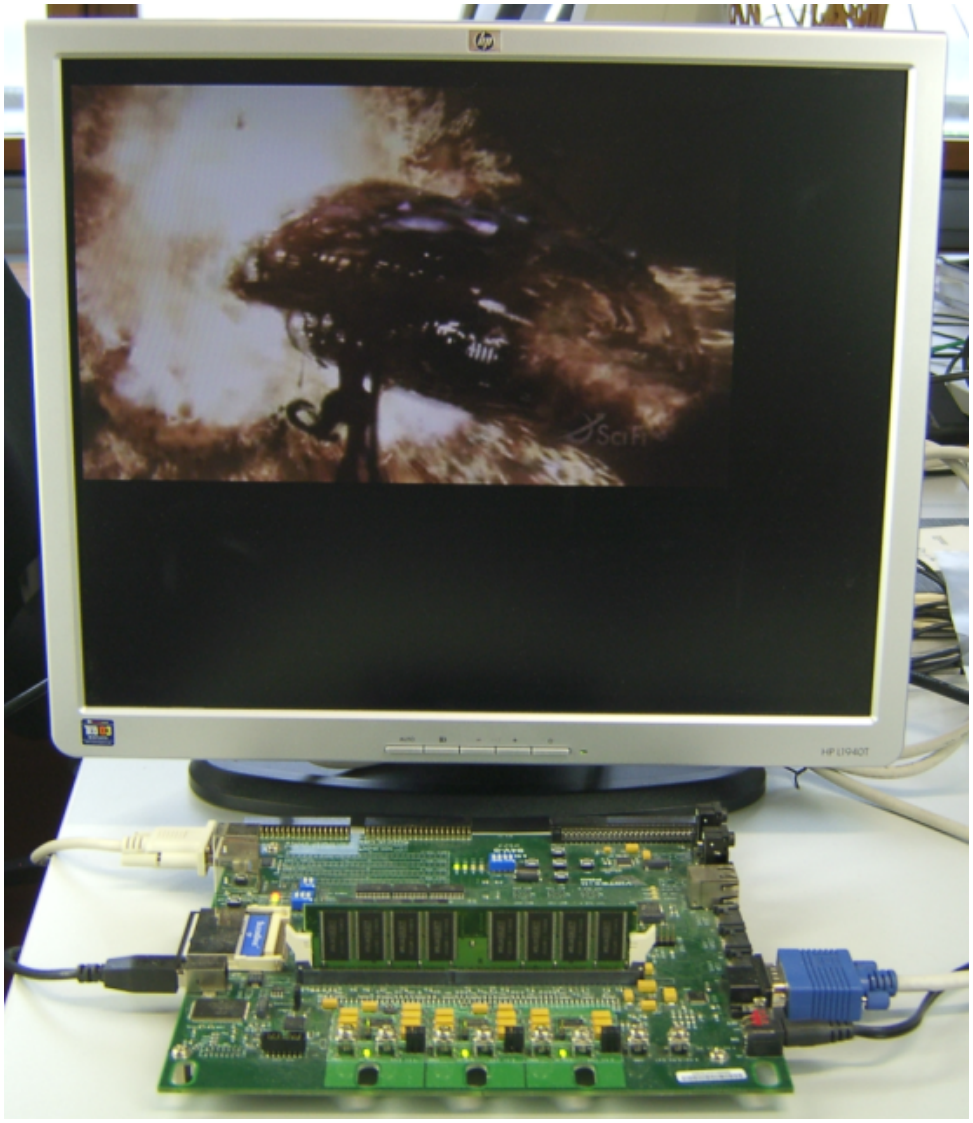


Figure 6.7: The *Xilinx XUP Virtex-II Pro Development System* board decoding an *MotionJPEG* movie and displaying it on a VGA monitor.

7 Future Prospects

The *JPEG* decoder implemented in this thesis is fully functional, yet there are ideas for quality improvement. The decoder may be used in various ways as demonstration and test application for Norbert Abel's *hardware scheduler* [Abe05]. Additionally, it is imaginable to use the decoder as a good basis for future projects like MPEG decoding.

Some ideas for enhancement shall now be discussed before describing the use of the decoder in possible future projects.

7.1 Tuning of the JPEG decoder

The efficiency in term of device usage can be improved by revising the implementation of the `upsampling` component (cp. chapter 3.7). Instead of the shift registers, an enhanced implementation will use Block RAM primitives. Such an implementation would be similar to the design of the VGA component (cp. chapter 5) since a major part of this task is buffering and rearranging data as well. The address calculations are more complex since different components of the image need different reordering.

To improve the decoder performance, the bottleneck of the decoding pipeline, the entropy decoding, has to be reviewed. For common images the decoder spends about half of the time in the `write_zeroes_eob` state and only the rest of the time decoding (cp. figure 7.1). This is not necessary, the decoding can proceed while the zeroes are written,

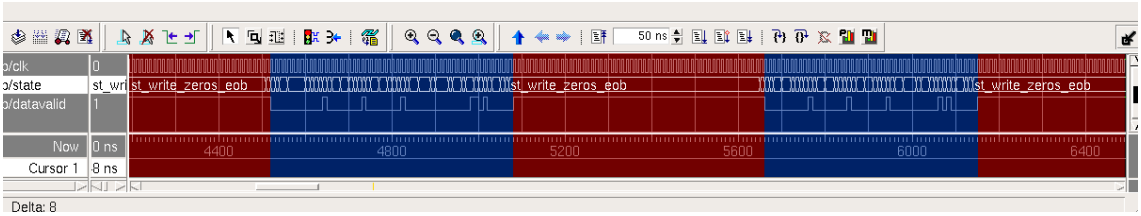


Figure 7.1: A snapshot taken from the simulation tool `Modelsim`, showing the states of the entropy decoder main finite state machine. Decoding is done only in the blue colored cycles, the red colored cycles are used to write the tailing zeros of a 8x8 block to the output. Those two phases can be done in parallel, resulting in significant performance-enhancement.

the in parallel decoded output data just needs to be buffered in a (small) FIFO. In most cases this would increase the performance considerably, in the best case it could double the performance.

In the particular case of this video decoding system it is feasible to reduce the requirements of the VGA core additionally to the performance improvement of the decoder. This can be accomplished by implementing a framebuffer in SDRAM, with better hardware an on-chip framebuffer is thinkable as well[1]. There is then no need to decode an image in less than $\frac{1}{60}s$ (the VGA frame rate). The framerate of the movie defines the decoding speed and is normally much smaller than $\frac{1}{60}s$. Plus, a framebuffer would solve some problems in the implementation as well. Beside the fact that it is no longer required to decode images multiple times to achieve the video frame rate, a framebuffer can render need for a low latency reverse flow control unnecessary. This would solve the problem with the IDCT deficient reverse flow control.

Another improvement would be the support for RST and DNL markers, which would not be very difficult since it can be done very similar to the handling of the EOI marker. On the other hand this is not very important either as these markers are hardly used.

7.2 Usage in a *Dynamic Partial Reconfiguration* Environment

The decoder is to be used in a *dynamic partial reconfiguration* environment. Two scenarios are especially suitable.

Scheduling of the Pipeline Components The decoder can be divided into its pipeline components, which can then be scheduled. This way an image can be decoded even if the chip does not provide enough space to instantiate the whole decoder. A major problem of this approach is the high data rates that have to be dealt with, which is exactly the topic of J. N. Meier's diploma thesis [Mei] carried out at the *Configurable Hardware* group at the Kirchhoff Institute for Physics.

Reconfiguring Whole Decoder Cores A baseline compliant *JPEG* decoder needs to support four different sampling rates, where just one at a time is used for the decoding of an image. The other sampling methods cannot be omitted, because they may be needed by future images of the stream. However, with dynamically reconfigurable hardware it is possible to replace the decoder with another one. So specialized decoders, that require less resources, can be implemented and a suitable decoder can be dynamically loaded on demand.

[1] Storing one frame would require $640 \cdot 480 \cdot 3\text{Byte} = 900\text{KByte}$ memory.

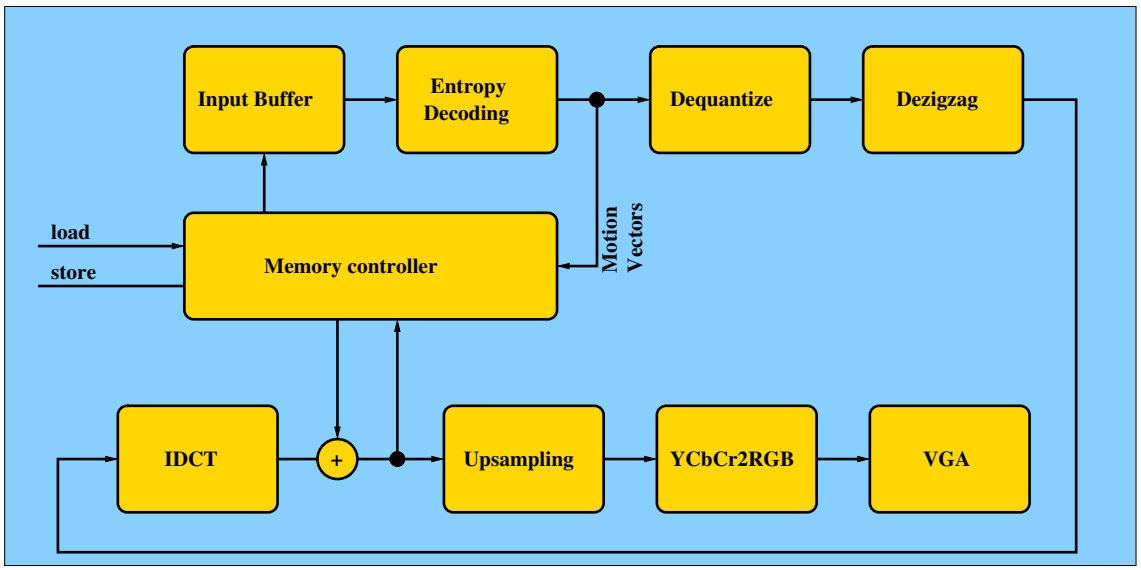


Figure 7.2: MPEG decoding requires an additional memory controller for external memory. Whole decoded image frames need to be preserved for the decoding of subsequent frames.

7.3 Upgrade to MPEG

MPEG bases heavily on the *JPEG* compression techniques, the MPEG i-frames are *JPEG* compressed images and the decoding of MPEG p- and b-frames follow the same principles after all[2]. So an upgrade to MPEG is a natural thought. The MPEG specification is more restrictive than the *JPEG* specification, therefore some parts of the design may be simplified for MPEG decoding. The Huffman tables quantization tables are not stored in the file but specified in the MPEG standard, so they can be hard-coded. However, memory management is much more difficult. The decoding of an MPEG frame may require information from other, previously decoded frames. These frames need to be stored to and loaded from SDRAM, usually not in the same pixel order. The order the pixels are needed is not even predefined but depending on the motion vectors that need to be interpreted. Figure 7.2 outlines a possible MPEG decoding "pipeline", however the block "`Memory controller`" includes complex logic.

[2]MPEG video frames can be i-, p- or b- frames. While i-frames (i for intra) are *JPEG* compressed images, p-frames (p for predictive) only store the difference to the previous image and b-frames (b for bidirectional) rely on previous and subsequent images.

Appendix A

Table of Important Jpeg Markers

Hex	Abbrev.	Full Name	Description
0xFFD8	SOI	Start Of Image	Every *JPEG* image starts with these two bytes. This marker has no payload.
0xFFE0	APP0	Application Segment 0	Immediately following the SOI marker. It starts with the sequence 0x4A46494600 (the zero terminated string "JFIF") which indicates that this is a *JFIF* file. The rest of the payload holds the following information: version number, density in x-and y-axis (dots/inch or dots/cm) or the aspect ratio of the image, an optional thumbnail of the image.
0xFFC0	SOF	Start Of Frame	It includes the following information: the data precision (usually 8 bit), width and height of the image, number of components (usually 1 (gray) or 3 (YCbCr)), sampling factors, the assignment of the *quantization tables* to the components.
0xFFDB	DQT	Define Quantization Table	This is where the *quantization tables* are stored. Usually there is one DQT marker present for each *quantization table*. But be aware that one single DQT marker might as well hold more than one *quantization table*.
0xFFC4	DHT	Define Huffman Table	This is where the *Huffman tables* are stored. Usually there is one DHT marker present for each *Huffman table*. But be aware that one single DHT marker might as well hold more than one *Huffman table*.
0xFFFE	COM	Comment	A zero terminated string.
0xFFDA	SOS	Start Of Scan	This marker includes the following information: the number of components of the image (same as in SOF), the assignment of the *Huffman table*s to the components. It is the last marker of the header.
0xFFD9	EOI	End Of Image	Indicates the end of the *JPEG* image. This marker has no payload.

Appendix B

Header Readout State Machine

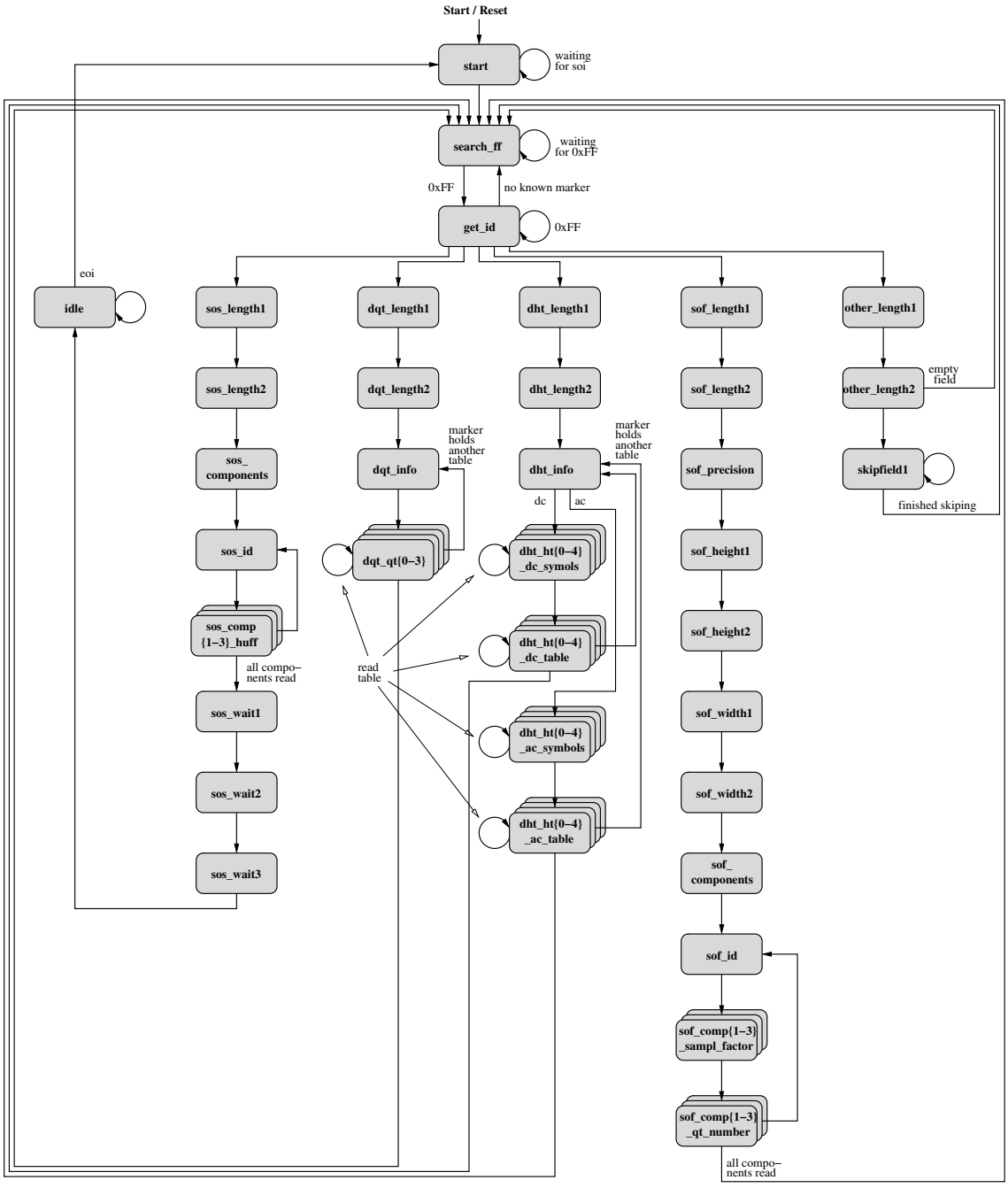


Figure B.1: The header readout state machine. Some states are grouped to keep a clear picture. The dqt and dht markers may contain more than one table but this is not mandatory. For the different tables there may as well be more than one dqt marker (or dht marker respectively) present in the header.

Bibliography

[AAN88] ARAI, Y., AGUI, T. and NAKAJIMA, M., **November 1988**. *"A Fast DCT-SQ Scheme for Images*. *Transactions of the IEICE*, 71(11).

[Abe05] ABEL, N., **2005**. *Schnelle dynamische partielle Rekonfiguration in Hardware mit Inter-Task-Kommunikation*. Master's thesis, Universität Leibzig.

[CCI82] CCIR - COMITÉ CONSULTATIF INTERNATIONAL DES RADIOCOMMUNICATION, **1982**. *Recommendation ITU-R BT.601*. Tech. rep., CCIR - Comité Consultatif International des Radiocommunication.

[GS06] GALLIMORE, E. and SMITH, N., **2006**. *Bitmapped VGA Demo for Digilent XUP-V2P*. URL `http://embedded.olin.edu/xilinx_docs/projects/bitvga-v2p.php`.

[Ham92] HAMILTON, E., **1992**. *JPEG File Interchange Format*. Tech. rep., C-Cube Microsystems.

[IEE91] IEEE, **1991**. *IEEE Standard Specifications for the Implementations of 8 x 8 Inverse Discrete Cosine Transform*.

[Ind98] INDEPENDENT JPEG GROUP, **1998**. URL `http://www.ijg.org/`. `libjpeg` - Version 6b.

[Int04] Intel, **2004**. *Intel Pentium M Processor Datasheet*.

[Jpe92] JPEG - JOINT PHOTOGRAPHIC EXPERTS GROUP, **1992**. *Information technology - Digital compression and coding of continuous-tone still images: Requirements and guidelines*. Tech. rep., Comité Consultatif International Téléphonique et Télégraphique (CCITT).

[Kha03] KHAYAM, S. A., **2003**. *The Discrete Cosine Transformation (DCT): Theory and Application*.

[Mei] MEIER, J. N. *unknown*. Master's thesis, Kirchhoff Institute for Physics. Publication planned.

[NTR74] N.AHMED, T.NATARAJAN and RAO, K. R., **1974**. *Discrete Cosine Transform*. *IEEE Transactions on Computers*, 23(1):90–93.

[Pil07a] PILLAI, L., **2007**. *Video Compression Using DCT*. XAPP 610, Xilinx Inc.

[Pil07b] PILLAI, L., **2007**. *Video Decompression Using IDCT*. XAPP 611, Xilinx Inc.

[Sch03] SCHMIDT, U., **2003**. *Professionelle Videotechnik*. Springer, 3. Auflage ed. ISBN 3-540-43974-9. The book is in german.

[Tel05] TELIKEPALLI, A., **2005**. *Virtex-4 Consumes 50% Power of Virtex-II Pro*. Xilinx.

[Tsc] TSCHUMPERLÉ, D. *The CImg Library*. URL `http://cimg.sourceforge.net`.

[Wat99] WATKINSON, J., **1999**. *MPEG-2*. Focal Press. ISBN 0-240-51510-2.

[Way99] WAYNER, P., **1999**. *Compression Algorithms for Real Programmers*. Morgan Kaufmann. ISBN 0-12-78874-1.

[Wik08] WIKIMEDIA COMMONS, **2008**. URL `http://commons.wikimedia.org/wiki/Image:Dctjpeg.png`.