FORTH-ICS / TR-114 January 1994

An Arithmetic Entropy Codec VLSI chip for JPEG image compression

*Georgios A. Dimitriadis**

Abstract

JPEG, which stands for Joint Photographic Experts Group, will be the next ISO standard on color and grey level still image compression. In this work we have designed and submitted for fabrication a 22000 gate, 150000 transistor VLSI chip, that functions as the entropy coder in JPEG Discrete Cosine Transform (DCT) based image compression using Arithmetic Coding. A static RAM and two PLAs are used to implement 295 identical FSM with 113 states, which are used for the probability estimation. The operations are performed using a four stage pipeline. Forwarding between pipeline stages is performed, in order to resolve any structural and data dependencies and to achieve an average encoding time of 1.5 cycles/bit. A problem arises, during decoding, when successive bits depend on bit-values which are still in the pipeline. In these cases, the following bits are predicted and consequently updated or invalidated, according to the final outcome of the previous decoding operations. Using this prediction scheme, the decoding time is reduced from 4 cycles/bit to an average of 2.4 cycles/bit. The designed chip was implemented in a $1.0\,\mu$m CMOS process, using standard cell technology, and was verified to operate at frequencies up to 20 MHz. An on chip JTAG controller is used to facilitate the post-fabrication test process and to initiate the BIST for the internal PLAs. As Arithmetic Coding achieves 10%-20% higher compression ratios than Huffman, the resulting VLSI chip can be used in environments in which high compression of images is especially important.

*Institute of Computer Science, F.O.R.T.H., Iraklio, Crete, Greece, E-mail: dimitria@ics.forth.gr.

Acknowledgments

I would like to thank Dr. George Papadourakis of ICS-FORTH, who supervised this thesis, for his help and encouragement throughout my graduate studies.

I also would like to thank Prof. Manolis Katevenis, Prof. Apostolos Traganitis and Prof. Panos Trahanias, for participating in my thesis committee. I thank Prof. Christos Chamzas of the Democretus University of Thrace for his helpful early discussions on Arithmetic Coding. Panagiotis Sikas helped me define the microprocessor interface and Chara Xanthaki provided some valuable suggestions during the final phase of the design. Peny Vatsolaki helped with the design software and the technical discussions with Dimitris Daniolos helped clarifying certain aspects of the design.

I would like to thank the Institute of Computer Science of the Foundation for Research and Technology–Hellas (FORTH) for its financial and technical support. This work was supported by the Hellenic VLSI (HVLSI) Esprit program (contract 5692) and the VLSI design software was provided by the EUROCHIP Esprit program (contract 3603), through which the design was also sent for fabrication.

Finally, and most of all, I would like to thank my parents and my family for their love and support throughout the period of my studies in Crete.

Contents

Chapter 1

Introduction

Digital images, both still and moving, have the drawback, that they contain a large amount of data and require a respectively large amount of storage space, making them effectively accessible only to high end, large cost, image processing systems. The advent of multimedia hardware and software in lower cost computer systems, which resulted from advancements in cost and performance of computer hardware, enabled the use of digital images together with sound, in various software applications. Still, the need for storage remains, thereby increasing the cost and lowering the usability of such systems. Therefore, image compression is essential to make storage cost affordable and communication times over networks acceptable. A standardization process engaged in image compression resulted in a currently Draft International Standard (DIS) for compression of grey level and color still images which follow the Joint Photographic Experts Group (JPEG) specifications. Additional standardization efforts are in progress for bilevel and moving images.

The lossy mode of operation in JPEG uses the Discrete Cosine Transform (DCT) followed by entropy coding. Two entropy coders are specified, Huffman and Arithmetic Coding. Most of the software and hardware JPEG implementations today, make use of the Huffman entropy coder. The work presented in this thesis is a VLSI hardware implementation of one of the two entropy coders —arithmetic coding— used by the JPEG DIS. In this chapter, an overview of image compression systems is given, together with the algorithms they use. The second chapter presents the detailed arithmetic encoding and decoding procedures for optimal software and hardware implementations. In the third chapter a functional overview is presented and the detailed VLSI chip implementation is given in chapter 4. The last chapter describes the design environment. Finally, the possible uses of the designed VLSI chip are outlined.

1.1 Image compression systems

An image compression system consists of an *encoder* which reduces the number of bits required to represent an image and the corresponding *decoder* which rebuilds the original image as closely as possible. If the output of the decoder is exactly the same as the input to the encoder, the image compression system is called *lossless*. If the resulting image from the decoder differs from the original, the term *lossy* is used. Such an abstract image compression encoding system is shown in Fig. 1.1. The raw image data (pixels) are first transformed into a number of descriptors [1] which are then coded through a generic *entropy coder*. The latter, first transforms the descriptor data into a symbol sequence according to its statistical model, followed by an entropy coding process. There exists a clear separation between the first stage (encoder model) and the second (entropy encoder). The descriptors of the encoder model are independent of the entropy encoder actually used.

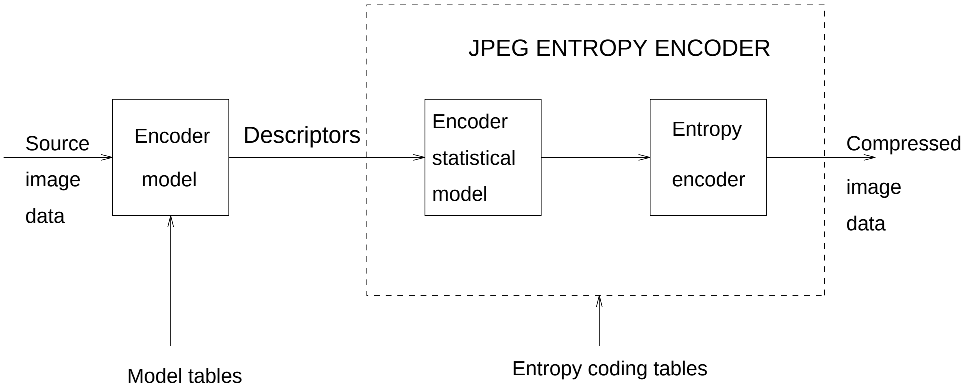


Fig. 1.1: Overview of an encoding image compression system

The corresponding decoder structure is shown in Fig. 1.2. The entropy coder [1] is *lossless*, in the sense that the entropy decoder generates exactly the same descriptors which are generated from the coding model and encoded by the entropy coder. The coding model is therefore responsible for introducing any amount of distortion. Clearly, the goal in *image compression systems* is to select an appropriate coding model, which minimizes this loss. Also, during both phases, coding tables are required both for the coding model and the entropy coder, such as the probabilities of the symbols in non-adaptive systems. These tables can be fixed and can be known *a priori*, both to the encoder and decoder, but they also can be transmitted by the encoder to the decoder as part of the coded image data. An alternative is to use an *algorithm* to generate these tables for the entropy coder, in which case the entropy coder is characterized as adaptive. The same algorithm is applied by the decoder to generate identical entropy decoding tables. This is the case with *arithmetic coding*. Two coding models, one lossless and one lossy are used by the JPEG standard, and they are shortly described in the following section.

1.2 Coding models used by JPEG

The coding model selected by a compression system dictates to one extend the compression that can be achieved. The simplest coding model is to use the raw pixel values as descriptors. Although simple, this coding model is very inefficient. A better approach is to use Differential pulse code modulation (DPCM).

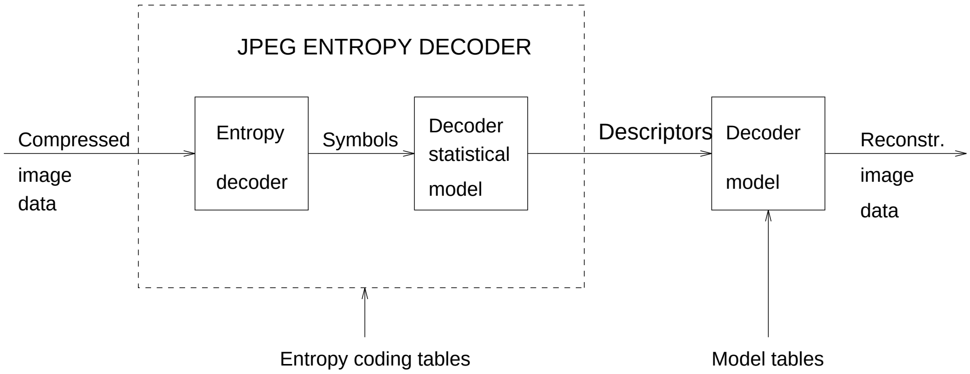


Fig. 1.2: Overview of a decoding image compression system

[1] in this text we will refer usually as coding both to encoding and decoding.

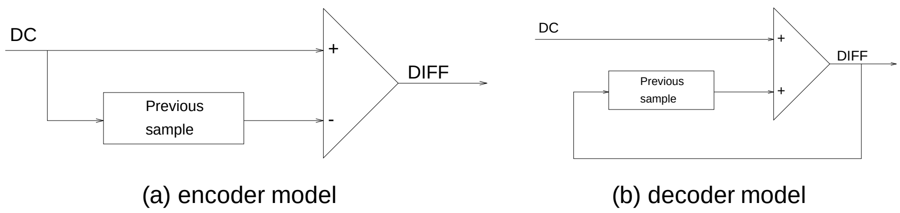


Fig. 1.3: DPCM coding model

1.2.1 DPCM based coding model

Differential pulse code modulation (DPCM) predicts the next sample value by looking at the current sample, therefore it uses as descriptors the difference between the current and the previous value. In continuous tone images the differences between neighbor image pixels are expected to be small. The advantage of this coding model over the simple one, which uses the raw pixels, is that by using as descriptors the difference values, we have generated a set of descriptors in which few of them are *very* probable (small differences), allowing the entropy coder to achieve higher compression ratios. The DPCM encoder and decoder models are shown in Fig. 1.3.

1.2.2 DCT coding model

The Discrete Cosine Transform (DCT) and its associated quantization and dequantization steps are used by the JPEG lossy mode of operation. As we will describe in the next chapter, the eye has the ability to see only particular *spatial frequencies*. Therefore the use of a coding model in which the pixel values are transformed into spatial frequencies allows the separation of the components, which the eye can see from the components that are imperceptible. The transformation selected is the two-dimensional 8×8 DCT transform, which is based on the 1-dimensional 8-point DCT transform. The image is partitioned in a number of 8x8 blocks, on which the transformation is applied. Few of the resulting DCT coefficients —spatial frequencies— have most of the "video energy", concentrated mainly in low frequency components, with the rest of the coefficients collapsing almost to zero. The DCT transformation stage is followed by a step, in which information is further reduced under control of the coding process. The quantization step is the primary source of loss. The following equations describe the 8x8 Forward and Inverse DCT for generic image data $f(i,j)$ and DCT coefficients $F(u,v)$:

$$F(u,v) = \frac{1}{4}C(u)C(v)\sum_{i=0}^{7}\sum_{j=0}^{7} f(i,j)\cos(\frac{(2i+1)u\pi}{16})\cos(\frac{(2j+1)v\pi}{16}) \tag{1.1}$$

$$\text{where } C(x) = \begin{cases} \frac{1}{\sqrt{2}}, & x = 0 \\ 1, & \text{otherwise} \end{cases}$$

$$f(i,j) = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v)\cos(\frac{(2i+1)u\pi}{16})\cos(\frac{(2j+1)v\pi}{16}) \tag{1.2}$$

An almost exhaustive survey on DCT implementations is given in [2]. The resulting 8x8 blocks of DCT coefficients are ordered according to the zig-zag sequence shown in Fig. 1.4. A simple, but naive implementation of a 2-D DCT from simple 1-D DCT almost takes 1000 multiplications and 900 additions. This large number of operations would make the DCT prohibitive, if not more efficient implementations were known. The most efficient fully 2-D DCT [3] requires

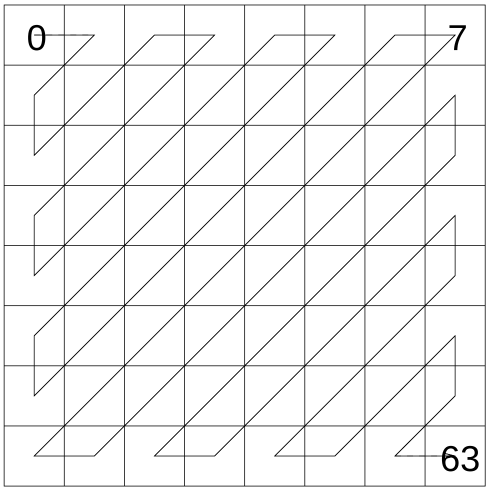


Fig. 1.4: DCT coefficient zig-zag coding order

only about 54 multiplications, 464 additions and 6 arithmetic shifts. Other fast DCT implementations are also presented by Ligtenberg in [4].

1.2.3 Quantization of DCT coefficients

By using the DCT transform, we have converted the raw pixel data of an image to the spatial frequencies which are present in the image. The next step is to eliminate the frequencies which are not visible to the eye as we will see in section 2.2. The way to accomplish this is by a controlled quantization of the DCT coefficients to a desired precision, which results in making zero most of the high spatial frequency components. Each of the 64 DCT coefficients can be quantized with its own quantization value.

The following equations describe the uniform quantization of the DCT coefficients $C(u,v)$ for a quantization value of $Q(u,v)$:

$$C(u,v) = \frac{\lfloor F(u,v) \pm (Q(u,v)/2) \rfloor}{Q(u,v)} \tag{1.3}$$

The $\pm$ operation indicates addition for positive DCT coefficients and subtraction for negative coefficients. The resulting quantized DCT coefficients $C(u,v)$ can be utilized for generating the original DCT coefficients using the following equation:

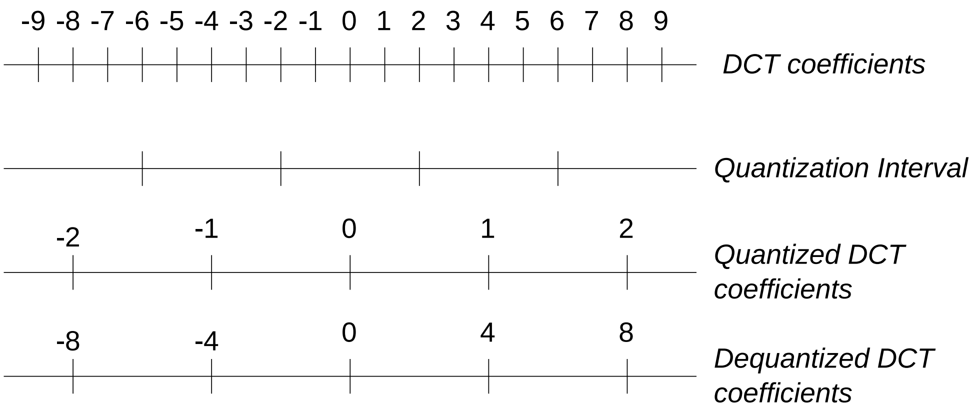


Fig. 1.5: Quantized and dequantized DCT coefficients

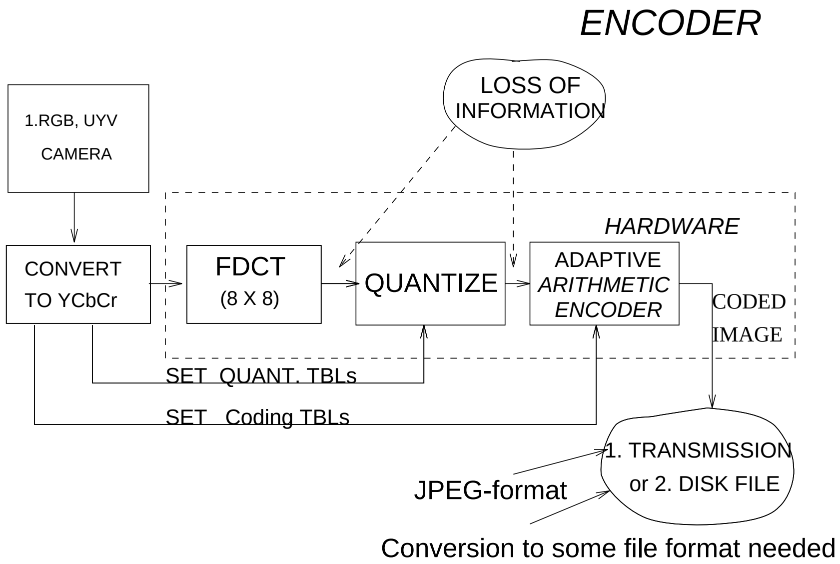


Fig. 1.6: Overview of the encoding process

$$F(u,v) = C(u,v)Q(u,v) \tag{1.4}$$

A respective diagram of the quantization operation is given in Fig. 1.5. The DCT transform adds at most 3 bits of precision to the input pixels, resulting in 11 bit coefficients for 8 bit input pixels. A quantization value of 1 leaves the DCT coefficient intact, whereby a quantization value of 16 reduces it to 7 bits.

1.3 JPEG code stream overview

In Fig. 1.6 and 1.7, a general overview of the DCT based encoding and decoding processes respectively are shown. During encoding, the raw image data are first converted to the internal color representation and accordingly the external supplied coding table parameters are setup in the FDCT and quantization steps. The results from the previous stages are fed to the entropy coder, which outputs the code stream. In cases where the code stream is used by applications, which also have initiated the encoding of the image the resulting code stream can remain in whatever format the applications had chosen for it. In cases, that the coded image data is transmitted to other applications, specific formats are defined by JPEG, which include all the necessary information, in order for the other applications to decode successfully the code stream. This format will be described shortly in the following paragraphs.

The decoder receives the code stream, which is fed into the entropy decoder, followed by the dequantization and IDCT steps. Included in the code stream or known a priori, the table specifications are used by the above processes and they are utilized for reconstructing the original image. The loss of information in the coder is in the quantization and the IDCT steps and therefore, the final image differs from the original as specified by the coding parameters.

The code-stream which is defined by the JPEG specifications contain all the necessary data to decode a coded image. Other parameters of the images, such as the aspect ratio and the meaning of the image components, are omitted from the specifications since they are application depended. JPEG has defined the method through which such application depended parameters can be included in the code-stream. Three different kinds of code streams are defined:

- Interchange format for compressed image data.

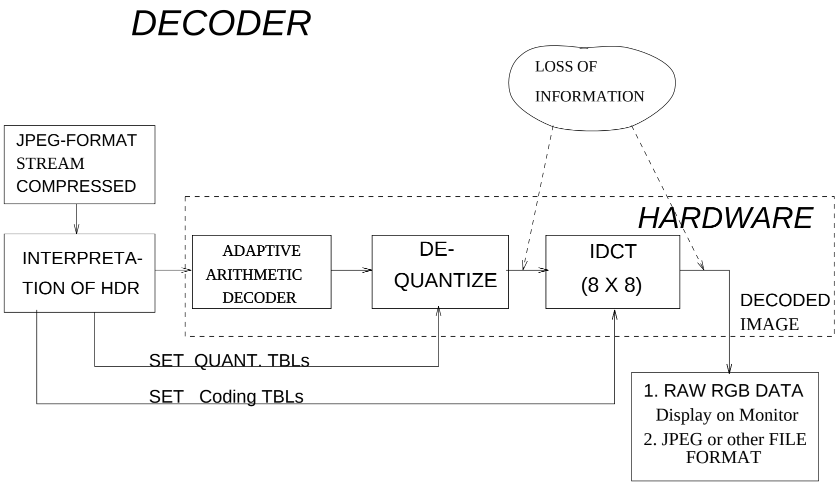


Fig. 1.7: Overview of the decoding process

- Abbreviated format for compressed image data.
- Abbreviated format for table specification data.

The interchange format includes both the image coded data and the table specifications, which are necessary for the coding model and the entropy coder. The abbreviated format for compressed data contains only the coded data, in which case the table specifications should already have been defined and the abbreviated format for table specification contains exactly only these table specifications. The code stream used by JPEG is composed from coded image data and marker segments which are identified by a unique two byte sequence. These marker codes contain the necessary table information in the interchange and table specification formats.

Two modes of operation are defined, sequential and progressive. In the sequential mode, the top row or segment of data in the image is coded and either stored or distributed; then a second row or segment is coded and either stored or distributed. The process continues until the entire image has been coded. In each case, each segment is fully coded. The decoder decodes the segments in the order in which they were coded.

In the progressive mode of operation, an entire image is first coded at some level of quality which is less than the final quality needed. This same image is then coded again, but at a higher level of quality. Each time the image is coded, only the incremental information is needed to improve the transmitted quality. The process is repeated successively until the desired level of quality has been obtained. The decoder decodes the first quality level image and the incremental information in the same order in which they were coded.

A single image consists of a single frame (non-hierarchical) or of more than one frame (hierarchical) coding processes. Each frame is composed from one or more scans and each scan uses one or more image components. A frame header supplies information applicable to the whole frame and each scan contains an appropriate scan-header for specifications applicable to each particular scan. Each scan may contain one or more entropy coded segments, also called restart intervals which are delimited by the restart-interval marker code. Restart intervals are used to reset the entropy coding process parameters in transmission cases over noisy channels in order to enable error recovery. The corresponding structure is shown in Fig. 1.8.

JPEGs notion of an image is a set of image components up to 255. Each component is divided into 8x8 blocks for the DCT transformation, which are named Memory Data Units (MDU) and either each component is coded continuously or its MDUs are coded interleaved. In the case that the image is coded in more the one scans, and each scan uses a different set of components, the term MCU (Memory Coded Unit) is used to define the set of MDU to be coded.

The various modes defined by JPEG are categorized to:

- baseline sequential.
- extended sequential, Huffman coding, 8-bit sample precision.
- progressive, Huffman coding.
- lossless, Huffman coding.
- Differential sequential, Huffman coding.
- Differential progressive, Huffman coding.
- Differential lossless, Huffman coding.
- *extended sequential, arithmetic coding, 8-bit sample precision.*
- extended sequential, Huffman coding, 12-bit sample precision.
- *extended sequential, arithmetic coding, 12-bit sample precision.*
- progressive, arithmetic coding.
- lossless, arithmetic coding.
- Differential sequential, arithmetic coding.
- Differential progressive, arithmetic coding.
- Differential lossless, arithmetic coding.

From the above modes of operation which use arithmetic coding, the following two are supported by our hardware implementation.

- extended sequential, arithmetic coding, 8-bit sample precision.
- extended sequential, arithmetic coding, 12-bit sample precision.

Of particular importance from the set of markers, some of which are shown in Fig. 1.8, is the Define Arithmetic Coding (DAC) marker, which supplies the necessary parameters to the arithmetic entropy statistical model. Also the total number of context areas defined by JPEG is 16, but only two such areas are actually necessary. One for the luminance components Y and one for the chrominance components CbCr, as will be explained in the next chapter.

1.4 Overview of our work

We have designed and submitted for fabrication a one chip VLSI hardware implementation of the statistical model and entropy coding part for DCT based image compression according to the JPEG standard. Our entropy coder uses the more advanced Arithmetic Coding algorithm instead of Huffman which is used in the baseline configuration. Therefore the compression achieved by this implementation is 10-20% higher than using Huffman, as it is shown in [1]. Since the code stream produced by our coder is not compatible with the code stream from the baseline system, our coder is mainly targeted in special application environments in which storage requirements and therefore compression ratio is particular important. In cases in which images coded with our coder have to be transmitted to environments which use Huffman produced code streams, transcoding to a Huffman code stream is possible as will be outlined in sec 2.3.2.

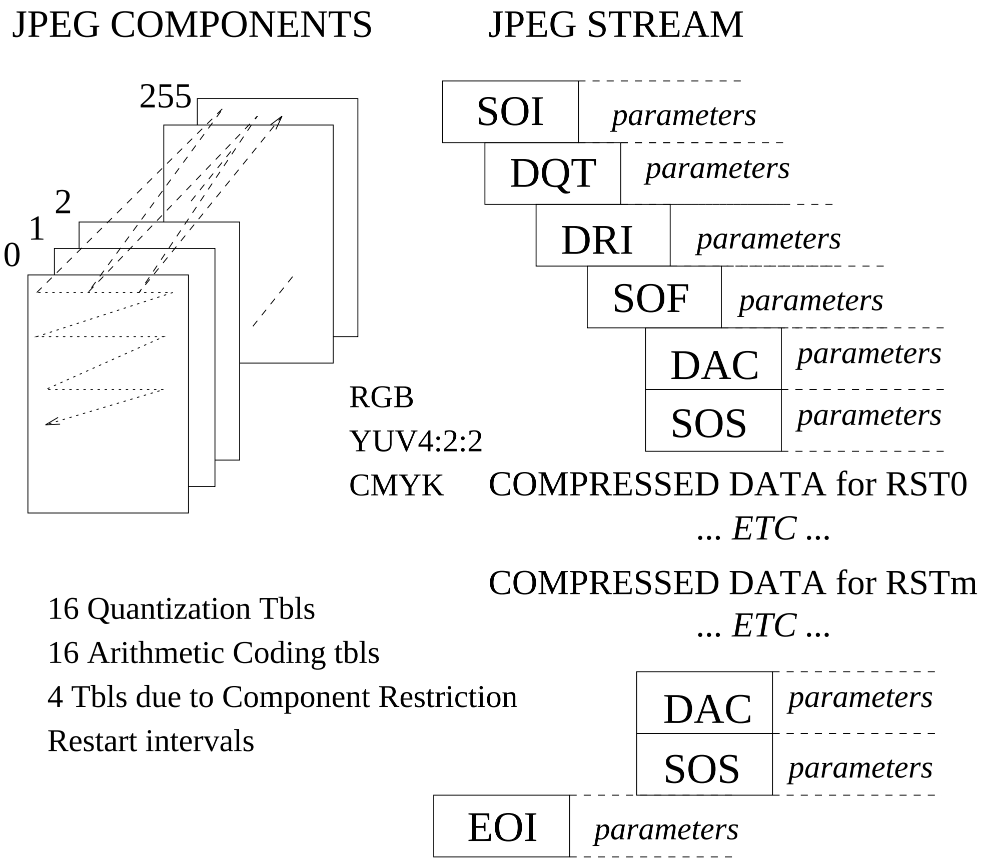


Fig. 1.8: JPEG code stream

The VLSI chip submitted for fabrication uses a 1.0μm CMOS process from European Silicon Structures (ES2) and was designed using the standard cell library provided to us through the EUROCHIP Esprit program. It can operate at frequencies up to 20MHz. The die size is ≈ 50 mm^2 and contains 22000 gates. A 640x8 static RAM is used for the storage locations of the contexts of the statistical model, a 128x32 PLA implements the probability estimation state machine and a small FIFO 8x16 is used to interface to the external source of the DCT coefficients. A general microprocessor bus interface is provided for the setup of the chip, through which also the code stream is received and transmitted.

In the next chapters we will present the procedures and their implementation in our entropy coder. The next chapter introduces basic image compression concepts and presents the arithmetic coding algorithm as it is adopted by the JPEG specifications. The corresponding hardware algorithm, which differs slightly from the algorithm in the JPEG standard, is presented and the method to generate compatible code-streams is described. In chapter three a data flow overview is given and the functionality of each unit is described. The pipeline used to encode and decode the DCT coefficients is introduced and the various data and structural hazards are analyzed. In chapter four, the detailed implementation of each unit is shown and the circuits, which are of particular importance to the function of each unit, are analyzed.

Chapter 2

JPEG arithmetic entropy coding

2.1 Color perception and Color spaces

According to the *trichromatic theory*, ideally, three arrays of sample values are sufficient to represent a color image [5]. There are two ways to combine three array samples, either adding them together, which is used primarily in CRT displays with the three kinds of phosphors or subtracting them, which is used mostly by color printers. One example of an additive color representation is RGB (Red, Green, Blue). The three-dimensional space generated from the values, R, G, and B, define each color as one point in this space, in which the shades of gray are located along the diagonal line. CMY (Cyan, Magenta, Yellow) is a subtractive color space used in printing technology by superimposing certain amount of ink on white paper. The reflected light, which is not absorbed by the ink, hence the term subtractive, is perceived by the eye as a color. In practice another color K (blacK) is added to achieve complete absorption.

Although RGB and CMYK representations are sufficient to specify any color, they are not the most convenient ones in terms of intuitivity. Color spaces using so called *luminance-chrominance* representations are preferred. In these spaces the *luminance* component is associated with the brightness and the *chrominance* component with the Hue (which color is present) and Saturation (how much is present). The following related color spaces are in use.

YUV with Y the luminance and U, V the chrominance components. This color-space is related with the RGB space with the following equations, according to the CCIR Recommendation 601-1:

$$Y = 0.299R + 0.587G + 0.114B \tag{2.1}$$

$$V = R - Y, \quad U = B - Y \tag{2.2}$$

Whenever $R = G = B$, U,V are zero (grey values have no chrominance). YUV color-space is used by both European Phase Alternation Line (PAL) and SEquentiel CouleurA Mémoire (SECAM) television broadcasting systems.

YIQ color-space is just YUV space with the UV components rotated by $33°$. YIQ is used in the American NTSC television system.

YCbCr is the color-space used in JPEG. It has the same Y component as the other color spaces and the Cb, Cr components are related to UV with:

$$Cb = (U/2) + 0.5 \tag{2.3}$$

$$Cr = (V/1.6) + 0.5 \tag{2.4}$$

The above equations guarantee that when the variables U, V take values in the 0 to 1 range, the variables Cb, Cr take values also in the same range.

The use of a luminance-chrominance color-space (YCbCr) in JPEG has the advantage that in these spaces most of the "image information" is concentrated in the luminance component, therefore permitting subsampling of the chrominance components. The reason for this, is the human visual system and its varying sensitivity along the visual spectrum, as will be described in the next section.

2.2 Human visual system sensitivity

The maximum number of samples necessary to represent an image in the luminance-chrominance color-space is bounded by the *sensitivity* of the eye to detect changes in the *brightness* and the *color* of the image displayed. The sensitivity of the eye to detect changes in the *luminance* component is maximum for spatial frequencies about 5 cycles/degree and practically zero at frequencies above 100 cycles/degree [6]. This maximum sensitivity translates to objects of size 0.2 cm for a 40 cm image viewed from a distance of 1 meter. For the same image, the eye has difficulties to resolve objects smaller than about 0.01 cm. A 40 cm image should therefore contain approximately 4000 pixels on either direction (horizontal and vertical). Current high resolution CRT displays don't exceed 2000 pixels per line with an average of 1000 pixels. Clearly the limiting factor here is current display technology.

The *chrominance* sensitivity of the eye is further limited to about 12 cycles/degree. This allows even smaller sampling rates for these components than the sampling rate of the luminance component, without significant loss of the color information, thereby increasing the compression that can be achieved. This is also the main reason that *luminance-chrominance* spaces are preferred in image compression systems as color representations.

2.3 Entropy Coding

Entropy as a measure of information was derived from statistical mechanics. Is is closely related to the probability of a symbol. If a symbol occurs, which is highly probable, the information that it carries is very limited, because it was already expected. However, if it has a low probability, we are surprised and therefore the information which is transmitted is of high value. The amount of information I transferred by a symbol with probability p is given by:

$$I = \log(\frac{1}{p})$$

Therefore the entropy H, which is the average information per symbol s is:

$$H = \sum_{s} p(s) \log(\frac{1}{p(s)})$$

where $p(s)$ is the probability of symbol s. Entropy is a very useful measure because in systems where the probabilities of the symbols are fixed, the entropy is the lower bound for the compression that can be achieved. The most known method of entropy coding is *Huffman coding*, in which code words are assigned according to the probabilities of the symbols. The inefficiency of Huffman coding comes from the coding method in which the *probabilities* assigned to symbols must be *negative powers* of two. The difference between the assigned and the actual probabilities generate this inefficiency.

Symbol	Probability	Subinterval
a	0.20	[0, 0.2)
b	0.30	[0.2, 0.5)
c	0.05	[0.5, 0.55)
d	0.15	[0.55, 0.7)
e	0.20	[0.7, 0.9)
f	0.10	[0.9, 1.0)

Table 2.1: Fixed Probability Model

2.3.1 Arithmetic Coding

Another method which approaches the entropy limit is *arithmetic coding* [1,7,8]. In arithmetic coding the symbols are ordered on the interval from 0 to 1, occupying a subinterval proportional to the probability of each symbol, in a sequence which is known both to the encoder and the decoder. All the symbols precisely fill the interval [0, 1]. An example of a simple alphabet and the associated probabilities of each symbol in the alphabet is given in Table 2.1. According to the symbol coded, the initial interval is *narrowed* to the subinterval associated with the symbol. This subinterval in subdivided again into subintervals proportional to the symbols probabilities and the coding process is repeated. After the input is encoded, any fractional number in the final interval is the coded version of our input. The coding process for the alphabet and for the string "bda" is show graphically in Fig. 2.1. The initial interval is [0, 1). After coding the symbol "b", the interval is narrowed to [0.2, 0.5), which is subdivided into the new subintervals. After seeing "d" the associated interval [0.365, 0.41) is subdivided again and the same process is repeated for the symbol "a". Any number in the final interval [0.365, 0.3740) is the code string. Should it be necessary ro code further symbols, this interval would be subdivided again until no more input data are available.

The decoder can *accurately reproduce* the input from this number and from the probabilities associated with each symbol of the alphabet, which he must know beforehand. The decoder also starts with an interval from 0 to 1 and compares the number received against the subintervals generated from the probabilities of the symbols. The number can only be in one subinterval, therefore the first symbol has been decoded. By subdividing the resulting interval again into subintervals and repeating the above process, the input stream can be fully decoded. The encoder and the decoder must agree as to which interval the boundaries of the subinterval belong (one boundary belongs only to one subinterval and one subinterval has only one inclusive

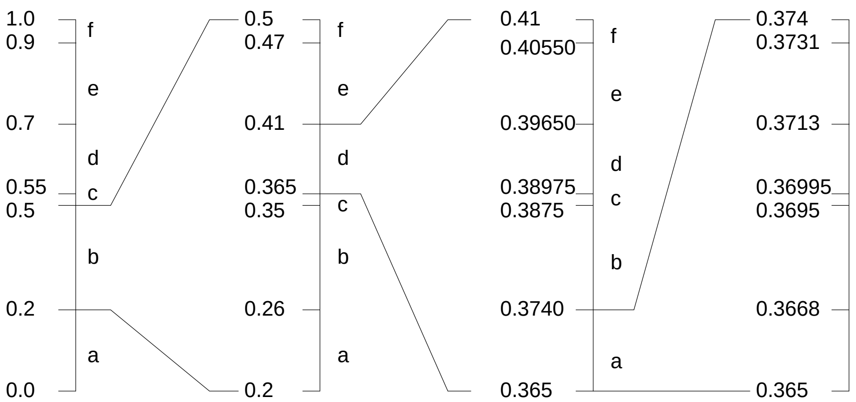


Fig. 2.1: Arithmetic encoding of the string "bda"

boundary). In the above example we assigned the boundary to the lower symbol but the opposite is also possible. It is shown [8] that the entropy achieved by arithmetic coding is:

$$-\log_2(P) = -\sum_{i=0}^{i=N} \log_2(p_i)$$

where P is the final interval and p_i the probabilities of the symbols coded. Therefore, the average number of bits/symbol is very close to the entropy limit.

On the other hand the inefficiencies of the above coding procedure are:

1. The multiplication necessary for the narrowing of the interval to the previous subinterval can limit the performance of such a coder. Also, in adaptive coders, keeping up the exact probabilities of the symbols may use a lot of non-integer computations.

2. Practically infinite accuracy is needed to represent the interval. Using a renormalization procedure an interval always close to one can be obtained, thereby enabling the use of fixed-point arithmetic.

2.3.2 JPEG binary arithmetic coding

JPEG uses *binary arithmetic coding*, that is the alphabet has only two symbols, 0 and 1. Since coding models usually use more than two symbols, the restriction to only two symbols does not affect efficiency, but it does require a mapping from the multi-symbol alphabet to a sequence of binary symbols which results in a so called binary decision tree. Although the entropy limit is the lower bound for compression in stationary systems, this lower bound can be decreased by the statistical model (assignment of the probabilities to the symbols) or by using adaptive coding, which changes this probabilities according to the actual symbols which are coded. The statistical model can be changed using the following methods [1]:

- Changing the alphabet.

- Removing redundancy between symbols of the alphabet.

- Grouping symbols into sets, which have the desired property that small number of symbols are very probable.

The binary arithmetic coder used by JPEG, tries to overcome the limitation of the algorithm mentioned in the previous section and is based on the Q-coder [9–12] incorporating some further enhancements, resulting in the so called QM-coder used by both JPEG/JBIG[1] [13,14]. The QM-coder does not directly assign the subinterval to the 1 or 0 symbols, but rather it assigns intervals to the symbols MPS (Most Probable Symbol) and to LPS (Least Probable Symbol). The assignment of the 0 and 1 symbols to the symbols MPS, LPS is controlled by the coding procedures and they are updated dynamically. Also, the pointer to the interval is defined to point to one of its boundary, therefore this boundary is included in the subinterval and the other boundary is excluded. Two registers are used to store the boundary pointer and the current interval. The Code register (C) contains the pointer to the interval and the interval register (A) the current interval size. Four different orderings of the Symbols MPS, LPS and the C register are possible as they are shown in Fig. 2.2:

1. LPS above MPS (C register points to bottom of interval).

2. LPS above MPS (C register points to top of interval).

[1] Joint Bi-level Image Group is the corresponding standardization effort for bilevel images

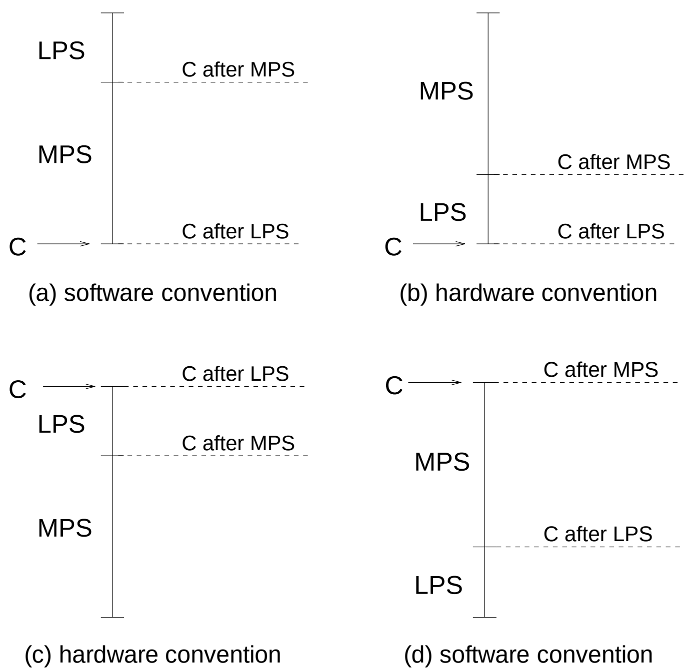


Fig. 2.2: Symbol ordering and code stream conventions

3. MPS above LPS (C register points to bottom of interval).

4. MPS above LPS (C register points to top of interval).

The baseline system in the JPEG specifications, the system with the minimum configuration, is based on the better known Huffman entropy coder. Transcoding between the Huffman coder and the Arithmetic coder is possible since the image descriptors they generate are independent from the underlying entropy coder. Such a transcoding procedure is shown in Fig. 2.3.

2.4 Arithmetic encoding procedures

Considering the first of the above cases, in which the LPS interval is ordered above the MPS interval, we have the following operations to perform after coding an MPS or LPS.

/ LPS interval is ordered above MPS interval */*

After MPS:	$C = C$	*/* C reg remains the same */*
	$A = A(1 - Qe) = A - A \times Qe$	*/* A reg narrowed to MPS */*
After LPS:	$C = C + A(1 - Qe) = C + A - A \times Qe$	*/* C reg incr. by MPS interval */*
	$A = A \times Qe$	*/* A reg assigned LPS interval */*

In order to use fixed-point integer arithmetic we need to use a renormalization procedure in which each time the interval A is dropped below 0.75 both the C and A registers are doubled. The value of 0.75 is chosen so that the multiplication $A \times Qe$ can be replaced with Qe when $0.75 \leq A < 1.5$, in which case the above operations are approximated to:

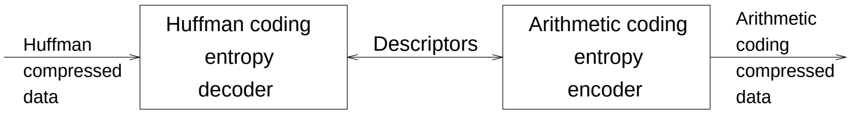


Fig. 2.3: Transcoding between Huffman and Arithmetic Entropy Coder

/ LPS interval is ordered above MPS interval */*
After MPS: $C = C$ *— /* C reg remains the same */*
$A = A(1 - Qe) \approx A - Qe$ *— /* approximation of MPS interval */*

After LPS: $C = C + A(1 - Qe) \approx C + A - Qe$ *— /* C reg approximated */*
$A \approx Qe$ *— /* A reg approximated */*

and incorporating the renormalization procedure into the above operations, we have:

/ LPS interval is ordered above MPS interval */*
After MPS: $C = C$ *— /* C reg remains the same */*
$A = A - Qe$ *— /* A reg is assigned MPS interval */*
if $A <$ 0x8000 **then** *— /* A reg less than 0.75 */*
  renormalize A and C *— /* shift left both A and C, so that */*
end *— /* approximation $A \times Qe \approx Qe$ holds */*

After LPS: $C = C + A - Qe$ *— /* C reg inc by MPS interval */*
$A = Qe$ *— /* A reg assigned LPS interval */*
renormalize A and C *— /* A reg is less than 0.75, renormalize */*

Using the approximation $A \times Qe \approx Qe$ has the problem that when Qe is in the order of 0.5 the size of the MPS subinterval can be as low as 0.25. To avoid this interval size inversion, the assignment of the MPS and LPS to the two intervals is interchanged. When this occurs, we also have $A - Qe < Qe \leq 0.5$ in which case a renormalization must occur. We finally have the following operations to perform:

/ LPS interval is ordered above MPS interval */*
After MPS: $C = C$ *— /* C reg remains the same */*
$A = A - Qe$ *— /* A reg is assigned MPS interval */*
if $A <$ 0x8000 **then begin** *— /* renormalization necessary */*
  if $A < Qe$ **then begin** *— /* interval order inversion */*
    $C = C + A$ *— /* C reg is incremented by MPS interval */*
    $A = Qe$ *— /* A reg is assigned LPS interval */*
  end
  renormalize A and C *— /* renorm A and C */*
end

After LPS: $A = A - Qe$ *— /* A reg is assigned MPS interval */*
if $A \geq Qe$ **begin** *— /* interval order inversion necessary */*
  $C = C + A$ *— /* C reg is incremented by MPS interval */*
  $A = Qe$ *— /* A reg is assigned LPS interval */*
end
renormalize A and C

The number of renormalizations performed are counted by a 3-bit counter (CT) and after 8 logical shift left operations, one byte of compressed data is transferred from the C-register to the output buffer by the `byte_out` procedure. The two registers have the following layout.

`C register`	`0000 cbbb bbbb bsss`	`XXXX XXXX XXXX XXXX`
`A register`	`0`	`aaaa aaaa aaaa aaaa`

The A register is represented by a 16-bit integer. The C register contains a 16-bit low part (`x`-bits), three spacer bits (`s-bits`), and one code byte (`b`-bits). The operation performed on the C register during encoding is an addition between its 28 low significant bits and a 16-bit number as specified by the above routines. Full addition is therefore needed for the lower 16-bits of C and only half addition (incrementing) for the higher 12-bits. The spacer bits are there to absorb any carry over in the low 16-bit part and the `c`-bit signifies such a carry over in the 27 low part bits. After a carry-over has occurred, the byte currently in the output buffer has to be incremented and in the case that there exists a number of buffered 0xFF bytes, they are converted to zeros. The exact `byte_out` procedure follows:

```
byte_out:  T = C >> 16;                 /* T is assigned to C-reg[27:21] */
           if (T > 0xFF) begin          /* if there is a carry out */
                 B = B + 1;             /* increment the byte in the output buffer */
                 if (B == 0xFF) begin   /* if it is incremented to 0xFF */
                       BP = BP + 1;     /* a zero byte must be stuffed */
                       B = 0;           /* byte transmitted, zero byte in buffer */
                 end
                 while (SC > 0) begin   /* Any 0xFF bytes stacked have been */
                       BP = BP + 1;     /* converted to zero and are transmitted */
                       B = 0;
                       SC = SC - 1;
                 end
                 BP = BP + 1;           /* finally buffer current byte from C */
                 B = T;                 /* hold byte for possible future transmission */
           end
```

First, we check for carry over and in this case the byte in the output buffer is incremented. In the case that we have incremented it to 0xFF, we transmit it together with a stuffed zero byte. Next, any stacked 0xFF bytes, which have propagated the carry to the byte in the output buffer, are also transmitted along with a stuffed zero byte and finally the current byte is put into the output buffer. In the case that the output byte is 0xFF, we only have to increment the SC counter:

```
byte_out:  else if (T == 0xFF) begin   /* if output byte is 0xFF */
                 SC = SC + 1;          /* increment 0xFF byte counter */
           end

byte_out:  else if (T < 0xFF) begin     /* only one another output byte */
                 while (SC > 0) begin   /* if there are any stacked Oxff bytes */
                       BP = BP + 1;     /* transmit them */
                       B = 0xFF;
                       BP = BP + 1;     /* and transmit also a stuffed 0x00 byte */
                       B = 0;
                       SC = SC - 1;
                 end
                 BP = BP + 1;
                 B = T;                 /* add current byte to output buffer */
           end
           C = C & 0x7FFFF;             /* clear byte from C register */
```

In the usual case, we check first the SC counter for any stacked `0xff` bytes and transmit them, inserting also a zero byte (byte stuffing) to distinguish them from the JPEG marker codes. Then we add the byte to the output buffer. The byte remains in the output buffer for possible modifications due to future carry overs. Finally the high 11 bits of the C register are cleared.

The encoding renormalization procedure is very simple. The A and C registers are shifted left by one position and the CT is also decremented by one. When the CT counter reaches 0 at least 8 bits of coded data have been shifted into locations 28 to 16 of the C register and no more operations are performed on them. Therefore the `byte_out` procedure is called which stores them in the output buffer until the future carry overs are definitive resolved.

```
renorm_e:  do begin
               A = A << 1;          /* shift left A reg by one */
               C = C << 1;          /* shift left C reg by one */
               CT = CT - 1;         /* CT counts renormalizations */
               if (CT == 0) begin
                   byte_out;        /* time to transmit a byte */
                   CT = 8;          /* reset CT counter */
               end
           end while (A < 0x8000)   /* repeat the process until A ≥ 0.75 */
```

2.5 Arithmetic decoding procedures

The decoder makes use of the same two registers, C and A of the encoder. The C register is now split in two halfs, Cx and C-low. The bits assignment in 32-bit architectures are as follows:

	Cx	C-low
C register:	cccc cccc cccc cccc	bbbb bbbb 0000 0000
A register:	aaaa aaaa aaaa aaaa	0000 0000 0000 0000

Cx points to the subinterval and C-low contains one byte of new data. By comparing Cx to the lower subinterval $A - Qe$ size, we determine if the encoder has added the lower or the upper interval to the C register. Decoding in this way either an MPS or LPS doesn't inform us which symbol was actually coded, because in the case that an LPS subinterval is decoded and a renormalization is needed, the interval assignments might have been interchanged in the encoder, in which case the decoder must test for conditional interchange.

```
                A = A - Qe                  /* narrow A to MPS interval */
C[31:16] ≥ A:   if (A ≥ Qe) begin           /* conditional change */
                    D = 1 - MPS             /* LPS is decoded */
                end else begin
                    D = MPS                 /* MPS is decoded */
                end
                C[31:16] = C[31:16] - A     /* decrement C by MPS interval */
                A = Qe                      /* assign LPS interval to A */
                renorm_d
```

```
              A = A - Qe                      /* narrow A to MPS interval */
C[31:16] < A: if (A < 0x0000 begin
                  D = MPS
              end else begin
                  if (A ≥ Qe) begin
                      D = MPS                 /* MPS is decoded */
                  end else begin
                      D = 1 - MPS             /* LPS is decoded */
                  end
                  renorm_d
              end
```

The decoding renormalization procedure is analogous to the encoding renormalization procedure. The only difference is that in `renorm_d` the test for the CT counter being zero is done in the start of the procedure and then the A and C registers are shifted left.

```
renorm_d:  do begin
               if (CT == 0) begin
                   byte_in;          /* get a new byte of coded data into C[15:8] */
                   CT = 8;           /* reset CT counter */
               end
               A = A << 1;
               C = C << 1;
               CT = CT - 1;
           end while (A < 0x8000)    /* until A ≥ 0.75 */
```

Analogous to the encoders *byte_out* procedure, the decoder has a *byte_in* procedure through which new bytes of coded data are received from the input buffer and stored to bits [15:8] of the C register. The previous bits in these location should have been shifted by that time to the upper 16-bit part of the C register. But before that, it first checks if the new byte is 0xff, in which case the new byte is either a stuffed 0x00 byte or a marker code. If the next byte is not zero then it must be a marker code, in which case a procedure to interpret the marker code is called.

```
byte_in:  BP = BP + 1;                  /* increment input pointer in input buffer */
          if (B == 0xFF) begin          /* can be a stuffed 0x00 or a marker */
              BP = BP + 1;
              if (B == 0) begin         /* it is a 0x00 byte */
                  C = C & 0xFF00;       /* clear bits C[15:8] */
              end else begin
                  interpret_marker;
              end ;
          end else begin
                  C = C + B << 8;       /* put byte in bits C[15:8] */
          end
```

2.6 Optimal Software/Hardware Implementations

As was mentioned in a previous section (2.3.2) two symbol orderings and two code stream conventions are possible. The four combinations are shown in Fig. 2.2. In the previous section we have also seen the necessary operations that have to be performed in the case in which the LPS is ordered above the MPS interval and the C register points to the bottom of the interval. The above case is better suited for software implementations because it requires less computation in the MPS path. If an MPS is to be coded and no renormalization is needed, only one operation $A = A - Qe$ is necessary. In the LPS case, two operations are necessary, which have to be performed serially: $A = A - Qe$ followed by $C = C + A$. However, if the MPS interval is located above the LPS interval the necessary operations that have to be performed are:

```
/* MPS interval is ordered above LPS interval */
After MPS:  A = A - Qe                    /* narrow A to MPS interval */
            if A < 0x8000 then begin      /* if renormalization is needed */
                if A < Qe then begin      /* interval inv. necessary, LPS is coded */
                    A = Qe                /* C remains same, A assigned LPS */
                end else begin            /* No, after all we code a MPS */
                    C = C + Qe            /* increment C reg by LPS interval */
                end
                renormalize A and C
            end else begin
                C = C + Qe                /* increment C reg by LPS interval */
            end
```

```
/* MPS interval is ordered above LPS interval */
After LPS:  A = A - Qe             /* narrow A to MPS interval */
            if A ≥ Qe begin        /* no cond. change, assign LPS interval to A */
                A = Qe
            end else begin
                C = C + Qe         /* cond. change, C points to MPS interval */
            end
            renormalize A and C
```

In this case the MPS path has also two operations $A = A - Qe$ and $C = C + Qe$ but these operations are independent and therefore can be performed in parallel. However this case is not suited very well for software implementations, since the two operations in the MPS path have to be performed serially. The first convention mentioned above is used by the JPEG QM-coder and the second one is used by the Q-coder.

The other convention in which the C register points to the top of the interval is also subdivided into two cases. In the first case the LPS is ordered above the MPS and in the second case the MPS is ordered above the LPS. The former can be made to generate compatible code stream with the case in Fig. 2.2(a) and the latter with the case in Fig. 2.2(b).

Since our aim was to implement the JPEG arithmetic coder in hardware, our implementation followed the Q-coder convention. The different conventions can be made to generate compatible conventions [1]. Our coder generates the same code stream as the QM-coder by following the above procedures for the MPS, LPS symbols and inverting the code stream produced by the encoder. The decoder also inverts its input code-stream so that it can use its own coding procedure, which is the following:

```
                 A = A − Qe                          /* narrow A to MPS interval */
C[31:16] ≥ Qe:   if (A < 0x8000 begin               /* renormalization is necessary */
                      if (A ≥ Qe) begin              /* no interval size inversion */
                           D = MPS                   /* MPS is decoded */
                      end else begin                 /* interval size inversion */
                           D = 1 − MPS               /* LPS is decoded */
                      end
                      C[31:16] = C[31:16] − Qe       /* decrement C by LPS interval */
                      renorm_d                       /* perform renormalizations */
                 else begin
                      C[31:16] = C[31:16] − Qe       /* decrement C by LPS interval */
                      D = MPS                        /* decode a MPS */
                 end
```

```
                 A = A − Qe                 /* narrow A to MPS interval */
C[31:16] < Qe:   if (A ≥ Qe) begin          /* no interval size inversion */
                      D = 1 − MPS           /* LPS is decoded */
                 end else begin             /* interval size inversion */
                      D = MPS               /* MPS is decoded */
                 end
                 A = Qe                     /* C remains, LPS interval assigned to A */
                 renorm_d
```

The above procedures are similar to the software decoding procedures, with the only difference that now the Cx register is compared against the LPS interval Qe and not the MPS interval $A - Qe$. Also, in both conventions, the A register is checked for interval size inversion, but renormalization is always needed in the second case, because Qe is assigned to A, in contrary to the first case, in which renormalization is needed only if $A < 0.75$.

The procedures `byte_out, byte_in, renorm_e, renorm_d` remain the same as in the QM-coder. Also the initialization procedures remain the same. The only additional procedure that needs to be modified is the `flush` procedure which is called at the end of the encoding phase, when all the input bytes have been coded. The procedure for the QM-coder is as follows:

```
flush:   T = C + A − 1            /* T is assigned highest number in final interval */
         T = T and 0xffff0000     /* low 16-bits are cleared */
         if T < C then begin      /* T outside the final interval */
              T = T + 0x8000      /* T again inside the proper interval */
         end
         C = T ≪ CT
         byte_out
         C = C ≪ 8
         byte_out
         discard_final_zeroes
```

C register points to the lower bound of the final interval. As was pointed out, any number in the final interval is acceptable and we would like this number to have as many as possible of its lower order bits zero. Therefore, we assign to the temporary variable T the largest number in the final interval $C + A - 1$ and clear the 16 lower order bits of T. If the resulting number is still in the final interval (the zeroing of the 16 least significant bits may have assigned to T a value lower than the value pointed to by the C register), we transmit the last $CT + 8$ bits. In the case that T falls below C, we add the value 0x8000 to T, which guarantees that T is now in the final interval. The A register is always greater than 0x8000, so the first addition of A-1 followed by zeroing (subtraction) of a maximum value of 0xffff, followed by an addition of 0x8000 results in a value greater than C. Also by adding a value which has only one bit set in position 15, the rest 15-bits remain zero. Therefore T has either the last 16-bits or the last 15-bits zero.

Using the inverse convention in which the MPS is ordered above the LPS interval, this final procedure has to be modified accordingly. In this convention C also points to the bottom of the interval, but the value of C differs from the value of C in the software convention. As the output bytes are inverted in this convention, so that the code stream produced is identical to the code stream produced by the QM-coder, we would like to set most of the low order bits of C. The inverted 16 low order bits of C are assigned to X. The resulting value is compared to the final interval and in the case that it is larger, the new C value is decremented by 0x8000. C register has the low-order 16-bits set and therefore, the subtraction can be implemented by clearing the corresponding bit in position 15.

```
flush:  X = (C and 0x0000ffff) xor 0x0000ffff   /* set X inverted low 16-bits of C */
        C = C or 0x0000ffff                     /* set low order 16-bits of C */
        if (X > A − 1) then
            C = C and 0xffff7fff                /* clear bit 15 of C */
```

2.7 Probability estimation

In the previous section we have seen the operations which have to be performed for coding a symbol using both the software and the hardware convention. In this section we will see how the values Qe for the MPS and LPS are assigned in an adaptive way. The probability estimation used by the QM-coder is based on the renormalization-driven estimation developed for the Q-coder [10]. The estimation process is based on a form of approximate counting, in which whenever a MPS or LPS renormalization occurs, also a new estimate from a table is obtained that provides larger Qe values for LPS renormalizations and smaller Qe values for MPS renormalizations. The estimation process is based on a finite state machine with 113 states [15,10]. Each state is assigned a 14-bit Qe value, two 7-bit ($Next\_Index\_MPS$, $Next\_Index\_LPS$) indexes, which provide the next state for MPS or LPS symbol respectively and a one bit ($Switch\_MPS$) which controls the assignment of 0 or 1 to the MPS symbol. Whenever $Switch\_MPS$ is not zero and an LPS renormalization is required, the symbol assigned to the MPS interval must be inverted. Therefore, the operations to be performed after an MPS renormalization are:

After MPS renormalization:
$I = Index(S)$
$I = Next\_Index\_MPS(I)$
$Index(S) = I$
$Qe(S) = Qe\_Value$

Accordingly the operations for the LPS coding are:

After LPS renormalization:
$I = Index(S)$
if $Switch\_MPS(I) = 1$ begin
    $MPS(S) = 1 - MPS(S)$
end
$I = Next\_Index\_LPS(I)$
$Index(S) = I$
$Qe(S) = Qe\_Value(I)$

where S, is the context which is generated from the statistical model of the coder for the image descriptors (DPCM or DCT based coding model), presented in sec. 1.2. The arithmetic entropy

DC-BINS				
$C = 0$	sign	$SZ < 1P$	$SZ < 1N$	
0	1	2	3	-large
4	5	6	7	-small
8	9	10	11	0
12	13	14	15	+small
16	17	18	19	+large
20 to 34		$X_1, X_2, \ldots, X_{15}$		
35 to 48		$B_2, B_3 \ldots, B_{15}$		

Table 2.2: Binary decisions (bins) assigned for DC DCT coefficients

coding statistical model is described in the next section. If an MPS is coded, the next state machine index for the current context S is looked up with $Next\_Index\_MPS(I)$, which is saved in the context table. Next the value $Qe(S)$ is also retrieved from the state-machine and supplied to the necessary routines for the MPS coding. The LPS path is similar, with the exception that whenever $Switch\_MPS(I)$ is not zero the assignment of 0 or 1 to the MPS interval is interchanged. The derivation of this 113 state probability estimation machine is explained in detail in the book *JPEG, still image data compression standard*, [1].

2.8 Arithmetic Coding Statistical Model

We have seen in the previous sections that JPEG uses DCT and DPCM based coding models, which generate descriptors that are binary arithmetic entropy coded. The descriptors (DCT coefficients) generated from the coding model have to be transformed first into a binary sequence appropriate to be used by the entropy coder. This is the task of the statistical model. The procedures used by the entropy coder were analyzed in the previous chapter and are: `initenc`, `Code_0(S)`,`Code_1(S)`,`Flush`, `initdec`, `Decode(S)`. The procedures `initenc`, `initdec`, initialize the encoder and the decoder respectively, where `Code_0(S)` and `Code_1(S)` code a 0 and 1 decision for a given context S, which contains an index into the probability estimation state-machine. The procedures for updating this index were mentioned in sec. 2.7, and are included in `Code_0(S)` and `Code_1(S)` procedures. This probability estimation state machine is identical for both the encoder and the decoder.

Each descriptor (DCT coefficient) is translated into a set of binary decisions in the encoder and a certain set of binary decisions regenerate the DCT coefficient in the decoder. Since some information is known both to the encoder and the decoder, certain binary decisions do not require coding of a symbol and therefore no coding procedures are executed. Exact flowcharts of the binary decision generating procedures are illustrated in the JPEG specifications [15,16]. The following subsections outline these procedures.

2.8.1 Statistical model for DPCM based coding model

There are 49 contexts assigned to the DPCM coded DCT coefficient, see Table 2.2. The DPCM difference $DIFF$ of the DC DCT-coefficient is assigned to the variable V, on which the following binary decisions are applied [1,15].

1. The first binary decision ($S0$) determines, if V is zero and in that case the coding process for $DIFF$ is finished.

2. If $DIFF$ is not zero then the sign of V is coded with context SS and the value $|V| - 1$ is assigned to the Sz register.

3. The decision ($Sz > 0$) is coded with context SN and SP respectively depending on the sign coded in the previous decision. If ($Sz \leq 0$) the coding process is finished.

4. The decision ($Sz > 2^n - 1$) is coded with context Xn, starting with $n = 1$, until the decision is false or n becomes 15. With this coding procedure the logarithmic magnitude category of the Sz variable is determined.

5. If ($n > 1$) then the decision (Sz¡i¿, ith bit of Sz) is coded, starting with $i = n$ until $i = 1$, with context Mn. The actual magnitude bits of Sz are coded, with the same context Mn.

The number n is bounded by the input precision of the pixel data. For 8-bit pixel data/component the DCT coefficients have 11 bit precision and for 12-bit pixel data the DCT coefficients have 16 bit precision.

Adding the contexts, which were enumerated above we sum up to 32. But the first 4 ($S0$, SS, SN, SP), that is, $DIFF = 0$, sign, $Sz < 1P$, and $Sz < 1N$, are actually conditional probabilities based on the previous difference coded for the same component. This difference is classified into five categories: large negative, small negative, zero, small positive and large positive, each of them having its own set of contexts, for a total of 20 contexts. These categories are created by two parameters L and U, which are set before the coding process in the coder. Each of the two parameters is 4 bits long and specify a range from 0 to 15. The five integer intervals established by the two values are:

$$\begin{array}{lcl}
\text{large negative} & : & DIFF_{prev} \in [-32768, -2^{(U-1)}). \\
\text{small negative} & : & DIFF_{prev} \in [-2^{(U-1)}, -2^{(L-1)}). \\
\text{zero} & : & DIFF_{prev} \in [-2^{(L-1)}, 2^{(L-1)}]. \\
\text{small positive} & : & DIFF_{prev} \in (2^{(L-1)}, 2^{(U-1)}]. \\
\text{large positive} & : & DIFF_{prev} \in [2^{(U-1)}, 32767).
\end{array}$$

The five sets of statistic bins based on the previous DC component $DIFF$ is shown in Table 2.2.

2.8.2 Statistical model for AC DCT based coding model

We have seen that the 8×8 DCT block is ordered into a zigzag sequence, with the DC component at location 0 and the AC components at locations 1 to 63. In the previous paragraph we have presented the process of transforming the DC component into a binary symbol sequence. The AC components are coded with the same decision tree, only that the AC components are assigned to the variable V. In addition each of the 63 AC components has its own set of contexts on which the decisions are applied. The AC component decisions are as follows:

1. First a End of Block (EOB) decision is coded, that is if all the AC coefficients (1 to 63) are zero. If this is the case, the coding process for this block has finished.

2. Next the first AC coefficient $ZZ(K), K = 1$, is assigned to V and the same coding process as for the DC component if followed. In the case that $ZZ(K) = 0$, K is incremented and no EOB decision has to be coded, since there must be another non-zero AC coefficient, because if there isn't, an EOB decision would have been coded in the previous step. This means that a EOB decision is only coded after a non-zero AC coefficient.

3. If K reaches 63 and the coding process is not terminated through an EOB decision, then after coding the last AC coefficient, the coding process for this 8×8 DCT block has finished.

AC-BINS	
0	AC_SIGN
1 to 63	EOB
64 to 126	$C = 0$
127 to 189	$SZ < 1, SZ < 2$
190 to 203	$X_2, X_3 \ldots, X_{15}, \quad K \leq Kx$
204 to 217	$X_2, X_3 \ldots, X_{15}, \quad K > Kx$
218 to 231	$B_2, B_3 \ldots, B_{15}, \quad K \leq Kx$
232 to 245	$B_2, B_3 \ldots, B_{15}, \quad K > Kx$

Table 2.3: Binary decisions (bins) assigned for AC DCT coefficients

The decision tree used for coding the DC and the AC components are exactly the same, although the contexts which they use are slightly different. The sign context is common for all AC coefficients, because it is random and evenly divided between negative and positive values. Therefore, it is coded with a fixed probability of 0.5, which means that the contents of the storage location of the sign context remains the same during the coding procedure. The $EOB, S0, SP/SN, X1$ contexts are conditioned only on the position of the AC coefficient in the 63 zigzag sequence, that is on the number K. Also, the contexts $SP/SN, X1$ are identical, which means, that they use the same storage location. It is quite reasonable to code the SP and SN decisions with the same bin, since there are no statistical differences between positive and negative values. The decision to code $X1$ also with the same bin is based on experiments showing that these three decisions are very similar [1]. The rest of the binary decisions $X1, \ldots, Xn, M1, \ldots, Mn$ are conditioned on the value of the index K into two categories. Values of K less than a specified value Kx, use one set of contexts and values above Kx use another. Kx is chosen so that the DCT coefficients are split into *low-frequency* and *high-frequency* components. The default value for Kx is 5. The assignment of storage locations to contexts and the conditional decisions are shown in Table 2.3.

The total number of AC bins is 246 and together with the 49 DC bins, there is a need for 295 storage locations. Each one of the storage locations contains one 7-bit index into the probability estimation state machine, plus one bit which signifies the assignment of 0 or 1 to the MPS. The 7-bit index and the MPS bit are dynamically changed after each coding process in the encoder and decoder as specified in the coding procedures presented in the previous sections. The assignment of the indexes for each of the 295 contexts, specified in Tables 2.2, 2.3, is not suited for hardware implementation, because arbitrarily additions to the index have to be made in order to code a DCT coefficient. For a DC coefficient for example, there may be necessary to use the contexts 0, 1, 20, 21, 22, 33, 34, 35. The jumps from one context to the other may require an addition. A more simplified assignment in which only logical bit operations have to be performed on the indexes is preferred. We have chosen such an assignment, in which for the sake of simplicity, some storage locations and thereby silicon area are sacrificed. The total number of memory locations that we use is 320, of which 25 remain unused. The detailed bin-assignment to memory indexes is presented in the chapter describing the implementation.

Chapter 3

Design Overview and Functional Description

In this chapter we will present and describe the functionality of the various units that compose the entropy coder. In addition, the pipeline stages of the hardware implementation will be presented and the various data and structural hazards will be analyzed. Initially, we will present the interface of the chip to the external circuit.

3.1 Design Overview

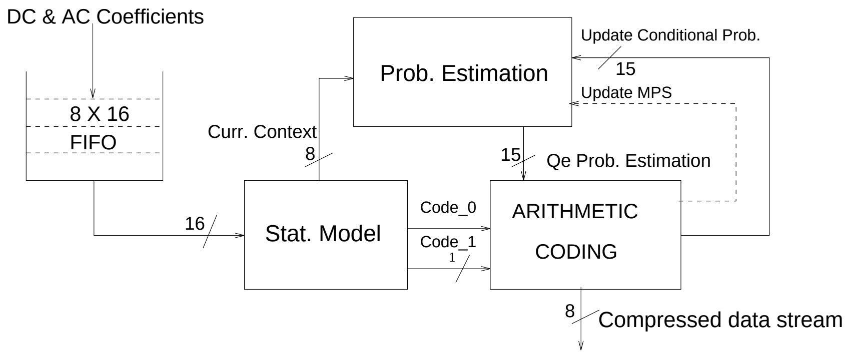


Fig. 3.1: General encoder data flow diagram

A general data flow diagram of the entropy coder is shown in Fig. 3.1 for the encoding mode and in Fig. 3.2 for the decoding mode. In the encoding phase, 16-bit (or 11-bit internally sign-extended to 16-bit) DC and AC quantized DCT coefficients are received from the quantization module through a 16-bit bidirectional data bus and are stored temporally in a small FIFO. The order of the incoming DCT coefficients is assumed to be the zigzag coding order presented in Fig. 1.4. An internal counter is used to keep track which of the 64 DCT coefficients is currently received. Each one of the DCT coefficients is coded by generating a decision tree —yes (1) or no (0) answers to a number of decisions, the knowledge of which is sufficient to regenerate the DCT coefficient— and supplying the `Code_0(S)` and `Code_1(S)` decisions to the arithmetic coding unit and the respective context `CX(S)` to the probability estimation unit. The latter, uses the context to look up, in the respective storage area, the pointer into the probability estimation state machine and the associated assignment of the symbols 0 or 1 to the MPS. Following that, the state index is used to access the values assigned to each state and the 15-bit Qe probability value is sent to the arithmetic coding unit. The arithmetic coding unit is now able to complete the coding of the binary decision, performing the operations on the C and A register values with

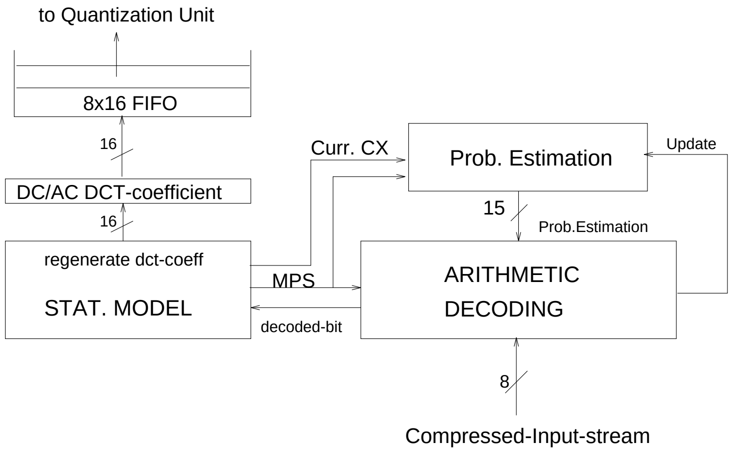


Fig. 3.2: General decoder data flow diagram

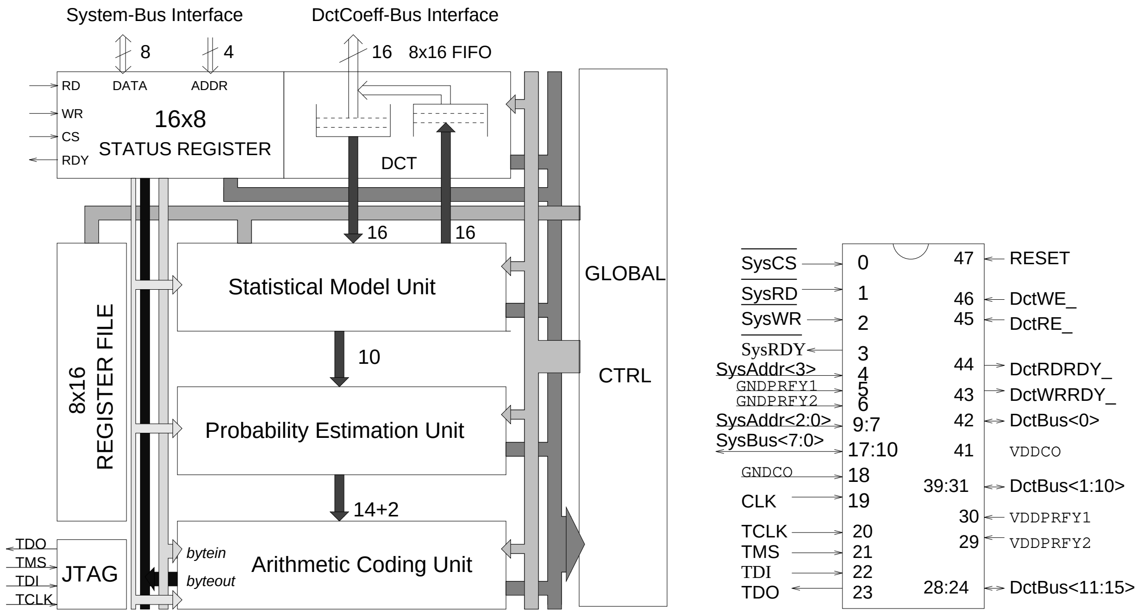
the supplied Qe value, according to the coding procedures described in chapter 2. Since the binary decision generation, the probability estimation and the arithmetic coding operations are performed in different units, they can be performed pipelined in order to increase the throughput. The detailed implementation of each of these units will be presented in the next chapter.

The respective data flow in the decode mode of operation is shown if Fig. 3.2. Coded data, one byte at a time, are fed into the arithmetic coding unit. The statistical model unit is again responsible for generating the appropriate binary decisions that will be decoded and the corresponding contexts. Then, the contexts are given to the probability estimation unit, which supplies the necessary Qe values to the arithmetic coding unit. The latter, after a bit is decoded —the answer to the binary decision— is ready to provide it to the statistical model unit.

From the above simplified description an inefficiency of the coding process during decoding has to be taken into account. During encoding, the binary decisions issued by the statistical model unit, are fairly independent of each other, that is the context generated for each binary decision does not depend on the output of the arithmetic coding unit. However during decoding, many binary decisions and hence the associated contexts, are based on the outcome of a previous decoded binary decision from the arithmetic coding unit. Detailed specification of the data dependency will be presented in the following sections, but an example of such a case is when the binary decision decoded specifies one of the bits of the DCT coefficient. Clearly such a binary decision cannot proceed until the previous binary decision, which specifies that the DCT coefficient is non-zero, is decoded —in case that the DCT coefficient is zero, the next binary decision that have to be decoded differs from the binary decision, when the DCT coefficient is non-zero and therefore the statistical model unit has to know the outcome of the previous decoding operation, in order to start decoding the correct binary decision. Although the decoding process for a later binary decision can be started and proceeded until the outcome of the previous decision is known, this is not always possible, because the decoding process may update status registers, which are used in the following decoding operations.

3.2 Functional Overview

A general block diagram of our implementation is shown is Fig. 3.3(a). It mainly consists of three units: the statistical, the probability estimation and the arithmetic coding, which also correspond to three distinct pipeline stages. Furthermore, the probability estimation unit occupies one additional stage, for a total of 4 stages for the three units. In the first stage of the probability estimation unit, the index for a specific context is looked up and in the second stage the corresponding Qe value is retrieved from the probability estimation state machine. Two more



(a) Functional Block Diagram (b) 48-PIN DIL (Dual In Line)

Fig. 3.3: Functional Block Diagram and package

units are responsible for the interface to the outside world, for the coded data and the DCT coefficients. The first interfaces to a system bus through an 8-bit bidirectional data-bus and a 4-bit address bus. The second interfaces through a small 8x16 FIFO to a 16-bit bidirectional data bus. The 4-bit address, in the system bus, specifies one out of 16 possible external addressable registers, which are used to setup, start, stop or stall the operation of the chip. Additionally, another 16 registers are used in the coding process across 8x8 coding block boundaries, besides the local registers that are used independently by the other units. The probability estimation and the arithmetic coding unit are self-controlled since they include locally a small control unit. A global control, which implements the coding FSM, is responsible to supply the necessary control signals to all units and to the respective local control circuits.

3.3 I/O Interface

The chip interfaces with a standard 8-bit data, 4-bit address, non-multiplexed, asynchronous microprocessor system-bus, through which control and coded data are transmitted. The 16-bit DCT-coefficient interface, is in reality a small FIFO, through which decoded DCT coefficients are send to the external DCT unit and from which DCT coefficients that have to be coded, are read into the internal circuit. The FIFO contains two ports, which are configured according to the operation mode (encoding/decoding). Each port can be read or written independently. Specific synchronization of asynchronous signals both in the system bus interface and in the DCT bus interface is performed in order to avoid any metastability in the internal edge-triggered registers. The I/O pads are ordered around the die, in a 48-DIL package as shown in Fig. 3.3(b) and their respective functions are briefly described below. Overlined signals are assumed to be active low.

$\overline{\text{SysCS}}$, $\overline{\text{SysRD}}$, $\overline{\text{SysWR}}$, $\overline{\text{SysRDY}}$ compose the control part of the microprocessor interface. The address latched during the falling edge of SysWR or SysRD is used to address one of the 16 internal registers. When a read operation is performed, the value of the respective register is placed onto the data bus after a specific amount of time after the falling edge of the SysRD signal, and remains on the bus, until the SysRD signal is deasserted.

SysRDY goes low, as soon as SysRD or SysWR is asserted and remains low until the internal circuit synchronizes the SysRD or SysWR signal and it is ready to accept a new read or write operation.

SysBus[7:0] This is the system data bus, through which values are written into the internal registers or during a read operation the values of the respective registers are placed onto the bus.

CLK This is the single phase clock input. Frequencies up to approximately 20 MHz are permitted, with a 40%-60% duty cycle.

TRST, TCLK, TMS, TDI, TDO These five pins connect directly to the Joint Test Action Group (JTAG) controller and their function is described in [17–19].

$\overline{\textbf{reset}}$ This is the reset input. For proper initialization of the chip, the reset signal should stay low for a minimum of 12 cycles. Our chip synchronizes internally the reset input and remains in an initialization state until this signal is deasserted.

$\overline{\textbf{DctWE}}$ This signal belongs to the DCT-coefficient interface side. On the positive edge of $\overline{\text{DctWE}}$ the Data on the DctBus is written into the internal FIFO.

$\overline{\textbf{DctRE}}$ This signal belongs also to the DCT-coefficient interface side. On the positive edge of $\overline{\text{DctRE}}$ a DCT-coefficient is read from the internal FIFO and is placed to the DCT-bus.

$\overline{\textbf{DctWRRDY}}$ This signal is in reality the internal fifo-full signal, which signifies that no more data should be written into the FIFO.

$\overline{\textbf{DctRDRDY}}$ This signal is the internal fifo-empty signal, which signifies that no more DCT-coefficients are available to be read.

DctBus[15:0] This is the 16-bit bi-directional DCT-coefficient bus.

GNDCO, VDDCO, GNDPRFY1, GNDPRFY2, VDDPRFY1, VDDPRFY2 These six primary pins are used for ground and power. Two pins of each type are used to supply the IO-ring and one pin for the core of the chip.

3.3.1 System Bus Interface

The registers that are addressable through the system bus are shown in Fig. 3.4. These registers should be set up appropriately before the starting of a coding process and also through these registers the coding process can be started, stopped or freezed.
The following flags are in status register 0.

E This bit set (1), specifies that encoding should be performed. The actual encoding starts when bit **C** is also set.

D This flag specifies that decoding should be performed. Only one of **D** or **E** should be activated at the same time when also **C** is on.

C Signals that coding should start when also one of **D** or **E** is on. This is very convenient at start-up, when all the registers have to be set-up. When the initialization has been completed, setting this bit to 1 starts the coding process. This bit is automatically reset after coding has finished. The size of the input data (coding) or output data (decoding) is specified by

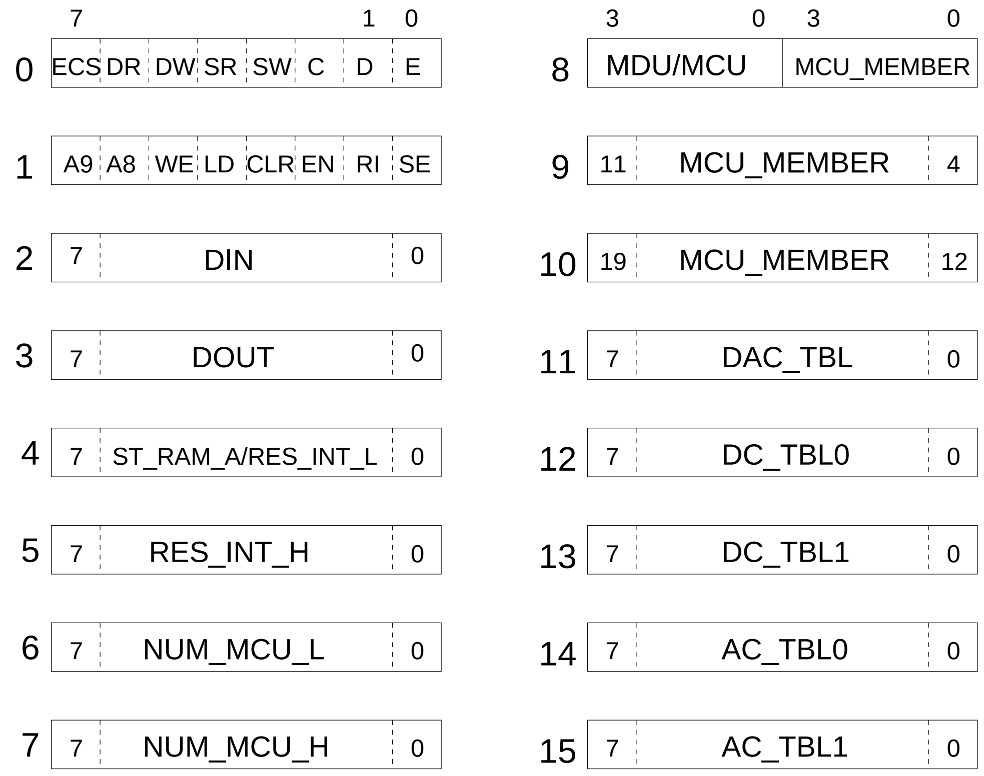


Fig. 3.4: External addressable internal registers

other registers. Also setting this bit to zero, during the coding process, has the effect to freeze the entire chip. The control unit remains at the same state, until this bit is set again. At that time the coding process is resumed.

SR, SW These signals are a mirror of the internal synchronized $\overline{\text{SysRD}}$ and $\overline{\text{SysWR}}$ signals.

DR, DW These signals are also the internal synchronized $\overline{\text{DctRD}}$ and $\overline{\text{DctWR}}$ signals. These and the previous signals are provided in the first status register for monitoring purposes. Since the system bus interface registers can be written and read independently from the internal circuit, the values of these registers can be read any time, from the external circuit.

ECS signifies the end of the coding process.

The following flags belong to status register 1.

SE (**S**ign **E**xtension) is used to internally sign-extend 11-bit DCT-coefficients to 16-bit. Our entropy coder can operate both with 11-bit and 16-bit DCT-coefficient precision since they are both converted internally to 16-bit. This bit is useful in the case that no external sign-extension of the 11-bit coefficients is performed.

RI (**R**estart **I**nterval) Although included, this bit is not currently used. It would signify restart intervals, which reset the encoding process at appropriate intervals, specified in the coding process, to help error recovery in transmission of JPEG coded images over noisy channels. Since the use of our coder is basically targeted in computing systems and not communication systems the implementation of such a feature was considered but finally not adopted.

EN This bit is currently not used.

CLR, LD, WE, A8, A9 These signals are used together with ST_RAM_A[7:0] for reading and writing the internal 640×8 static RAM, which contains the contexts for the current coding process. The possibility to read and write independently from the normal operation the RAM is mainly used for fault-testing. Instead of providing internal BIST circuits, this additional feature is included, since the testing of the RAM under normal operation requires a large number of test vectors. The **CLR** signal specifies if the internal context RAM should be first initialized to zero. This is recommended each time the coding process begins. Therefore, this bit should be set. The **LD** signal specifies that a specific value which was previous written into the register **DIN** should be written or read according to the flag **WE** in address {A9, A8, ST_RAM_A[7:0]}. This operation can be used to write and read the RAM during fault-tests.

Two data registers are used to send and receive coded data during encoding and decoding.

DIN[7:0] register supplies coded data during decoding. This register is written through the system bus. During the tests for faults of the internal context RAM, the value of the register is written into a specified location in the context RAM.

DOUT[7:0] register receives from the internal circuit the coded data during encoding and also supplies the word, which was read from the context RAM, during fault-testing.

The following items specify the restart interval length, which is currently not implemented but the registers are reserved for future implementations. The size of the decoded data, during both mode of operation, is provided as a combination of the number of MDU over MCU and the number of MCU.

ST_RAM_A, RES_INT_L contains address bits [7:0] of the 10-bit address, which is used to index the internal context RAM during fault-testing.

RES_INT_H is the high 8-bit part and along with the previous register, define a 16-bit restart-interval for the coding parameters (the contents of the context internal RAM).

NUM_MCU_L, NUM_MCU_H this 16-bit value defines the length of the decoded data as an integral number of MCU. An internal counter is used to keep track which MCU is coded.

MDU/MCU this four bit value specifies the number of MDU in each MCU. Since the number of MDU in one MCU is limited to ten, according to the JPEG specifications, four bits are used. After each MDU is coded, a respective internal counter is decremented. When the counter reaches zero, the respective MCU counter is decremented until the entire image is coded or decoded.

The following registers define the context area to be used for each component in one MCU and also the LU and Kx values to be used for the conditional coding of the DC and AC coefficients. The corresponding structure is shown in Fig. 3.5.

MCU_MEMBER[19:0] as was mentioned above, the number of MDU in each MCU is limited to ten. For each such MDU this 20-bit register contains a 2 bit index, which specifies the image component to which the MDU belongs. A total of 4 image components can be specified. As we have seen in sec. 2.1, JPEG uses mainly the YCbCr color space, so the three entries of the four are commonly in use.

DAC_TBL[7:0] this 8-bit register is split in two parts. Each one contains four 1-bit indexes for each of the four components in one scan. Each index points to one out of two possible context areas, which are implemented in our chip using two context RAMs. Since different components in our color model (YCbCr) are permitted to use different context areas,

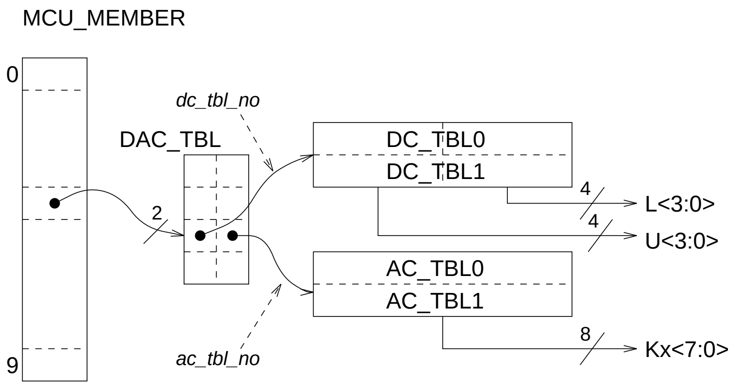


Fig. 3.5: L, U and Kx parameter selection circuit for the two internal context areas

the JPEG specifications define a maximum of 16 such context areas to be used. Practically, only two such areas are in use, one for the Y component and one for the CbCr components and therefore, we have decided to include in our implementation only two such context areas. It is also possible to use only one area, but in this case the components have to be coded one at a time and no interleaving is permitted. In the case of interleaved coding the values of the indexes, which are used by one component, have to be saved externally and later written again into the appropriate context RAM. We have avoided this by including two context areas.

DC_TBL0[7:0], DC_TBL1[7:0] these two 8-bit registers contain the LU values for conditional coding of the DC components for each of the two context areas.

AC_TBL0[7:0], AC_TBL1[7:0] these two registers contain the two Kx values, on which the magnitude bits binary decisions of the AC coefficients are conditioned for each context.

3.3.2 DCT Bus Interface

The DCT-bus Unit interfaces with the rest of the chip through the signals shown in Fig. 3.6. As was pointed out previously, the interface is composed mainly from a small 8$\times$16 dual port static RAM. Two bidirectional 16-bit buses interface to the two ports of the RAM. Each port can be read or written independently More details on the operation of the interface will be presented in the implementation chapter. The signals *dct_fifo_full* and *dct_fifo_empty* are send directly to the two output pins $\overline{DctWRRDY}$ and $\overline{DctRDRDY}$ and are used for flow control of the external circuit, which is responsible to supply the DCT-coefficients. Since these signals are asynchronous to the internal clock of our chip, two special signals (*dct_fifo_full_synch, dct_fifo_empty_synch*) are generated which are synchronous to the internal circuit and can be used safely by the global

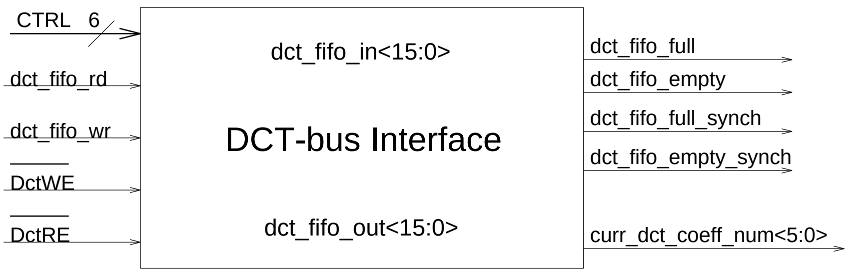


Fig. 3.6: DCT-bus interface unit

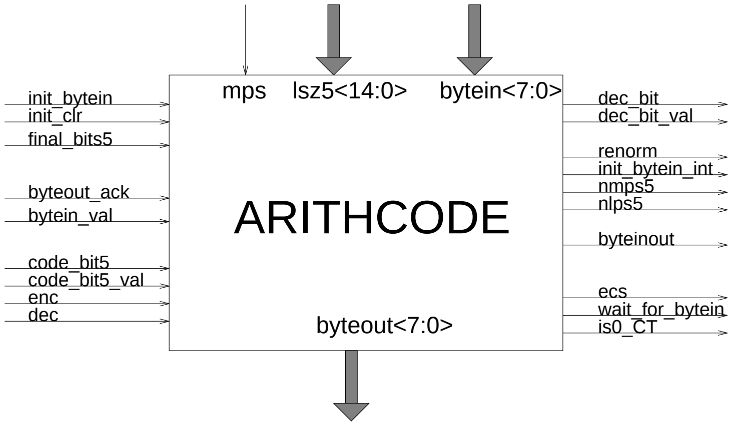


Fig. 3.7: Arithmetic Coding Unit I/O signals

control unit. Also, two additional synchronous signals (*dct_fifo_rd*, *dct_fifo_wr*) read or write the DCT-coefficients from and to the statistical model unit, are provided to the control unit. The other two signals (*DctWE*, *DctRE*), are provided by the corresponding external circuit and specify if the FIFO is to be read or to be written through the external DCT-bus.

3.4 Arithmetic Coding Unit

The interface of the Arithmetic Coding Unit to the rest of the chip is shown in Fig. 3.7. It contains the 32-bit C and A registers necessary for the arithmetic coding process. It also includes a 4-bit CT counter, a 24-bit counter for the stacked 0xFF bytes (described in sec. 2.4) and the 8-bit intermediate output buffer on which the possible carry overs are resolved before the byte is actually send to the $DOUT$ register. Three adder units —two 16-bit and one 28-bit— produce the necessary operations for the arithmetic coding procedures, as was outlined in sec. 2.4 and sec. 2.5. The control signals for the coding process are produced internally by a small control unit which operates according to the internal values of the registers and the external supplied signals from the control and the probability estimation unit. The local control unit also provides feedback to both of these units (global control and probability estimation), on the operations performed internally.

During encoding, the Arithmetic Coding Unit is able to receive on each cycle the binary decision bit (*code_bit5*) along with its corresponding Qe probability (*lsz5[14:0]*) and the MPS bit (*mps*). Following that, it performs the necessary operations on the internal C and A registers and outputs at the end of the cycle two signals, which specify the path that has been followed. The possible paths are the MPS path (*nmps5*) or the LPS path (*nlps5*). In case that the probability estimation unit is not ready to supply any binary decision to be coded, this is indicated by the validity signal (*code_bit5_val*). If this signal is inactive (0), no operation in the unit is performed and the values *lsz5[14:0]* and *mps* are ignored. For each binary decision coded it may be necessary to perform additional renormalizations cycles, in which case the *renorm* signal is asserted.

The *renorm* signal is also send to the probability estimation unit, which stalls until it is deasserted. In case that the probability estimation unit is sending a new binary decision that has to be coded, the values on the *mps*, *lsz[14:0]*, *code_bit5* and *code_bit5_val* signals remain the same until the renormalization cycles are finished and the corresponding *renorm* signal is deasserted. In reality the *renorm* signal does not correspond to a renormalization cycle according to the coding procedures, since a renormalization is already incorporated in the coding cycle. Only when additional cycles are necessary, that is when A falls below $0.75/2$ and the first renormalization still keeps A outside the correct interval $0.75 \leq A < 1.5$, then the *renorm* signal is asserted and the next binary decisions have to be stalled. Finally, when a coded byte is ready to be transmitted to the $DOUT$ register, the signal *byteinout* is asserted. This signal is also the load-enable signal for the $DOUT$ register and since there is no guarantee that a byte will be read from the outside circuit until the next byte from the arithmetic coding unit is ready to be transmitted, a handshake signal (*byteout_ack*) is provided. This signal goes low immediately after the *byteinout* signal is asserted and is set when the external circuit has read the byte from the $DOUT$ register. The internal control unit guarantees that the next *byteinout* signal will remain high during the period that the *byteout_ack* signal for the previous byte remains low. Therefore, another reason that may cause the pipeline to stall is when the coded bytes are not absorbed fast enough from the external circuit.

During decoding, the arithmetic coding unit receives the Qe value and the *mps* for the current binary decision, but the *code_bit5* value is ignored. Also, in this case the *code_bit5_val* bit indicates if the previous values are valid. The *renorm, nmps5, nlps5* signals have the same function as during encoding but this time, a new signal is used to provide the binary decision that has been decoded (*dec_bit*). The signal *dec_bit_val*, specifies if the signal *dec_bit* is valid. During the cycles in which no bits are decoded, this signal is inactive. Also, at the same time, when a new coded byte is required by the arithmetic coding unit, the signal *byteinout* is asserted. When the signal *bytein_val* is active, a new byte of coded data is already in the DIN register and therefore, if *bytein_val* is high when the *byteinout* signal was asserted, no cycles have to be stalled. In the other case, in which the external circuit was not fast enough to provide another coded byte, the signal *wait_for_bytein* is asserted by the arithmetic coding unit and provided to the probability estimation unit, to signify that the first unit is not ready to continue decoding until a new byte of data is acquired.

From the above presentation we can infer that the primary source which prevents a one-cycle coding of a binary decision, during the operation of the arithmetic coding unit, is the renormalization process, which requires ≥ 2 renormalizations. Since renormalization is performed just by shifting left the A and C registers, the achievement of a one cycle coding and renormalization would require a small (3-bit) priority encoder in order to calculate the necessary number of such shifts, and the provision of additional paths to the input multiplexers of the A and C registers with their corresponding values shifted left by one and two. We have decided that it is not necessary to include the above circuits, since the probability of such renormalizations is low and the inclusion of such circuits would increase the clock cycle time of the coding process. Therefore, in each renormalization cycle the A and C registers are shifted left by one position and then the A register is checked against the correct interval. Also, the pipeline stalls of the arithmetic coding unit which depend on the input and the output data should normally not occur. The minimum number of cycles required until the arithmetic coding unit needs to output a byte is eight cycles, when in each cycle only one renormalization is performed. In most cases, the number of cycles needed will be larger. Therefore, if the external circuit is able to read the bytes during encoding and is able to provide coded bytes to the DIN and $DOUT$ registers respectively every 8 cycles, such clock cycle delays can be avoided.

3.5 Probability Estimation Unit

The probability estimation unit lies between the statistical model unit and the arithmetic coding unit. It implements the probability estimation machine and the two context areas for the binary decisions provided by the statistical model unit. The state machine is implemented as shown in Fig. 3.9 and it occupies two stages in the overall pipeline. The corresponding I/O signals of the Unit are shown in Fig. 3.8. As was mentioned in sec. 2.7 the probability estimation state machine contains 113 states. Each state contains a 15-bit probability estimation value (Qe) and two 7-bit indexes for the state transitions that may be followed when a MPS or LPS is coded. Also, a one bit value in each state signifies if a conditional exchange has to be performed in that

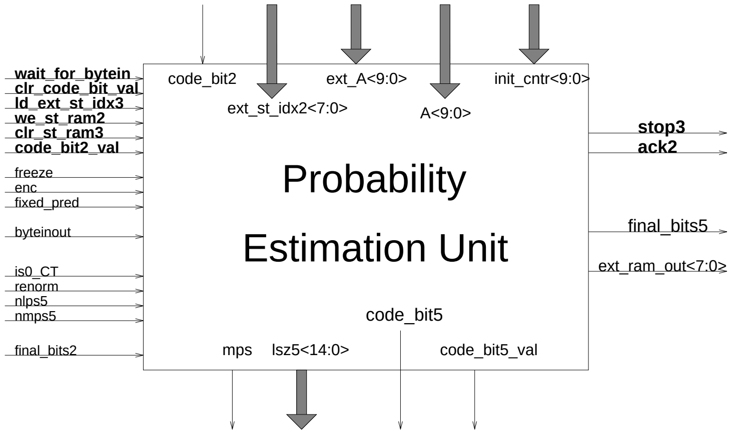


Fig. 3.8: Probability Estimation Unit I/O signals

particular state. Therefore, the total number of bits assigned to each state is 30. Such a FSM can be implemented as a ROM with an 8-bit index and a 30-bit word length. In the next chapter we will present an alternative implementation using PLAs that we used in our design.

Each of the binary decisions generated by the statistical model unit or decoded by the arithmetic coding unit is assigned a context —an index into a static RAM. The corresponding storage location contains an index into the probability estimation state machine and a bit which is used to assign the symbols 0 or 1 to the MPS symbol. This storage area, which was introduced in sec. 2.8, is implemented as a static RAM with 295 words of 8-bits. Since two such areas are in use, the total size of the RAM is 2×295. The natural way to implement the probability estimation unit is to use two pipeline stages. In the first stage, the index into the state machine is retrieved from the context and in the second stage the Qe probability estimation value is read from the state machine and send to the arithmetic coding unit. Since in many cases the index to the probability estimation state machine for a specific context has to be updated according to the coding procedures and these indexes are stored using a one port static RAM, in these cases which accounts for approximately 60% of the bits to be coded, no new read access for the next binary decision can be initiated.

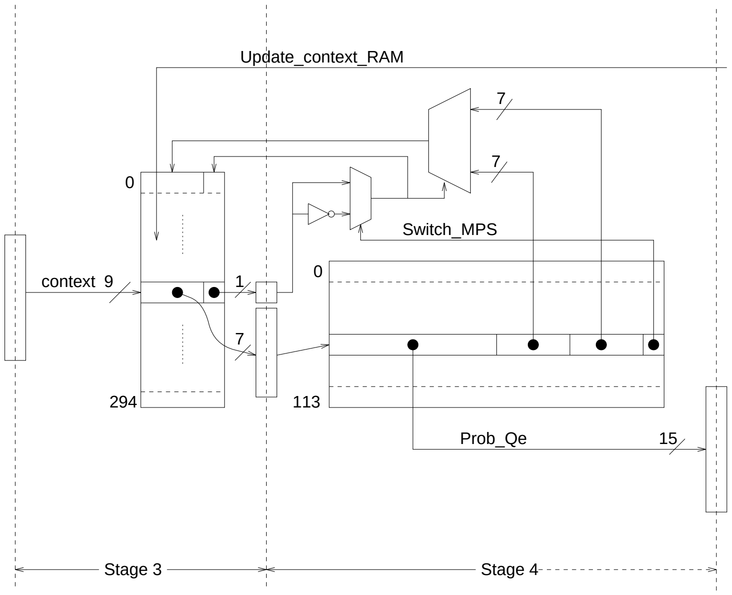


Fig. 3.9: L, U and Kx parameter value generation

Also, in cases in which the pipeline is stalled due the arithmetic coding unit, stages 3 and 4 remain the same. Naturally when the context storage location, which is updated, is identical with the context for the next binary decision, no additional read access for the following coding procedure is required, but the value just written into the context RAM is used for the next binary decision. Therefore, no cycles are lost in the pipeline. These dependencies will be analyzed in the following sections in which the the pipeline data hazards are described in more detail.

The I/O signals of the probability estimation unit are shown in Fig. 3.8. As in the arithmetic coding unit, this unit also includes a small control circuit, which is responsible for the internal datapath and provides some kind of independence to its operation. It receives the binary decision to be coded through the *code_bit2* signal and the corresponding context through the A[9:0] bus and supplies the Qe probability value and the *mps* to the arithmetic coding unit. It also receives the *renorm, nlps5, nmps5, byteinout, wait_for_bytein* and *is0_CT* signals from the arithmetic coding unit which are used to decide if the pipeline has to be stalled. Also, two additional signals (*stop3, ack2*) are provided to the statistical model unit and are used to signify when the pipeline has to be stalled. The same operations are performed during encoding as well as during decoding. The rest of the signals are used only in test mode in which the internal static RAM is tested as was described in sec. 3.3.1. In these cases the value on the bus *ext_st_idx2[7:0]* is written into the location *ext_A[9:0]* if the signal *we_st_ram2* is high or the value in location *ext_A[9:0]* is placed on the bus *ext_ram_out[7:0]* if *we_st_ram2* is inactive. The index *init_cntr[9:0]* and the signal *clr_st_ram3* are used to initialize the indexes of the static RAM before the coding process begins. Also, the *clr_code_bit_val* signal is generated from the global control unit when a decoding operation is started in the probability estimation unit but after one or two cycles this operation has to be canceled. In these cases, the valid flags in stages 3 and 4 are cleared and the internal registers are not updated.

3.6 Statistical Model unit

The statistical model unit, shown in Fig. 3.10, generates the binary decisions, that is the `Code_0` and `Code_1` procedures (see sec. 2.8) and their corresponding contexts for each one of the 64 DCT-coefficients in each MDU and for all MCU units. Therefore, it uses the two shift registers Sz and M. The first contains the absolute value of the DCT-coefficient and the second is used to generate the magnitude category bits and the magnitude bits. The conditional probabilities on which the DC and AC coefficients are based, as indicated in sec. 2.7, are provided through the *arith_dc_U[3:0], arith_dc_L[3:0]* and *arith_ac_Kx[7:0]* parameters. These values, which may differ for each component in a MCU, are provided through the selection circuit shown in

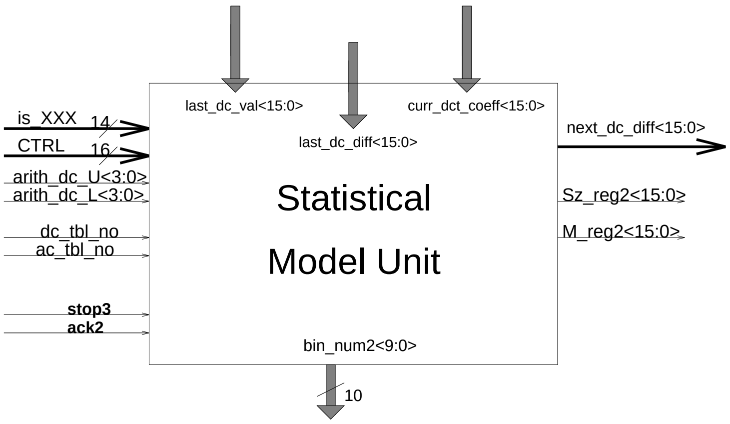


Fig. 3.10: Statistical Model Unit I/O signals

Fig. 3.5. Since two contexts areas are implemented, two one-bit indexes are provided (*dct_tbl_no, ac_tbl_no*) that specify the context area to be used for coding the DC and AC coefficients respectively. The statistical model unit receives during encoding the new DCT-coefficient (*curr_dct_coeff[15:0]*) to be coded, the previous DC-DIFF value and the previous DC DCT coefficient in the case that a DC coefficient will be coded. The next DC-DIFF value is supplied by *next_dc_diff[15:0]* and is has to be written into the register file together with the current (*curr_dct_coeff[15:0]*) DCT coefficient. These values will be used in coding the next DC coefficient of the same component in a MCU. Also, the contents of the Sz and M registers are provided to the control unit. Control signals from the global control unit specify the exact operations that have to be performed by the statistical model unit. The *is_XXX* signals specify the binary decision to be coded in each cycle as described in sec. 2.8.

Since many binary decisions are skipped for several DCT-coefficients, the statistical model unit resembles the instruction decode unit in conventional CPUs. The binary decisions correspond to the instructions executed and since for each DCT-coefficient another set of binary decisions are coded, "branch" instructions are also used. These binary decisions are then "issued" to the following units (probability estimation unit, arithmetic coding unit).

The flow of control is provided by the two signals (*stop3, ack2*) from the probability estimation unit. When these signals are active, the pipeline stages 3 and 4 have to be stalled, for reasons mentioned previously, and the statistical model unit must also suspend its operation for the same number of cycles. These two signals provide the control for the operations in the statistical model unit which either stalls its operation providing the same context to the probability estimation unit or advances to the next binary decision to be coded. The exact operations will be described in more detail in the next chapter.

3.7 Pipeline data and structural hazards

After the previous presentation of the various units involved in the coding process, the overall pipeline during encoding and decoding will be presented. Considering the DCT FIFO in the DCT-interface at stage one, which receives or supplies the DCT-coefficients, the pipeline is composed from 5 stages of which the last 4 are shown in Fig. 3.11.

DCT This stage, which is not shown in Fig. 3.11, is used to supply or receive the DCT-coefficients from the statistical model unit.

BD In this stage the binary decisions, calculated along with the coded bits and the 10-bit context, are supplied to the probability estimation unit during encoding. During decoding, the bits decoded by the arithmetic coding unit are used to reconstruct the DCT-coefficient and to supply the next context to the probability estimation unit.

RAM The index into the probability estimation state machine is looked up from the context, supplied from the statistical model unit, and the corresponding index is supplied to the next stage of the probability estimation unit.

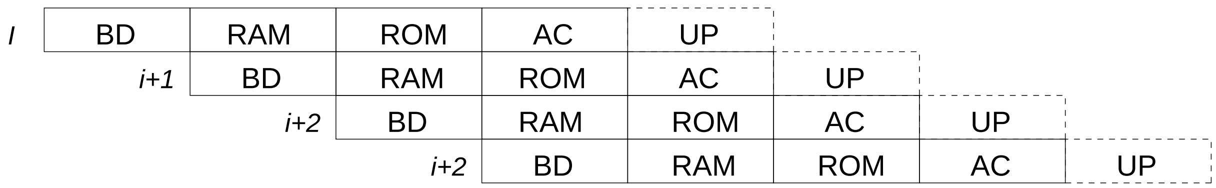


Fig. 3.11: Arithmetic Coding without pipeline-stalls

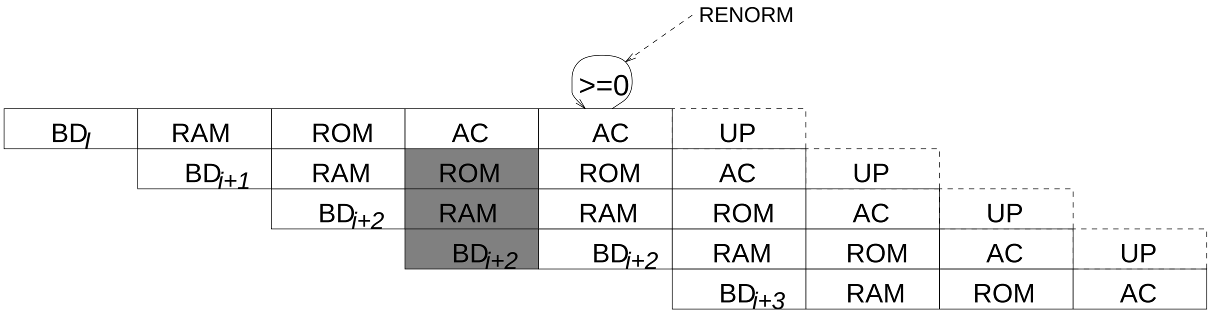


Fig. 3.12: Pipeline stalls due to renormalizations

ROM Finally, the probability estimation Qe for the corresponding context in stage **RAM** is looked up according to Fig. 3.5 and is supplied together with the bit to be coded and the *mps* to the arithmetic coding unit.

AC In this stage, during encoding, the arithmetic coding operations are performed. The probability estimation is supplied from the previous stage.

UP This optional stage is used only in cases in which the index into the probability estimation state machine in stage **RAM** has to be updated. In these cases, the current access in stage **RAM** is delayed but the next stages ROM and AC can proceed. This leaves "bubbles" in the pipeline, which has the result that the arithmetic coding unit will be idle in one of the following cycles and which also decreases the throughput of the coding process.

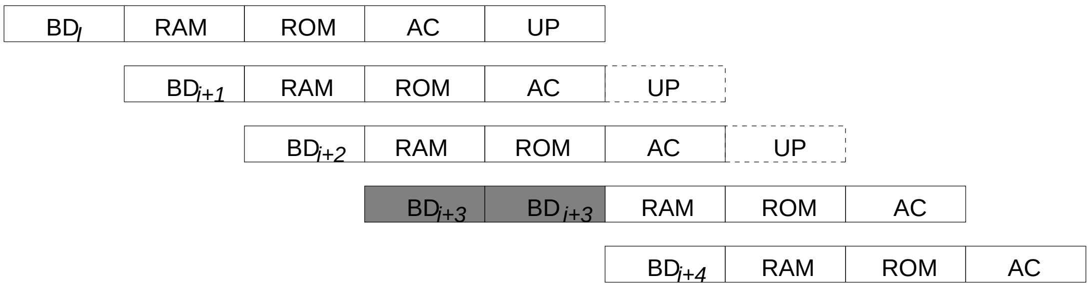
There are three cases, depending on the arithmetic coding, that may stall the pipeline. One due to renormalizations which hold for approximately 20% of the bits to be coded and the other two when the external circuit is relatively slow to absorb the coded data during encoding and can't supply fast enough coded data during decoding. The last two cases should normally not occur in a properly designed external circuit. The first reason is unavoidable and is shown in Fig. 3.12. All the stages before the arithmetic coding stage remain the same until the necessary renormalizations are performed. This occurs during encoding as well as during decoding. Although we have mentioned how this could be avoided (sec. 3.4), it has been decided to tolerate the cycles that are lost due to such renormalizations since they compose a small percent of the total coding cycles and their elimination would have lenghten the cycle time.

3.7.1 Pipeline control-hazards during encoding

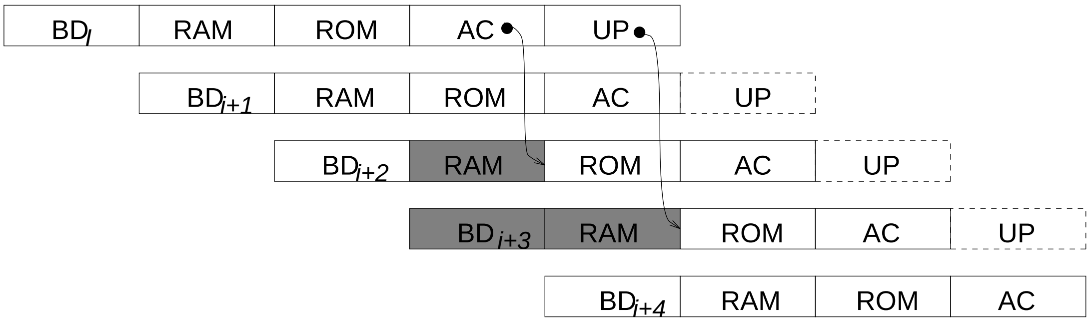
Fig. 3.13 contains the possible combinations of the contexts in stages **BD, RAM, ROM, AC** and the corresponding actions that are performed, in order to forward values from the later stages to the previous one, avoiding to read the same values from the context RAM or the probability estimation PLAs. The four cases are:

Case (a) is the simplest one. All the contexts in the four pipeline stages are different and therefore forwarding is not possible. The binary decision in stage $\mathbf{BD}_{i+3}$ has to be delayed one cycle, in order to update the context of stage **AC**.

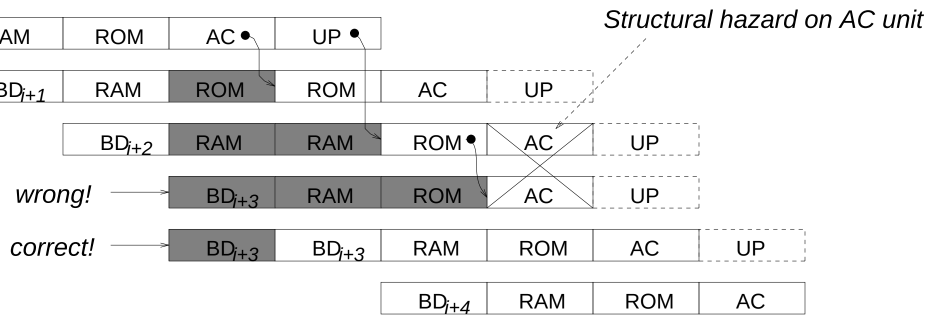
Case (b) implies two different events that may also occur simultaneously. The first arrow from left to right forwards the updated value of the context from stage **AC** to the current context, which is accessed in stage **RAM**. The second arrow forwards the updated context, which is currently written into the static RAM, also to stage **ROM** if the corresponding context for the new binary decision in $\mathbf{BD}_{i+3}$ points to the same storage location.



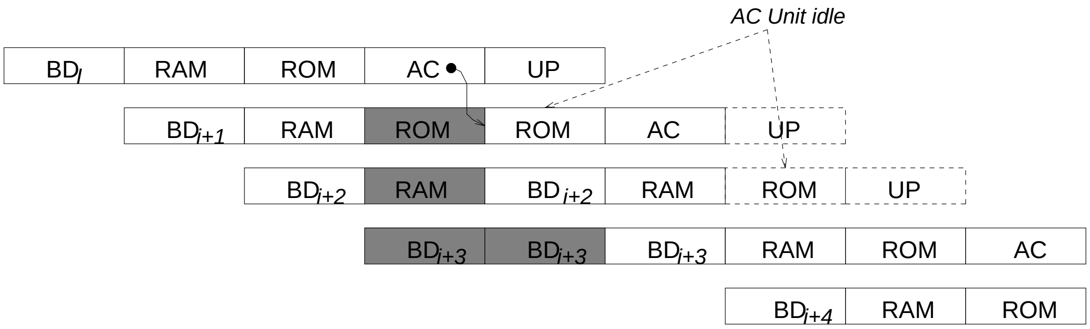
(a) context$_{AC}$ $\neq$ context$_{ROM}$ $\neq$ context$_{RAM}$ $\neq$ context$_{BD}$



(b) context$_{AC}$ $=$ context$_{RAM}$ **or** context$_{AC}$ $=$ context$_{BD}$



(c) context$_{AC}$ $=$ context$_{ROM}$ $=$ context$_{RAM}$ $=$ context$_{BD}$



(d) context$_{AC}$ $=$ context$_{ROM}$ **and** context$_{AC}$ $\neq$ context$_{RAM}$ **and** context$_{AC}$ $=$ context$_{BD}$

Fig. 3.13: Encoding pipeline data hazards

Case (c) In this case the contexts in stages **AC, ROM, RAM** are equal. Direct forwarding from stage **AC** to stage **ROM** is not possible since the index into the probability estimation machine for the corresponding context has changed and a new probability estimation has to be retrieved using the new index. Nevertheless, a one stage forwarding is performed from stage **AC** to stage **ROM**. The binary decision currently in stage **RAM** remains in the same stage and the value written into the static RAM is forwarded to the next stage **ROM**. At first glance, since the context in stage **BD**$_{i+3}$ is the same as the updated value from stage **AC**, the probability estimation is already known before any access in the static RAM or the probability estimation PLAs has been performed. Nevertheless forwarding the binary decision from stage **BD**$_{i+3}$ directly to stage **AC** is useless, since the **AC** unit can only handle one coding operation in each cycle. Therefore, the binary decision **BD**$_{i+3}$ is delayed one cycle and the static RAM is accessed as usual.

Case (d) is similar to case (c), with the discrepancy that the context in stage **RAM** is different than the context in stage **AC** and the latter is equal to the contexts in stages **ROM** and **BD**$_{i+3}$. Again, a cycle is lost because the new probability estimation has to be retrieved; also, an additional cycle is lost because the context in stage **BD**$_{i+3}$ is delayed one cycle with respect to **BD**$_{i+2}$ which interferes with the possible forwarding from **BD**$_{i+1}$. The last context is used to access the static RAM a second time after the update from stage **AC** has completed. In order to use the result of the first access, that result has to be saved and following that, it could be used instead of accessing the static RAM a second time. Due to complications in the control unit this last optimization was not performed.

3.7.2 Pipeline control-hazards during decoding

The decoding pipeline contains exactly the same data dependencies like the encoding pipeline. Unfortunately, in order to decode a binary decision, the previous binary decision has to be available in order to select the appropriate path when multiple decoding paths are possible. This would actually eliminate the pipeline and the operations would be performed sequentially with a clock cycle decoding time of 4 for each binary decision. In order to exploit the available pipeline, the following paths are predicted and if they are predicted correctly they are permitted to update the respective state registers. When wrong predictions occur, the already issued binary decisions are canceled and the state in the statistical model unit is reversed to the last correct state. Shadow registers are available for the M register, Sz register and the magnitude bit counter, which are used to back up the current values in the corresponding registers. The respective pipeline is shown in Fig. 3.14. Binary decisions from stages **BD**$_{i+1}$, **BD**$_{i+2}$, **BD**$_{i+3}$ are permitted to advance to stages **RAM, ROM** and if the binary decision, which is decoded in stage **AC**, is different than predicted, the corresponding valid bits in stages **RAM, ROM** are cleared and the decoding process is restarted from stage **BD**$_{i+1}$. Since prediction is only necessary during decoding, the decoding of the magnitude category requires a sequence of binary decisions to decide if it is larger than 1, 2, 4, 8, ..., 2^{16}. Such adjoined binary decisions will be predicted as true and only the last binary decision will be false. Therefore the startup time of the

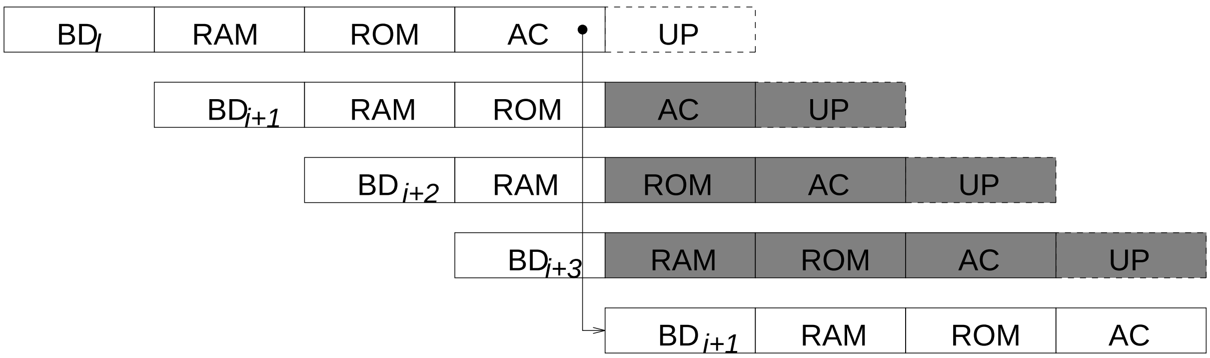


Fig. 3.14: Arithmetic Decoding with pipeline stalls and invalidations

pipeline is payed only once and following that, the binary decisions are decoded with the maximum throughput in the pipeline. Also, during decoding of a series of zero DCT coefficients, the high frequency components have a high probability of being zero, and the prediction is enabled. During decoding the actual magnitude bits, the prediction can also be used since the magnitude category is already decoded. In these three cases the decoding pipeline has only the data dependencies which are also present during encoding and therefore, a higher decompression rate can be achieved. Due to some minor complications in the control unit, the last optimization was not performed in the circuit shipped for fabrication due to time constraints in the design duration.

Chapter 4

Implementation

In this chapter we will present the detailed implementations, of the units whose operations were described in the previous chapter. Although the presentation is detailed down to the gate level on some circuits, critical to the operation of the chip, other circuits will be presented in a more abstract form. The detailed schematics and circuit specifications can be found in appendix C. Each Unit will be described independently and the overall operation, controlled by the global control unit, will be presented in the last sections.

4.1 Foundry and Technology Characteristics

As was mentioned in the introductory chapter, the technology that we used to design our chip was the $1.0\,\mu$m[1], double metal, single poly from European Silicon Structures (ES2). The CAD software available to us for the specific process included SOLO1400 [20] —an in house software package developed by ES2 for standard cell designs— and the SOLO2030 [21] design kit for the Cadence Edge design framework, also supplied by ES2. The Cadence Edge design framework is provided by EUROCHIP and is independent from foundry and process. The SOLO2030 design-kit is targeted mainly for standard cell design, although full-custom design, switch-level synthesizing and gate-level synthesizing are also possible within Cadence-Edge, but they are not directly supported by the SOLO2030 design kit. This directed the use of only standard cells in our design despite the fact that in many cases full-custom design of parts of the datapath would result in a significant faster implementation. On the other side, full-custom designs take longer to complete since analog simulations (spice) have to be performed in order to verify their correct operations and appropriate Verilog models have to be written in order to interface them with the rest of the design, which use the standard cell library.

After considering the advantages and disadvantages of each design methodology and taking into account the design options that we had, the SOLO2030 design-kit was used to design the chip. In this design-kit all the components of the library are used with their symbols and the design is carried out through schematics generated by the designer. The library can be categorized into three groups: a standard cell part, a soft-cell library implemented using the standard cells, and a number of generators for regular structures like Static RAMs. I/O cells with various driving strengths are also supplied and optional clock generators can be included that interface directly to external oscillators. Since our design uses only digital cells, no analog components of the library were used. The standard cells on which all the other components are based, are described and commented in App. A. Briefly, they include basic cells, like *nand, nor, and, or, multiplexers*, *tri-state drivers*, *buffers* and *inverters*. Many cells are also supplied with more than

[1] initially we were using the 1.5μm process, but during the design phase the $1.0\,\mu$m process become available and as the two processes use compatible standard cell libraries, no special conversion was necessary in order to use the more advanced process.

one inputs, like the *nand3, nand4* gates and the *inverters* and *buffers* include three variants with different driving strengths. The storage cell library is composed from fall-through latches and edge triggered flip-flops with all possible combinations of asynchronous set and clear signals. The *registers, multiplexers* and *tri-state drivers* are also supplied in cells of four. The soft-cell library includes flip-flops like *JK* and *T* which are missing from the standard cell library and also primitive gates like the *nand, nor*, etc, with various driving strength inverters at their outputs. Since these are soft cells, they should not be confused with the standard cells. The former are implemented using standard cells and therefore routing area must also be taken into account.

More complex soft-cells include *counters, registers* and *bus-drivers* and they are ported from TTL and CMOS discrete component schematics to the ES2 standard cell library. The generators are used for various sizes of RAMs, ROMs, PLAs and Multipliers. They create, according to user defined specifications, an abstract layout and the respective schematic symbol which can be used together with all the other components of the library. A datapath compiler for the $1.0\,\mu$m process is also available, which was not included in $1.2\,\mu$m and $1.5\,\mu$m processes. We used mostly the standard cell components and the generators for the static RAM which contains the context storage and for the PLAs of the probability estimation state machine. We have decided to use only synchronous signals in our design, even in the initialization phase. This means that we haven't used storage cells with asynchronous set or clear signals and that the initialization values are explicitly written into the flip-flops by the control unit. Furthermore, the external supplied reset signal is synchronized and supplied to the control unit, which then takes the appropriate initialization actions.

Although the standard cell library does provide all the necessary functionality for VLSI design, it is designed mainly for Application Specific Integrated Circuits (ASIC) design, which is distinguished from other design procedures in that all its components are static. This means that the clock can be slowed down and even stopped without loss of any data and the functionality can be resumed any time following that. In this context we can make the following observations concerning the library provided.

- out of the 8169 standard cells in our design, 1950 (about 23%) are inverters, 269 of which are with driving strength $\times 3$ and 128 of driving strength $\times 6$. Consequently, the three kinds of driving strength provided are sufficient for most of the circuits, but the large number of inverters can be justified only by the fact that a nand gate followed by an inverter is faster than the and gate and also the same holds for the nor and or gates. Therefore, the control unit, which contains 615 out of the 1553 simple inverters, was optimized for speed following the above observations.

- The *non*-inverting gates, although they contain the inverter, they are supplied with only one kind of driving strength. In cases where a large fanout has to be driven, either a buffer has to be inserted at their outputs or the respective inverting gate has to be used with a large buffer at the output. The designer would have been benefited from the availability of a series of such standard cells with various driving strengths; in the present configuration they are included in the form of soft-cells which consume extra space and routing resources.

- Although fall-through latches are included, their use in one-phase clocking schemes is not recommended. Unfortunately two-phase clock generators were not included, although classic CMOS design is based on it. The Design of two-phase clock generators is not possible with the library components. Therefore, the area consuming edge-triggered flip-flops are used with a single phase external supplied clock.

The implementation is based on the above library of components, a single phase external clock

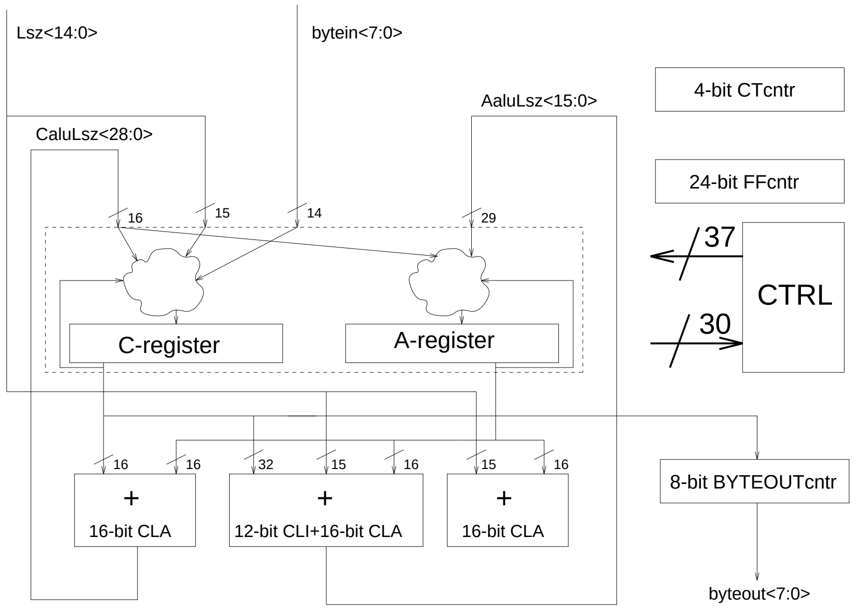


Fig. 4.1: Binary Arithmetic Coding

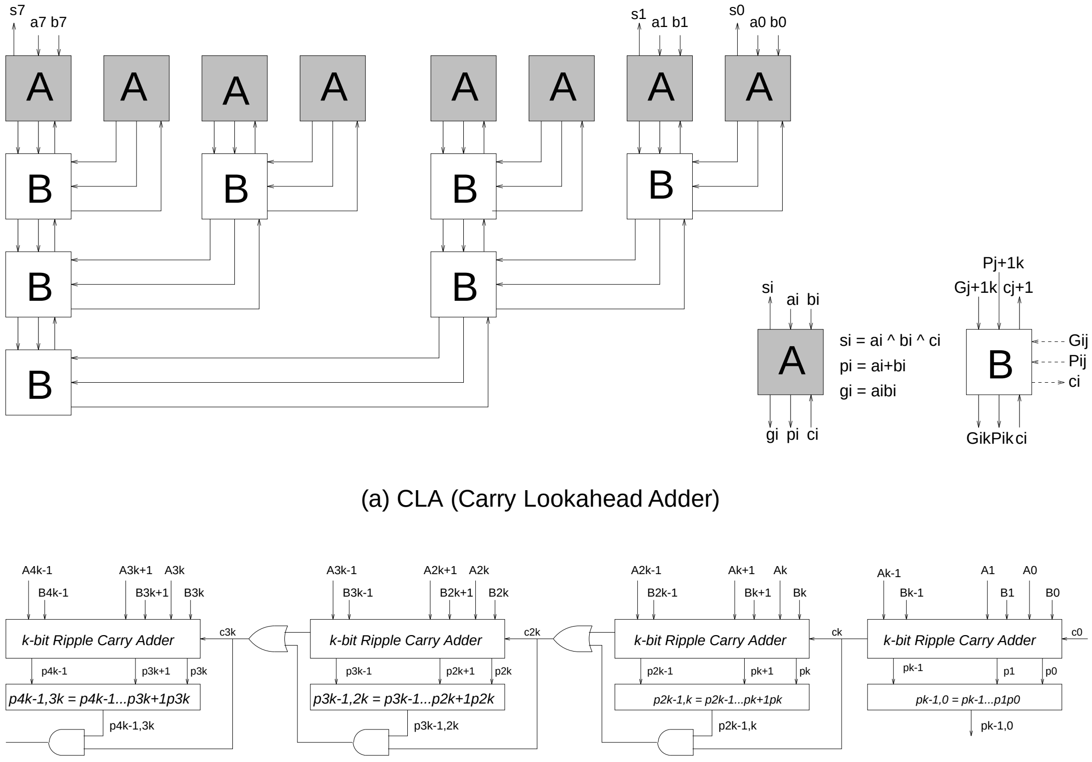
of cycle time of 50 ns with a 40%–60% duty cycle and simulations used maximum and minimum industrial delays for the standard cell components and for the generated regular structures. More details on the Design Environment will follow in the next chapter. The next sections of the present chapter are dedicated to the presentation of the circuits designed based on the standard cell library.

4.2 Arithmetic Coding Unit

The circuits that are included in the arithmetic coding unit are shown in Fig 4.1. They include the A and C registers and the circuits that select in each cycle their next values. Also two 16-bit adders and one 28-bit adder are included, which generate the results specified by the operations in the coding procedures (sec. 2.4 and sec. 2.5). The necessary operations that have to be performed on the values of the C and A registers are as follows:

- On the value of the C register:
  1. C[27:0] $+$ lsz5[14:0].
  2. C[27:16] $-$ lsz5[14:0].
- On the value of the A register:
  1. A[15:0] $-$ lsz5[14:0].
  2. A[15:0] $-$ lsz5[14:0] $<$ lsz5[14:0] $\Longleftrightarrow$ A[15:0] $-$ (lsz5[14:0] $\ll 1) < 0$.

During encoding, the C register may stay the same or may be incremented by the LPS (lsz5 [14 :0]) interval (the MPS interval is ordered above the LPS interval), in which case the operation (C [27:0] $+$ lsz5[14:0]) has to be performed. Either the LPS interval (lsz5[14:0]) or the MPS interval may be assigned to register A. In the latter case, the operation (A[15:0] $-$ lsz5[14:0]) has to be calculated. Also, in order to check for possible renormalizations, the value of lsz5[14:0] has to be compared against the MPS interval. Since the comparison of two numbers is equivalent to the comparison of the difference of the two numbers against zero, the operation (A[15:0] $-$ (lsz5[14:0] $\ll 1) < 0$) also has to be calculated in parallel with the previous two, resulting in a



(a) CLA (Carry Lookahead Adder)

(b) CSKA (Carry Skip Adder)

Fig. 4.2: Carry Lookahead and Carry Skip Adders

necessity of a total of three Adder-Units. The operation performed on the C register is a full 16-bit addition plus a 12-bit increment since the (lsz5[14:0]) value has only 15-bits precision. Actually, the outcome of the above operations are not assigned directly to the corresponding C or A register, since the results may have to be shifted left by one position in the case that a renormalization has to be performed, as it is shown in the circuit which selects the next values of the C and A registers in Fig. 4.1.

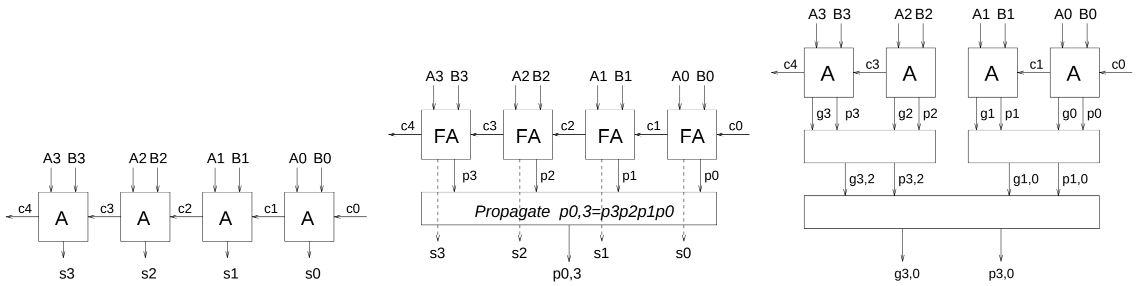
The operations that have to be performed during decoding are also the comparison of (A [15:0] $-$ (lsz5[14:0] $\ll$ 1) $<$ 0), the calculation of the MPS interval size (A[15:0] $-$ lsz5[14:0]) and the subtraction of the probability estimation from the C register (C[27:16] $-$ lsz5[14:0]).

Other blocks which are included are the CT counter, the 0xFF byteout counter and the byteout counter, which is the temporary output buffer until the carry-overs are resolved. Since the output buffer may have to be incremented due to a carry-over, it is implemented as a counter. The 0xFF byteout counter is used to keep track the number of 0xFF bytes that have been stacked and the CT counter is used by the renormalization procedures.

4.2.1 Adder-Units implementation

As it is known from basic digital design, carry-lookahead adders are fast, but also costly. In the design of the 16-bit Adders with the standard cell library provided to us, two such designs —Carry Lookahead Adders (CLA) and Carry Skip Adders (CSKA)[2]— were considered, implemented and timing analyzed. The CLA and CSKA implementations were based on the analysis on Adder design in [22]. The respective implementations suggested are shown in Fig. 4.2(a) for

[2]Carry Skip Adders are not abbreviated as CSA which is reserved for Carry Save Adders



(a) *Carry Ripple Adder* (b) *Carry Skip Adder* (c) *Carry Lookahead Adder*

Fig. 4.3: 4-bit Adder design examples

the CLA and Fig. 4.2(b) for the CSKA. The CSKA is partitioned into identical blocks of size k. Better results can be achieved by making the interior blocks larger than the blocks at the beginning and at the end. Examples of 4-bits CRA (Carry Ripple Adder), CLA, CSKA are shown in Fig. 4.3.

The following terms have to be calculated for each adder and for the 4-bit adder example [22].

CRA4 $4s + 4g + 4p + 4c$.

CSKA4 $4s + (4g + 4p + 4c) + P$.

CLA4 $4s + (4g + 4p) + (2p + 2g + 2c) + (p + g + c)$.

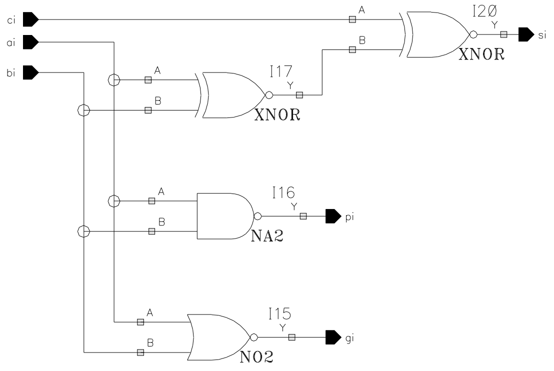
where $g = a_i b_i, p = a_i + b_i, s = a_i$ xor b_i xor $c_i, c_{i+1} = g + pc_i, i = 0, \cdots, 3$. The simple CRA has to calculate 4 sum terms, 4 generate terms and 4 propagate terms. The 4 carry-out terms are calculated using the 4 generate and propagate terms. The CSKA must additionally calculate a propagate term $P = p_{ik-1} \cdots p_{ik}, i = 0, \cdots, n/k$ using the k propagate terms of each one of the blocks of k CRA, that is, the CSKA differs from the CRA only by the last term P. The propagate terms in each CRA have to be calculated for the carry-outs. The CLA calculates 4 sum terms, 4 generate and 4 propagate terms in the A blocks, 3 generate, 3 propagate and three carry-out terms in the following two levels of B blocks. The propagate term P in the CSKA can be regarded as equivalent to the $3g$ terms of the CLA in terms of space requirements. Therefore, the main difference between the CLA and the CSKA, is that the former, has also to calculate the generate terms of each sub-block. As was mentioned in a previous section (4.1), the standard cell library that we used, also included an one and two-bit Full Adder cell. The two-bit Adder cell receives two 2-bit numbers and a carry-in and generates a 2-bit sum and a carry-out. It doesn't output the generate and propagate terms, since the carry-out is probably implemented with a complex CMOS function in order to save space.

The A and B blocks, used in our CLA implementation are given in Figs. 4.4(a),(b),(c). Since we used *nand* and *nor* gates instead of the *and* and the *or* gates, two B blocks are necessary. The A block (Fig. 4.4(a)) contains two *xnor* gates, one *nand* and one *nor* gate. In terms of area, the two *xnor* gates occupy about the same space as the one-bit full-adder cell, although the latter also contains the generate, propagate and carry-out terms. In terms of speed of the sum terms, they are produced with about the same delay. The drawback of the full-adder cell is that it doesn't supply the generate and propagate terms, which are necessary to the operation of the other levels of the CLA, but they are not necessary for the CSKA. Therefore, two CSKA designs were implemented, one with primitive standard cells and the second using two-bit full-adder cells. Another possibility is to use CLA with blocks of 4-bit CSKA in the first level and normal B blocks in the other levels. Using this approach we avoid to calculate the 3 generate blocks in the second level and the third level in each of the 4-bit components and there is a need only to calculate the propagate blocks like the CSKA.

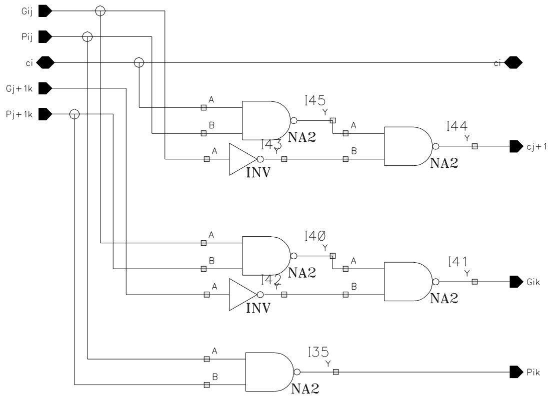
Adder	cells	length	vtime static		vtime hybrid		vlog delay		Est. Area
		μm	typ	max	typ	max	typ	max	μm^2
CLA16	179	2567	4.7	10.1	5.7	12.4	6.5	14. 2	$482^2\,\mu$m^2
CLA28	225	3276	5.9	12.7	6.8	12.6	8.5	18.5	$543^2\,\mu$m^2
CSKA16_2	46	1372.5	10.2	22.4	11.3	22.7	9.2	20.3	$345^2\,\mu$m^2
CSKA28_2	95	2141	14.7	32.2	11.5	23.1	13.8	30.3	$426^2\,\mu$m^2

Table 4.1: Adder areas and delays

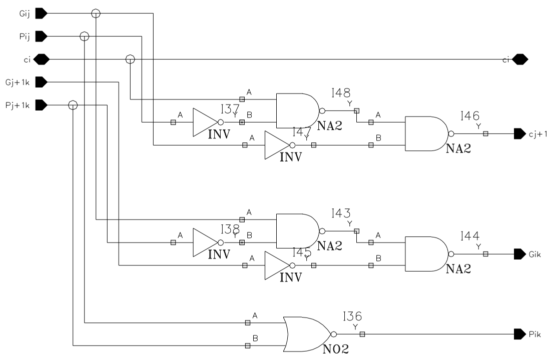
All the Adder alternatives were implemented and also the timing was analyzed. The results are shown in Table 4.1. In addition to the timing analysis, the number of standard cells (column 1), the length of the total number of cells in microns (all the standard cells have the same height, $47\,\mu$m), minimum and maximum times for Veritime static analysis, veritime hybrid analysis and Verilog with fanout delay calculation timing analysis for the maximum path are also shown. Finally, the last column indicates the area estimation together with the necessary routing of the standard cells, which is significant larger than the product length $\times$ cell-height.



(a) A block cell



(b) B block cell invert



(c) B block cell

Fig. 4.4: Carry Lookahead Adder cells

The various timing simulations will be presented in the following chapter in the Design Environment. Briefly, we can note that the verilog simulator and the respective delay calculator

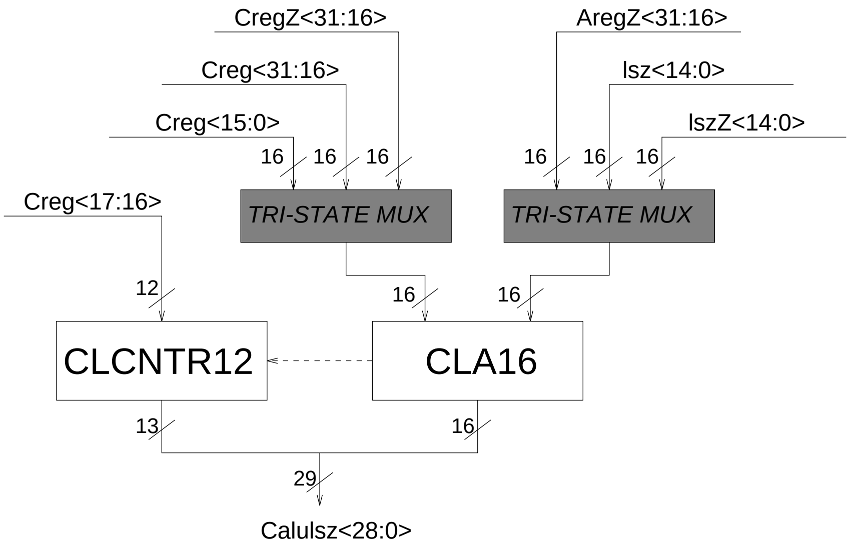


Fig. 4.5: C register Adder Unit

for the ES2 $1.0\ \mu$m process calculates both the intrinsic delays of the library cells plus the delay due to the fanout at each gate and therefore, it is the most accurate one. The veritime simulator in our case, only calculates the intrinsic delays, because the delay calculator supplied with the library was only for the Verilog simulator. The result from the veritime analysis would be correct if the respective delay calculator for the specific foundry were incorporated. The result shown in Table 4.1 can therefore only be an indication of the actual delays. The difference of the static from the hybrid analysis, is that in the former case no simulation is actually performed and the circuit is analyzed statically. The worst/best path between a prespecified set on starting points and end points is calculated. In the hybrid analysis, a simulation is also performed with a set of test-vectors and is therefore more accurate. In Table 4.1 the *typical* and *maximum* delays for each analysis are shown. In the case of the timing simulation, the typical and maximum delays are for the maximum path. Also minimum, typical and maximum delays for the shortest path can be calculated, but are not included in Table 4.1.

As can be inferred from the values in the above table, the CLA is significant faster than the CSKA. The latter, is much smaller and contains less standard cells, because it uses the Adder cell provided with the library. Since we have set a target cycle time of 50 ns, the most appropriate Adder for our purposes is the CLA. As was mentioned in the beginning of this section, the operations that have to be performed on the C register value are either an addition or a subtraction of the probability estimation value Qe = lsz5[14:0]. Also, as was pointed out in sec. 2.4, during the `final_bits` procedure, the highest number in the final interval has to be calculated by adding the last A register value to the C register. Therefore we must also calculate this value. This is done in the same Adder that calculates the other operations on the C register. The respective multiplexer circuit is shown in Fig. 4.5.

4.2.2 A and C registers

The A and C registers are normal D flip-flops on which at each clock cycle a new value is loaded. From the coding procedures these registers have to be loaded at each cycle with the following values:

- initialization of C register to 0x00000000 and A register to 0x00010000.
- C and A remain the same by loading their previous values.

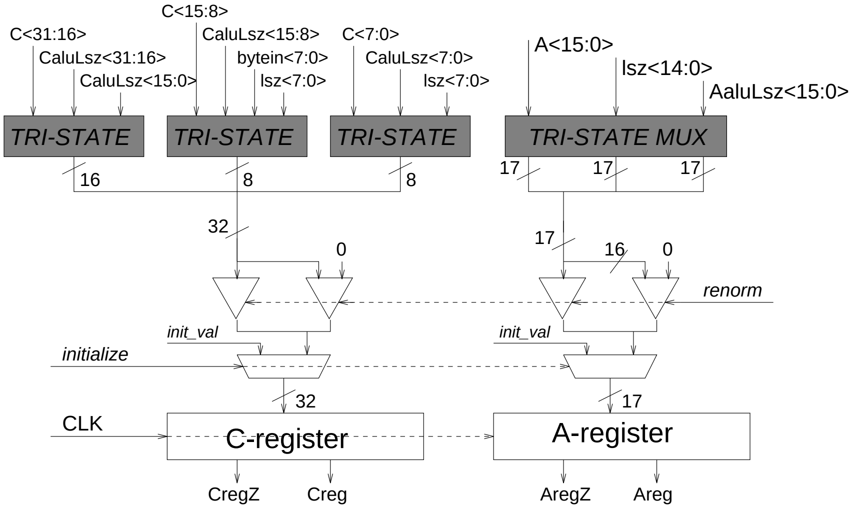


Fig. 4.6: A and C registers

- A register may also load the LPS interval value (lsz5[14:0]) or the MPS interval value (A − lsz5[14:0] = Aalulsz[15:0]).
- the next C register values are split into the 8 least significant bits, the bits [15:8] and the high order 16 bits of C. The least significant byte of C may also be loaded with the 8 least significant bits of the LPS interval (lsz5[7:0]) or with the least significant byte of the corresponding C register adder result (Calulsz[7:0]).
- the second byte of C may be loaded additionally to the corresponding bits of the least significant byte with the new coded byte during decoding.
- the 16 high order bits of the C register are selected among the value Calulsz[31:16] during encoding and the value Calulsz[15:0] during decoding.

The corresponding diagram is shown in Fig. 4.6. A two level tri-state tree performs the necessary selection among the different values for the next values of A and C. The first level selects among the different values on which the second level performs any necessary renormalizations by shifting them one bit to the left. Any additional required renormalizations are performed in the following cycles by stalling the coding pipeline.

4.2.3 Arithmetic Coding Control Unit

The arithmetic coding control unit is purely combinational and is designed with the standard cell library. The FSM does not contain explicit state storage cells, but as such can be regarded the bits of the CT counter. The value of the CT counter and the selected I/O signals are presented in the FSM diagram of Fig. 4.7 during encoding and Fig. 4.8 during decoding. Signals, which are not important to the understanding of its operation, are omitted from the present description. The total number of input signals are 37 and the outputs are 300. The control unit contains 294 standard cells of which 77 are inverters. It uses the outputs of the various storage cells, which are stable from the beginning of each cycle and the value of one of the Adder Units, which compares the MPS interval to the probability estimation Qe. Time-critical functions that have to be calculated are:

1. A 24-bit comparison of the 0xFF byte counter for equality to zero and 0xFFFFFF.
2. A 4-bit comparison of the CT counter for equality to zero, to one or to two.

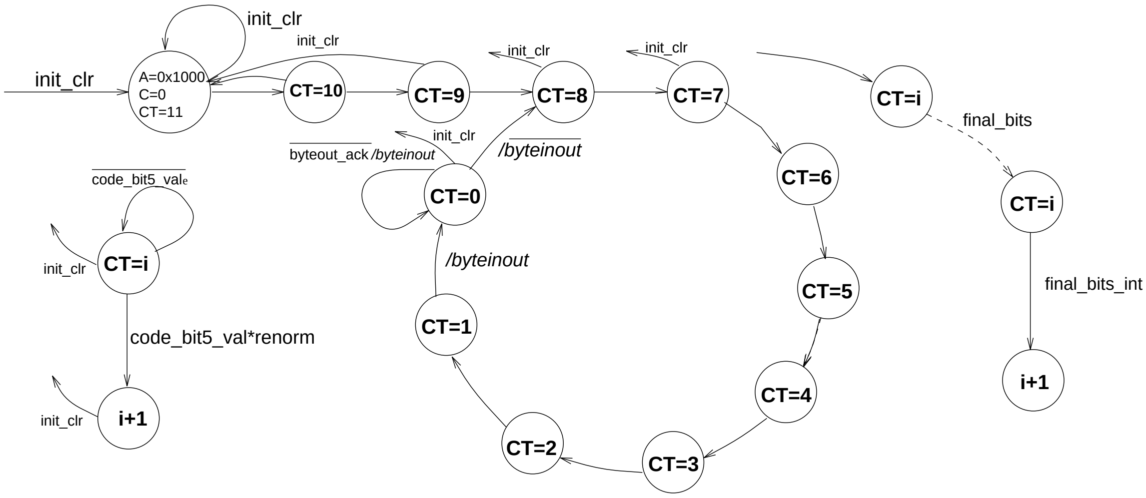


Fig. 4.7: Arithmetic Coding Control Unit

3. An 8-bit comparison of bits 27 to 20 of the C register for equality to 0xFF.

In addition, we have to check if the next C register value, as it is calculated from the circuit which supplies the next values of the C and A registers, generates any carry-over, that is bit position 28 of the C register is set. This, together with the necessary comparison against the MPS interval prescribe the clock cycle that can be achieved. As we have seen in Table 4.1 (sec. 4.2.1), the maximum delay for the 28-bit C value operation is 18.5 ns. This value has to pass through the next selection and renormalization circuit to supply the possible carry-over. The circuit which selects the next values of the C and A registers depends on control signals, which are a function of the MPS interval comparison and therefore the cycle time of the arithmetic coding unit increases to approximately 40 ns.

The operation of the control unit in encode mode is as follows. Initially the *init_clr* signal initializes the CT counter and the C and A registers to their appropriate values. The CT counter is then decremented according to the input signals on the left of the diagram in Fig. 4.7. Whenever the coding bit presented to the arithmetic coding unit is valid and a renormalization has to occur, the value of the C counter is decremented. After reaching zero, the first byte of data is written into the *byteout* counter. As the first byte has to traverse 11 bit positions to the left in the C register, the first time the CT counter is initialized to 11. Following that, the CT counter is reset to the value 8 and from then on, it is cycled through the values from 8 to 0. On each such cycle one byte of coded output is written to the byteout counter and therefore a minimum of 9 cycles are needed for 8 bits. The additional cycle number 8 is needed since we may have to increment the previous byte in the byteout counter before it can be transmitted to the DOUT register in the system bus interface unit. When we have reached cycle 1 and we have a byte to transmit in the following cycle, the *byteinout* signal is asserted signifying that the byte in the byteout counter can be written into the DOUT register. If the DOUT register is currently empty, the *byteout_ack* signal will be high and the byte from the byteout counter will be written into the DOUT register. On cycle zero, we decide that bit positions 28 to 20 of the C register should be cleared and a new byte will be written into the byteout counter. In case that the $DOUT$ register is full, the *byteout_ack* signal will be low and the FSM waits with CT at zero until the *byteout_ack* signal goes high, that is the current byte in the $DOUT$ register is read from the external circuit. Then the above process is repeated.

During the period that the *byteinout* signal is high and the *byteout_ack* is low, the *byteinout* signal remains asserted. When the byte from the C register is written into the byteout counter, the CT counter is reset to 8 and the cycle is repeated. It should be mentioned that the CT counter

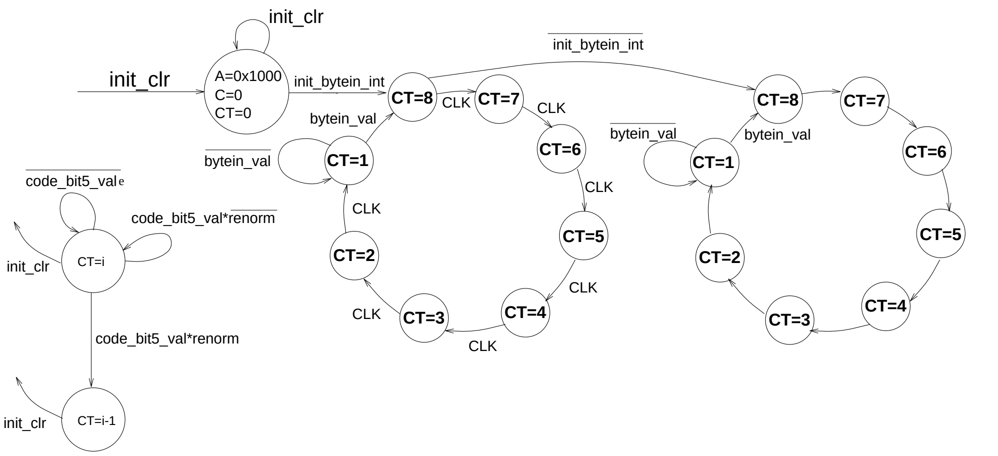


Fig. 4.8: Arithmetic Coding Control Unit during decoding

changes its value only when a renormalization is needed. This does not mean that the arithmetic coding unit stalls the pipeline, because as it was mentioned, the first renormalization is incorporated into the coding cycle. Only the following renormalizations stall the pipeline. Since the previous pipeline stages receive some control information from the arithmetic coding unit, this additional renormalizations are only performed when there exist a valid coding bit from the probability estimation unit. At the end of the encoding procedure, when all the DCT-coefficients have been coded, the `flush` procedure has to be executed. The respective transition from a CT counter value i to $i+1$ is shown at the left of Fig. 4.7. In contrast to the normal operation when CT only changes when a renormalization has to occur, at the end when the *final_bits* signal is high, the first clock cycles calculate the final interval pointer accordingly to the encoding procedures. The *final_bits* signal also starts an internal counter for the final three bytes. During the period that *final_bits_int* is high the CT counter is incremented on each cycle and the C register is shifted one bit to the left until all the final code bits are transmitted to the output.

The corresponding operations during decoding are shown in Fig. 4.8. Although similar to the encoding operations, the main difference is its extended initialization phase. As was presented in the `byte_in` procedure, which corresponds to the initialization routine, during this phase two coded bytes have to be received and inserted into the high order 16-bits of the C register. Therefore, when the decoding process is ready to start, an external signal *init_bytein* with one cycle duration signifies the begin of the initialization routine by resetting an internal two-bit counter. This counter is incremented for the first two coded bytes that are inserted into the C register and after that, the initialization has finished. During this period, no decoding is done and after each one of the two bytes are inserted into the C register in positions 15 to 8, 8 shift left cycles follow which will fill the high order 16-bits of the C register. Under normal operation, the coded bytes are inserted into positions 15 to 8, since it would complicate the multiplexing circuit of the next C register value if we decided to place that byte in one of the three high order byte locations of the C register.

The *bytein* handshake, which corresponds to the *byteout* handshake during encoding, is performed by asserting the *byteinout* signal. This time, it signifies that a new coded byte has to be inserted into the C register. If the *bytein_val* signal is high, which signifies that a new byte is in the DIN register, that byte is inserted into the C register. As long as the *bytein_val* signal is low, the byteinout signal remains high. When the byte is transferred from the DIN register to the C register the corresponding *bytein_val* signal is cleared. After the two bytes are read into the C register, the decoding process can start by cycling the CT counter from 8 to 1. This time, the minimum number of cycles to decode a byte is 8, because when the CT counter has reached

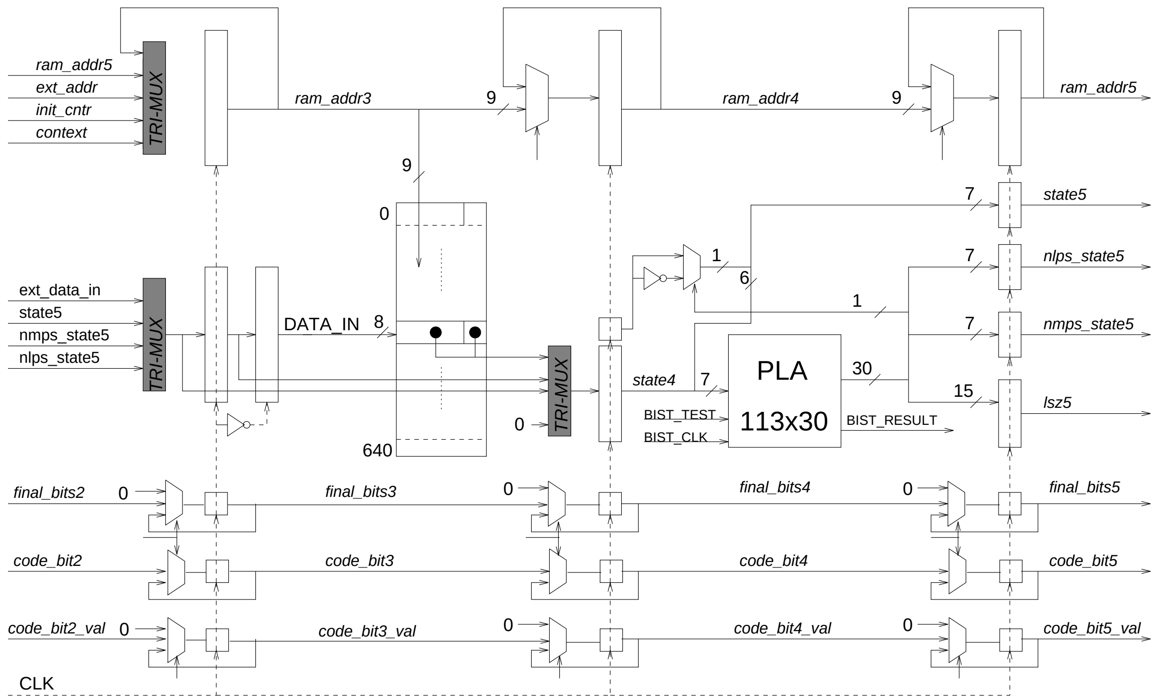


Fig. 4.9: Probability Estimation Unit

cycle 1 and the *bytein_val* signal is high, the new byte can be inserted immediately into the C register.

4.3 Probability Estimation Unit

The Probability Estimation Unit occupies stages 3 and 4 of the coding pipeline and was briefly presented in Fig. 3.9. A more detailed diagram is shown in Fig. 4.9. As was pointed out, in sec. 2.8.2, we use 320 storage locations for a total of 295 contexts. Also, since we have decided to implement two such context areas (sec. 1.3), the total size of the static RAM that will hold the contexts is 640, with a word size of 8. Therefore, the address bits are 10; nine of which are used for the context and the tenth for the selection of the context area. The probability estimation state machine is implemented with a PLA and occupies the 4th stage; it will be described in the following section.

We also use three flags to keep track of the coded bits in each stage. A number at the end of a signal signifies the stage to which it belongs. Therefore, the bits to be coded are named *code_bit2*, *code_bit3*, . . ., *code_bit5*. A validity bit in each state signifies if the respective *code_bit* to be coded is valid or invalid due to a pipeline stall. In case that no code bits are available, the respective valid bits are zero. An additional *final_bits* flag is used to signify the end of the coding process. As before, after all the previous bits have been coded, this flag passes also through the pipeline and at the time that it reaches the arithmetic coding unit, the coding process has finished. Three tri-state based multiplexing circuits, one at the input of the context RAM address, another for the data and the third at the input of the index into the probability estimation state machine, perform the various forwarding operations in the pipeline as described in Sec. 3.7. The Address register of the static RAM may be loaded with:

1. the address from *stage5* (arithmetic coding) in the case that an update of the context has to be performed.

2. an external address if the chip is operating in test mode and in this case this address is

supplied from the external circuit.

3. the value of an initialization counter which is used to clear the context RAM before coding begins.

4. a new context from the statistical model unit.

The corresponding *Data In* register of the static RAM can be loaded with:

1. the external data value from the *DIN* register during test mode.

2. the data value from *stage5* with the *MPS* bit switched.

3. the next state in case of a *MPS* interval coding.

4. the next state in case of a *LPS* interval coding.

The index into the probability estimation PLA may be loaded with:

1. zero in the case that the index from the static RAM corresponds to the context of the sign of the DC DCT coefficient, which is coded with a fixed probability of 0.5.

2. the value from the static RAM *output* in stage 3.

3. the value from the static RAM *input* in stage 2.

4. the value that was selected from the previous tri-state mux for the *Data In* register in stage 2.

4.3.1 Static RAM interface circuit

The static RAM, which contains the context areas, is generated with the static RAM generator supplied by ES2. The RAM generated has an access-time of 17.3 ns and a precharge time of 8.7 ns. The problem with the RAMs is that they are double-edge clocked, that is the address is latched at one of the edges of the clock, for example the positive edge, and the data are latched at the opposite edge of the clock. Therefore, as can be inferred from the circuit in Fig. 4.9, the address is latched at the positive edge of the clock into the external register, and as a result it is stable during the clock cycles and it is latched by the RAM at the negative edge of the clock. Accordingly, the data values to be written into the RAM must be valid during the positive edge of clock. This means, that the value which is written into the external register at the positive edge is also buffered through an additional register which is clocked with the negative edge. Since the address is latched into the RAM at the negative edge of the clock, after the access time has passed, the value is latched into the pipeline register of stage 4.

4.3.2 Probability estimation finite state machine

As was briefly shown in the second chapter, the probability estimation state machine is an FSM with 113 states. Each state is assigned a 14-bit Qe probability estimation, a 7-bit index to the state machine if an MPS is coded, another 7-bit if an LPS is coded and a one-bit interval interchange flag. Therefore the most natural way to implement the state machine would be to use a ROM with 113 words of 30 bits each. Unfortunately the ROM compiler provided and the ROMs designed in the $1.0\,\mu$m ES2 process are very slow. The cycle time formula for the ROMs generated by the ES2 compiler are according to [23]:

$$48.75 + \text{numRows} * 5.1/256.0 + \text{numCols} * 6.75/256.0,$$

where numRows must be a multiple of 2, therefore it is recommended to increase the numRows as much as possible, which is in our case 64. The increase from 32 to 64 adds 32 new rows and decrements the number of columns from 120 to 60. Due to a limitation of the ROM compiler the maximum number of rows in our case could only be 32 and therefore, the generated ROM had a cycle time of 52 ns. Adding the setup times for the pipeline registers and a one level multiplexing circuit, in case that we would like to use a BIST circuit for the ROM, we sum up to 55–58 ns. Since we had set a maximum cycle time of 50 ns, the use of such a ROM for the state machine was not possible.

An alternative choice was to use a PLA, which would function in our case as a ROM and would be much faster and also smaller in area. The drawback with the PLAs are that they consume static power. In order to use the PLA as a ROM, the number of product terms has to be identical to the number of the words in the ROM. Therefore the AND plane functions as the row decoder of the ROM and the OR plane actually contains the words.

Due to a size limitation of the PLA compiler the RAM was split in two parts. The first PLA contained 64 product terms and the rest 49 terms were placed in the other. This resulted in a maximum delay for the large PLA of 26 ns. The respective size and delay characteristics of the PLAs and the ROMs are given in Table 4.2.

ROM & PLA compiled megacells	Physical Size width × height	Access ns	Cycle ns	Max. Consumption static	dynamic
ROM 113x30 (32x120)	1200 μm × 576 μm	31.70	52.6	1.17 mW/MHz	
PLA 64x30	257 μm × 522 μm	31.58	none	13.30 mA	7.49 mA
PLA 49x30	257 μm × 432 μm	27.01	none	11.20 mA	7.30 mA
PLA 113x30 (total)	514 μm × 954 μm	31.58	none	24.50 mA	14.79 mA

Table 4.2: Probability estimation state machine ROM and PLA design characteristics

4.3.3 PLA Build In Self Test

As we have seen, in order to test effectively the static RAM and the PLAs for faults, direct paths to the inputs and outputs of such regular structures must be available so that, independently from the normal operation of the circuit, values can be written and read from all the locations of the regular structure. With the static RAM we have accomplished this by providing a direct external path to the respective I/O signals and therefore we are able to write and read the locations of the RAM. The same approach could also be applied to the PLAs. Another alternative is to use the ES2 megacell generator option to include a BIST soft-circuit of the respective regular structure and to let this circuit perform the necessary fault-tests. Most often, the output of such a BIST circuit is either a go-no go signal or a signature value which is compared to a pre-calculated correct value. In the case that the signature resulted from a self-test is different than the pre-calculated, a fault is present in the megacell. The advantages and drawbacks of the BIST approach to manually generated test-vectors [24] are:

- 100% fault coverage of single stuck-at faults and better than 99% for multiple faults.
- automatic generation of test-vectors for megacells.
- area and time penalty due to BIST circuit and megacell multiplexer gate delays.
- at maximum three additional primary pins for the first megacell and one more pin for each additional megacell. In the case that the output of the BIST circuit is multiplexed with the primary pins of the chip, only one additional primary pin is needed.

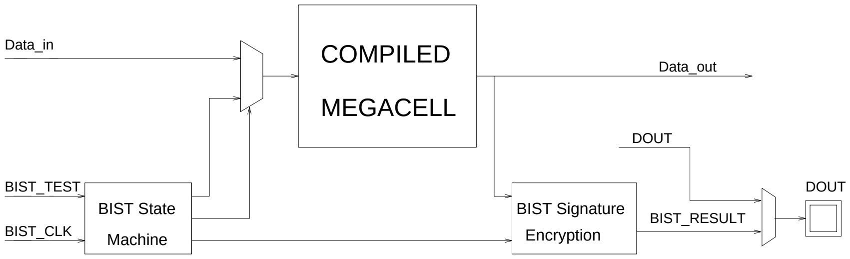


Fig. 4.10: Arithmetic Coding Control Unit

In order to test the PLA manually, all possible input values from the *AND* plane should be generated and the output of the PLA should be compared against the expected output from the *OR* plane. The don't care terms should be expanded to both zero and to one while the output should remain the same. Input terms, which are not included in the AND plane, should drive all the outputs of the PLA to zero.

The other approach is to use a BIST circuit. The interface of the BIST circuit to the megacell is shown is Fig. 4.10. Two additional input signals, *BIST_TEST* and *BIST_CLK* are required. The first should be a primary input pin, but the second can be assigned also to the normal CLK pin. A multiplexer, at the input of the megacell, selects among the normal input data and the data supplied by the BIST state machine. The output of the megacell is fed to the signature evaluation circuit and the signature is serially shifted out to the output pin (BIST_RESULT), which may also be a primary output pin, but can be also multiplexed to one of the normal output pins of the chip. In the later case, the BIST_TEST signal selects among the normal output and the BIST_RESULT signal.

Also, when more then one megacell, each with its own BIST circuit, are included in the same VLSI chip, the BIST_TEST and BIST_CLK signals can be shared among the BIST circuits but each of them contains its own BIST_RESULT signal. In this case, the tests can be performed in parallel for each megacell and every BIST_RESULT signal may be multiplexed with one of the primary outputs of the chip.

A sample test sequence is shown in Fig. 4.11. Two numbers, $n1$ and $n2$, are assigned to each BIST circuit. The first controls the number of clock cycles that are needed to test the megacell by the BIST state machine and the second signifies the number of bits in the signature. The required total number of cycles is $n1 + n2$. During this period the BIST_TEST signal must be high. The transition of the BIST_TEST from low to high starts a BIST sequence. The first five bits at the BIST_RESULT output, after the BIST_TEST signal is asserted, should be the bit-sequence 01100, which signifies proper initialization of the BIST circuit. After the first $n1$ cycles, $n2$ bits of the signature are shifted out from the BIST_RESULT pin which can be compared to the correct pre-calculated signature.

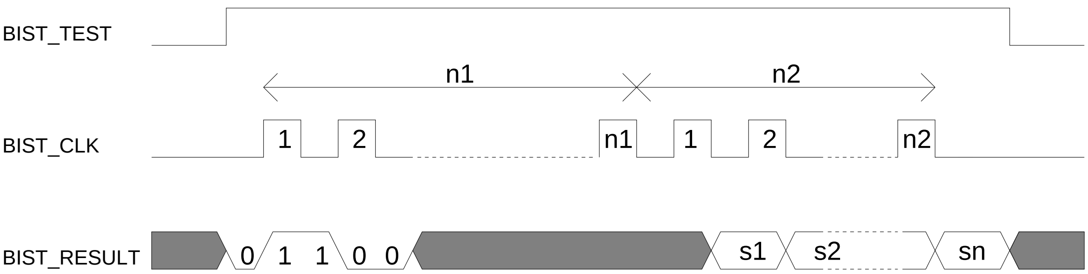


Fig. 4.11: BIST signal sequence

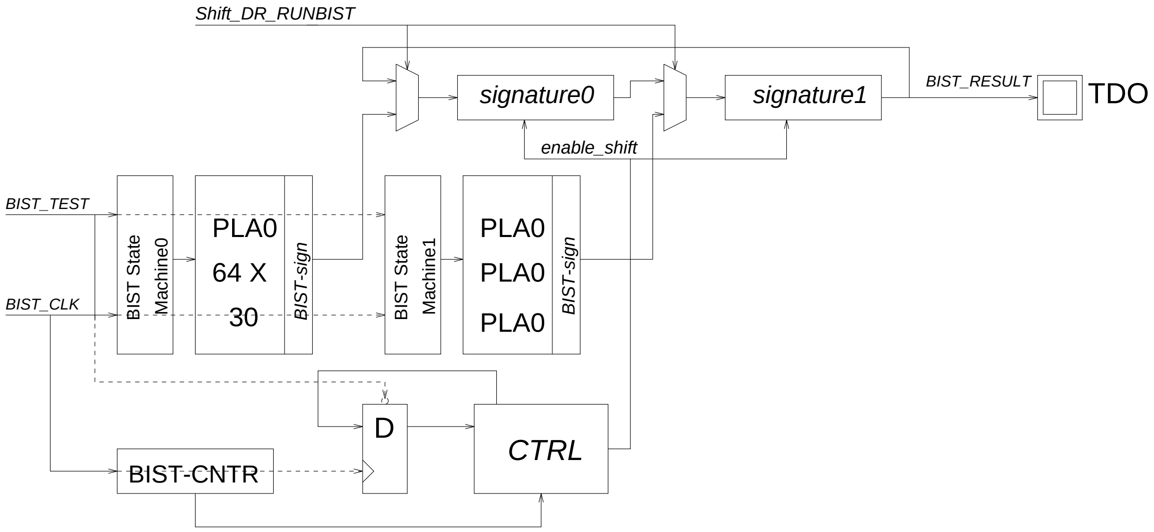


Fig. 4.12: BIST signal sequence

We have used a modified version of the circuit shown in Fig. 4.10 in order to interface the BIST circuit of each PLA to the JTAG controller. The JTAG controller and its associated instructions will be described later in this chapter. Presently, we can assume that when the JTAG controller executes the RUNBIST instruction, it asserts a respective signal which functions as the BIST_TEST signal. Furthermore, the BIST circuit is clocked with the JTAG TCLK signal and the duration of the BIST is directly controlled from the external circuit through the JTAG controller. Since both PLAs are tested in parallel, their respective signatures are also available at the same time. Therefore, we have added two 16-bit registers, which will shift the two signatures from the BIST circuits. These two registers form a 32-bit shift-register, which can be shifted out by the TDO pin of the JTAG controller. During the BIST, the JTAG controller remains in a respective state and during that period the two 16-bit signature registers receive their values from the BIST circuit. When the two signatures have been shifted into these registers, the JTAG moves into another state, from which the two signatures are serially shifted out to the TDO pin. A respective diagram is shown if Fig. 4.12. According to the figure, the signature register shifts only for the first 16 cycles of the BIST counter. During the first 16 cycles the D-type flip-flop is zero and therefore, the multiplexer at the input of the signature register receives the value from the BIST signature encoding circuit of each PLA. After the first 128 cycles, the D-type flip-flop is set and from then on, the signature registers shift their contents to the TDO pin and their values are also shifted back into the same registers, so that after 16 cycles, they contain their initial values. The two signatures remain in the 32-bit register and can be read subsequently, if necessary.

4.3.4 Probability Estimation Control Unit

The probability estimation control unit contains 270 standard cells, which are approximately 375 gates. It is responsible for the enable signals of the valid flags and the code bit register, the pipeline registers and moreover, for selecting the next values, as it was described in the previous section, taking into account the various possibilities of forwarding. Therefore, it has to compare the next state value on an MPS and the next state value on an LPS with the current state and if these two values differ and a transition is performed, which is signified by an input signal from the arithmetic coding unit; then it has to update the current index in the static RAM. Many of its output control signals are based on signals, which are known relatively late in the clock cycle, therefore, these additional functions performed on them contribute to the final clock cycle that can be achieved.

class	[6:3]	**bin**	[2:0]
NLARGE	0000	C0	00 T
NSMALL	0001	SIGN	01 T
ZERO	0010	PSZ1	10 T
PSMALL	0011	NSZ1	11 T
PLARGE	0100		
mag-sz	[6:0]	**mag-bit**	[6:0]
X20	01 0100 T	B36	*not used*
X21	01 0101 T	B37	10 0101 T
X22	01 0110 T	B38	10 0110 T
⋮	⋮	⋮	⋮
X31	01 1111 T	B47	10 1111 T
X32	10 0000 T	B48	11 0000 T
X33	10 0001 T	B49	11 0001 T
X34	10 0010 T	B50	11 0010 T

Table 4.3: DC coefficient coding

4.4 Statistical Model Unit

The statistical model unit is responsible for generating the appropriate binary decisions for each DCT coefficient and the corresponding contexts. Therefore, as it was described in sec. 2.7, it contains the Sz register which holds the DCT coefficient and the M register, which is used to generate the magnitude category and magnitude bits binary decisions, as presented in sec. 2.8.

As it was described in sec. 2.8.2, 295 different contexts are in use for the 64 DCT coefficients in one 8×8 block of image data. The assignment of storage locations to contexts is not performed linearly, in order to avoid any necessary arithmetic operations of the different contexts, which are used to code a DCT coefficient. Such a coding may require the contexts in different storage locations and their respective indexes may not have any relation to each other. Therefore an addition may be required in order to obtain the next index. This is avoided by using a special assignment of storage locations to contexts, as shown in Tables 4.3 and 4.4, which only require boolean operations in order to get from one context to the following in the list of binary decisions that have to be generated in order to code a DCT coefficient.

4.4.1 DC bin assignment

As explained in the second chapter, the DC coefficient is coded differently than the 63 AC coefficients in one block. DPCM is used for the DC coefficient and the difference between the previous and the current DC DCT coefficient is coded, generating the contexts of the Table 4.3. The total number of different contexts used for the DC component is 49, therefore normally 49 storage locations would be necessary. According to the assignment in Table 4.3, the number of storage locations is 100 for two context areas and a 7-bit index is used to address one of the different 100 contexts. The DC area occupies positions 512 to 639 in the context RAM. Therefore, 10 address bits are needed of which the three high order bits are 100 for the DC assignment. The least significant bit, shown in Table 4.3 as T, is used to select one of the two context areas.

The *LU* parameters, which are used to classify the DC difference into one of the five classes are used to specify bits [6:3] and the binary decision specifies bits [2:1]. The combinations of the five classes with the four binary decisions comprise the first 20 contexts. Storage locations 20 to 34 are used for the magnitude category binary decisions and storage locations 37 to 50

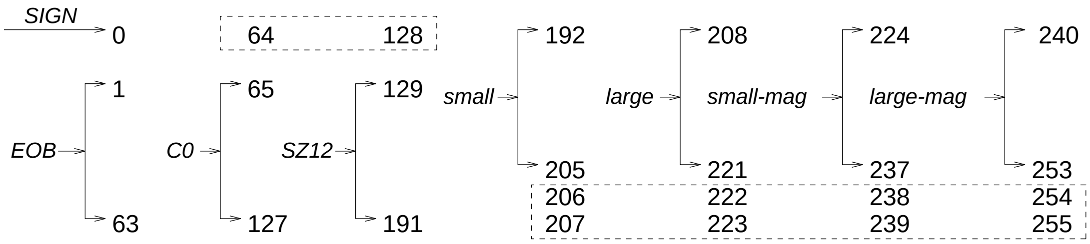


Fig. 4.13: AC coefficient contexts to storage locations mapping

1	SIGN	0x00	0T 0000 0000
64	EOB	0x00 *or* K	0T 0000 0000 *or* K
64	C0	0x40 *or* K	0T 0100 0000 *or* K
64	SZ12	0x80 *or* K	0T 1000 0000 *or* K
16	$K \leq Kx$	0xC0 *or* mag-sz	0T 1100 0000 *or* mag-sz
16	$K > Kx$	0xD0 *or* mag-sz	0T 1101 0000 *or* mag-sz
16	$K \leq Kx$	0xE0 *or* mag-bit	0T 1110 0000 *or* mag-bit
16	$K > Kx$	0xF0 *or* mag-bit	0T 1111 0000 *or* mag-bit

Table 4.4: AC coefficient coding

for the corresponding magnitude bits. Storage location 36 is unused. As can be verified from the assignment of the storage locations to the X and B contexts, the transition from the magnitude category to the corresponding magnitude bit is performed by switching only the bits in positions [6:5]. If the bit in position 5 is set then the bits in positions [6:5] are switched to 10, otherwise they are converted to 11. In case that we assigned the 14 magnitude bits to locations 36 to 49, an addition would be necessary to switch from the magnitude category to the magnitude bit context, which we have avoided.

4.4.2 AC bin assignment

The 246 AC DCT contexts are assigned to storage locations as shown in Fig. 4.13. In reality 256 storage locations are used, 10 of which remain unused and since two such context areas are in use, the AC DCT coefficient storage locations occupy positions 0 to 512 in the context static RAM. All the AC DCT coefficient use the same SIGN context, and the corresponding binary decision is coded with a fixed probability.

An EOB (End Of Block), C0 (Coefficient is 0), SZ12 (SZ greater than 1, SZ greater than 2) binary decision is assigned to each of the 63 AC coefficients. The following 16 magnitude categories and magnitude bits are conditioned on the external supplied Kx value. In the case that the coefficient number K is smaller than the supplied Kx value (by default 5) the small areas are selected and in contrary if the coefficient number K is larger than Kx, the large areas are selected. The detailed assignment is shown in table 4.4. A simple *or* operation is necessary to calculate the context, given a coefficient number K. The flag T, again signifies the context to be used.

4.4.3 LU classification

As was mentioned previously, the DC coefficient difference is classified into five categories, before the coding process can proceed. The same holds for the AC coefficients, which are classified according to the position they occupy in the 64 block of DCT coefficients. The five categories for the DC coefficients which depend on the two 4-bit external values L and U were presented in sec. 2.8.1 and are repeated here:

1.			last_dc_diff	$<$	-Udec
2.	-Udec	$\leq$	last_dc_diff	$<$	-Ldec
3.	-Ldec	$\leq$	last_dc_diff	$\leq$	Ldec
4.	Ldec	$<$	last_dc_diff	$\leq$	Udec
5.	Udec	$<$	last_dc_diff		

where Udec $= 2^U$ and Ldec $= 2^{L-1}$. At first glance, in order to classify a 16-bit value represented in two's complement arithmetic, a comparison has to be performed against the values: *-Udec, -Ldec, Ldec, Udec*. The *Ldec* and *Udec* values can be easily calculated from the supplied *L* and *U* values, with two 4 to 16 bit decoders. The comparison of two numbers is a costly boolean operation and is equivalent to the subtraction of the two numbers. As we can observe the *Ldec* and *Udec* values have only one bit set and therefore the comparison of a number, which is a power of two, for greater or equal than, to the other, can be performed by decrementing the number by one, inverting it and *and-ing* it with the second number. If the result has at least one bit set, than it can be concluded, that the second number is greater or equal than the first. For example, the comparison $A \geq 2^B$ can be performed by calculating $\overline{2^B - 1}$ *and* A and afterwards *or-ing* all the bits of the result. If the final result is high, then the relation holds. Using this procedure, we have replaced a full addition of two numbers with a half-addition (subtraction of 1) followed by a 16-bit *or* operation. Unfortunately, with this method we can only test for greater or equal. In order to classify the DC DCT coefficients in one of the five intervals, as mentioned above, we also have to compare the two values for only greater and for less or equal. Therefore, additional to the previous operations, an equality test of the two numbers has to be performed, and finally, using these four results, the DCT coefficient can be classified accordingly. The resulting circuit is shown in Fig. 4.14. As the assumption in the previous operations

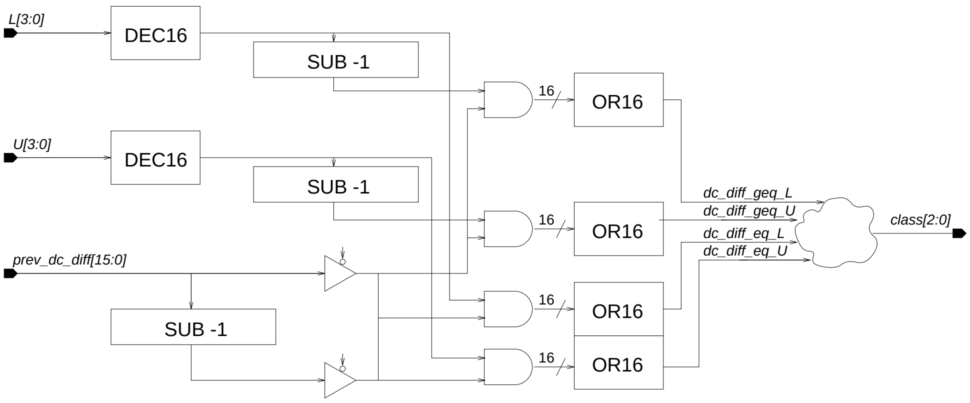


Fig. 4.14: LU classification circuit

is that the number, which is a power of two, is positive and we also have to check against the respective negative value, the comparisons are reversed in order to check always against the positive values of *Ldec, Udec*. Therefore, it may also be necessary to use the negative value of the previous DC DCT coefficient. In order to accomplish that, a multiplexing circuit is used to select either the previous DC DCT coefficient or its respective negative value. The latter is achieved by decrementing the inverted value of the negative DCT coefficient.

4.4.4 DC & AC coefficient coding

The rest of the statistical model unit contains the Sz and the M registers which are used to generate the contexts for the DC and AC coefficients. The use of the Sz and the M register was

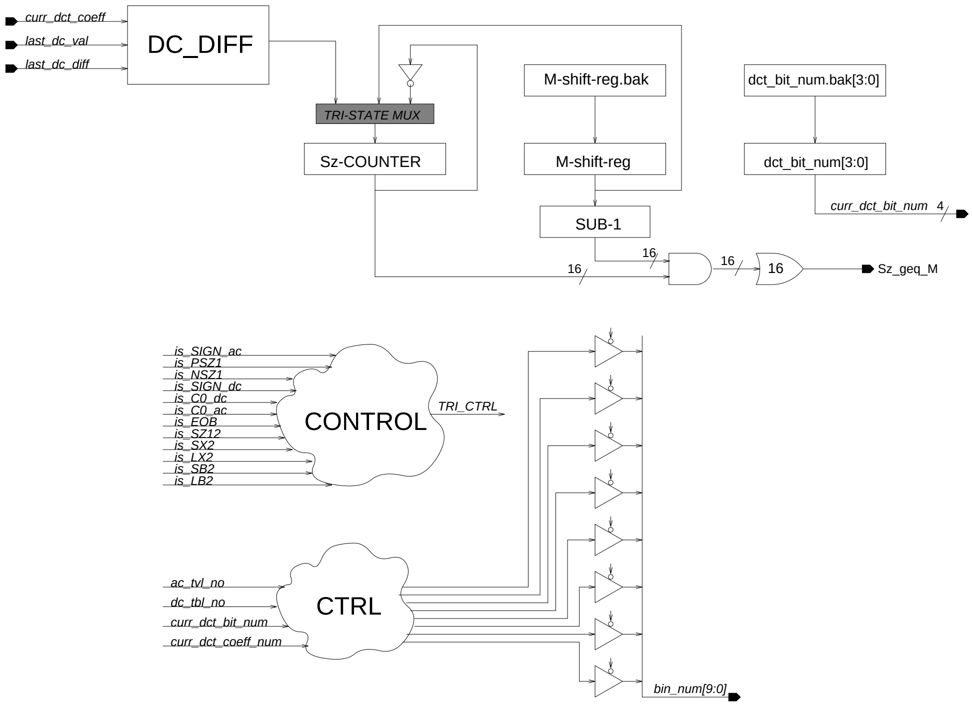


Fig. 4.15: Sz and M registers —bin calculation.

presented in sec. 2.8. The Sz register contains the absolute value of the DCT coefficient and the M shift register is used to define the magnitude category of the respective DCT coefficient. Once the magnitude category is coded, the M register is used as a mask value in order to code the respective magnitude bits of the Sz register.

The detailed circuit is shown in Fig. 4.15. The Sz register may be loaded with the DC difference value, which is generated by the *DC_DIFF* block, when a DC coefficient is coded, or it may be directly loaded with the AC DCT coefficient. In the latter case the DC_DIFF block simply passes the input *curr_dct_coeff* to its output. Since the Sz register must contain the absolute value, it may be necessary to invert the value loaded which is performed in the following cycle. Therefore, another path from the Sz register to its input is used. Finally, during decoding of the magnitude bits, when the corresponding bit decoded is high, the value of the M register is *or-ed* with the Sz register.

Two copies of the M register and two copies of the respective 4-bit counter, which is used to count the magnitude bits, are used during decoding, as was previously described in sec. 3.7.2, to revert to the previous values of the same register, in cases where the coding procedure issued from the statistical model was not correct. This occurs, during magnitude category decoding or when a sequence of zero DCT coefficients are decoded. As was pointed out, in such cases, each coding process depends on the output of the previous decoded binary decision and therefore the pipeline is essential useless. By predicting the next binary decisions that have to be coded and sending the respective contexts to the probability estimation machine, a new bit may be decoded at each cycle. When the prediction is wrong the already issued binary decisions are canceled in the probability estimation unit and the M register and the bit-counter are restored to their previous correct values.

The context number is generated through a tri-state multiplexing tree, which is shown in Fig. 4.15. In practice a tree is used in the place of the 8 tri-state buffers shown, in order to remain within the fanout limitation of the tri-state buffers. The various *is_XXX* signals are supplied from the control unit which will be described in the following sections and concern the binary decision that will be coded in each cycle. The context is generated from the DC and AC contexts area indexes (*dc_tbl_no*, *ac_tbl_no*), the position of the DCT coefficient in every 8×8 block of DCT coefficients, the 4-bit position of the magnitude bits and various combinations with the *is_XXX* signals according to tables 4.3 and 4.4.

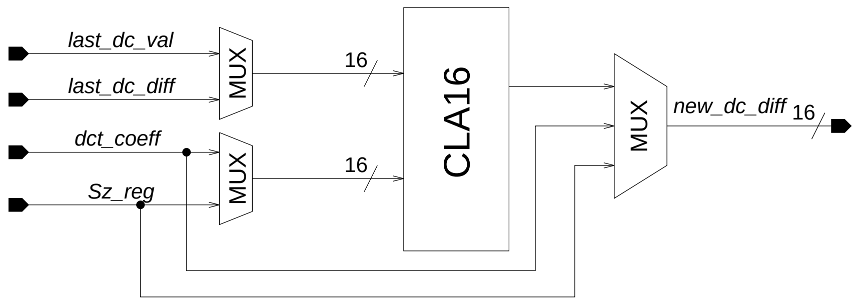


Fig. 4.16: DC_DIFF circuit.

The DC_DIFF circuit is shown in Fig. 4.16. The Carry Lookahead Adder is used to supply the DC difference or to regenerate the DC coefficient from the previous DC coefficient and the new DC difference. Therefore, its first operand is selected among the previous DC difference and the previous DC coefficient and the second operand is either the current DCT coefficient or the Sz register which contains the DC difference. When an AC coefficient is coded, the multiplexer at the output of the Adder passes the input DCT coefficient to the output.

4.5 I/O Unit

The I/O interface unit contains the external addressable registers and its function is to interface the rest of the chip to the system and the DCT bus. The respective diagram is shown in Fig. 4.17. It is comprised of two submodules which perform the necessary synchronization of the external signals, since they are asynchronous to the internal clock. The synchronized databus values and addresses from the system bus are used either to read or write the 16 external addressable registers, according to the value of the *SysRD* and *SysWR* signals. The DCT-bus interface contains the 8×16 FIFO that is used during encoding as well as during decoding. In the first mode, external DCT-coefficients are written into the FIFO and are read from the internal circuit. The external DCT-coefficients may either be 16-bits or may be sign-extended internally from 11-bits to 16. Since the signals which signify if the FIFO is *empty* of *full* (*fifo_empty, fifo_full*) are asynchronous to the internal clock and the control unit has to check these flags in order to read the DCT coefficients, they have to be synchronized. During decoding, the FIFO is written from the internal circuit which uses the same clock as the rest of the chip. Therefore, the *empty* and *full* flags have to be synchronized by the external circuit in order to receive the decoded DCT coefficients. Next, we will describe the two synchronization circuits for the system and the DCT-bus.

4.5.1 Microprocessor System Bus Interface

The system bus interface is comprised from the signals *SysBus*, *SysAddr*, *SysCS*, *SysRD*, *SysWR* and *SysRDY*. Their operation was briefly described in Sec. 3.3. All the signals are active low. A write cycle, which is shown in Fig. 4.18(b), starts by placing the address and the data into the address and data bus respectively and following that, asserting the signal *SysWR*. Since the *SysWR* signal is used to clock the internal edge-triggered registers, which will latch the address and the data, the latter must be valid a certain setup time prior to lowering the signal *SysWR*. Also, in case that the *SysCS* is inactive (high) the *SysWR* signal is ignored. After the *SysWR* signal is asserted, it first passes through the synchronization circuit so that it can be safely read by the internal control unit. This takes, as will be described shortly, approximately three internal

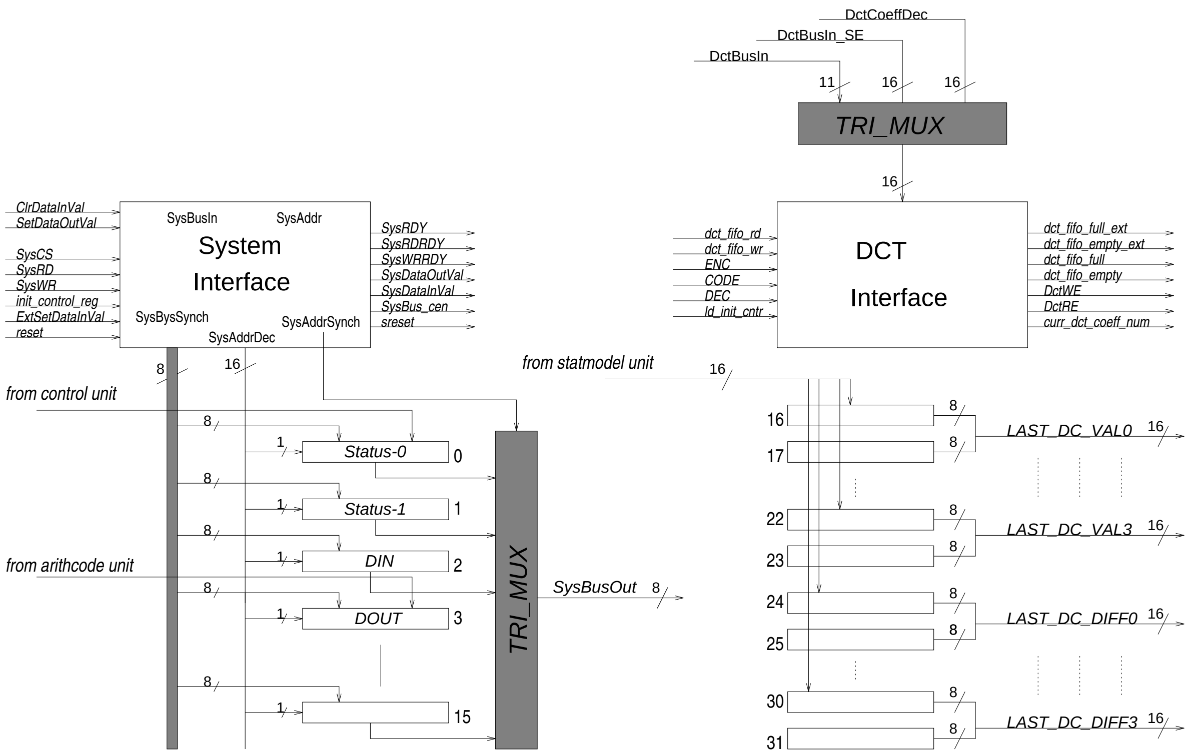
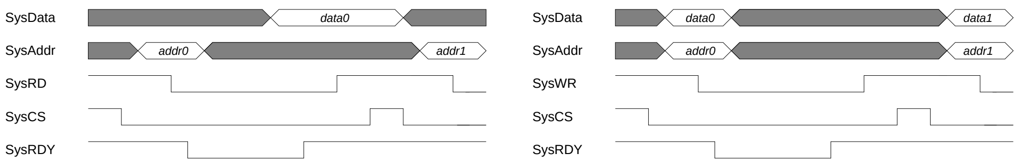


Fig. 4.17: I/O Interface Unit circuit

clock cycles. Immediately after the *SysWR* signal is asserted, the *SysRDY* signal goes low in order to insert if necessary, any wait-cycles for the external circuit. As long as *SysRDY* is low, no other read or write operation can be started. When the *SysWR* signal is read by the control unit, it writes the value latched from the data register into the appropriate register as specified by the address.

The read operation (Fig. 4.18(a)) is similar to the write operation except that the data-bus is configured in it's reverse direction. After the *SysRD* signal is synchronized, the value from the internal register, as specified by the latched address, is put on the data-bus. As long as *SysRD* is low the value remains on the bus and is valid. After *SysRD* goes high again and this transition is synchronized, the data-bus is tri-stated.



(a) read cycle (b) write cycle

Fig. 4.18: System read and write cycles

Synchronization circuits

The circuit used to synchronize the external signals is based on the suggestions in [25]. An asynchronous signal can't be used directly in synchronous circuits, since it may be observed in different state from different points of the circuit. A simple flip-flop may be used as a primitive

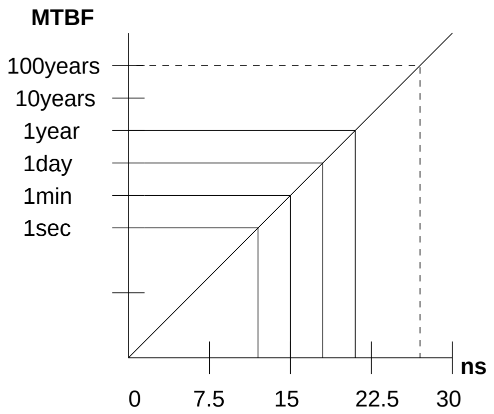


Fig. 4.19: Metastability MTBF against clock delay

synchronizer, which decides between the two values 0 and 1 when a transition occurs at the same time with an internal clock edge. The resulting signal after a specific delay is stable during the rest of the clock period and can be used safely.

The time that is needed for the the synchronizer to decide depends on the parameters of the fabrication technology and are analyzed in [26]. It is measured in Mean Time Between Failure (MTBF) for a specific synchronization time interval. Therefore, in order for the circuit to function properly, a high number of MTBF values should be chosen and then the respective synchronization wait-interval can be calculated. The asynchronous signals are fed through a number of flip-flops, which are connected serially and the time that these signals remain in the synchronizer circuits should be greater or equal to the time-interval calculated above. Such an evaluation was carried out by ES2 and the metastability results for the flip-flops of their library is described in [24]. The respective results are shown in Fig. 4.19. From the diagram it can be inferred that in order to have a MTBF value of 100 years the signal should stay for 30 ns in the synchronization circuit.

Since our maximum clock frequency is 20 MHz, two flip-flops connected serially are sufficient to sample the asynchronous input signals. Three variants of such synchronization circuits are described in [25], and are illustrated in Fig. 4.20. The first example uses two serially

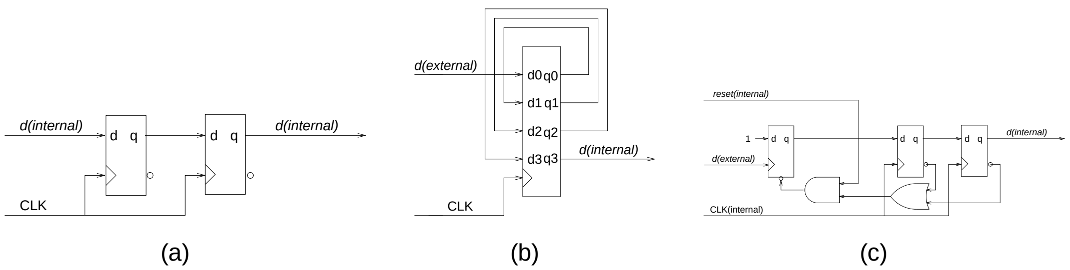


Fig. 4.20: Synchronizer circuit examples

connected flip-flops. The second uses a 4-bit register as a shift register that needs four cycles to propagate its input value to the output. The final example is used to detect positive edges in the input signal. Two cycles after a positive edge has occurred in the asynchronous input signal, a pulse with duration of two cycles appears at the internal synchronized data signal. During the two cycle period of the internal pulse, additional positive edges at the input are ignored.

Also, if the asynchronous input remains in the high position, after the positive edge, any additional pulses are not produced at the internal input. That is, only positive edges are sensed. We decided to use the third synchronization circuit in our design. Since we only need a one-cycle synchronized pulse, a simple modification to the circuit was performed, and the resulting circuit is shown in Fig. 4.21. Another flip-flop is added to the output of the original circuit, that

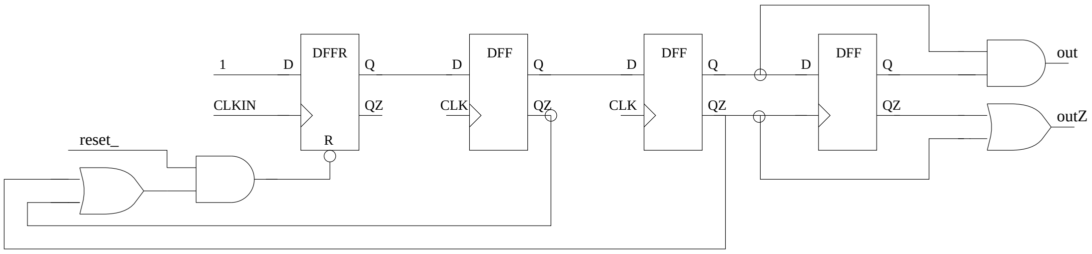


Fig. 4.21: Primitive synchronization circuit

is *and-ed* with the previous internal synchronized value, in order to narrow the two-cycle pulse into one. The corresponding synchronization circuit that will detect negative edges at the input, the inverse of the external data signal, has to be connected to the input of the synchronization circuit.

System bus synchronization

The application of the synchronization circuits to the system-bus interface signals (*SysRD, SysWR*) is shown in Fig. 4.22. Five synchronization circuits (synch) like the one shown in Fig. 4.21 are used to synchronize the three systems-bus input signals *SysRD, SysWR, SysCS*. The *and-ed* output of the *SysRD* and *SysWR* signals are also synchronized in order to generate the appropriate requests to the control unit, to read and write a specific register. The last two signals together with the initial three synchronized signals, are used to generate the output signals of the system bus (*SysWRRDY, SysRDRDY, SysRDY, SysBus_cen*). They are a function of the synchronized input signals and synchronous RS flip-flops generated from D flip-flops with an appropriate input function. The last of the output signal is used to control the tri-state drivers of

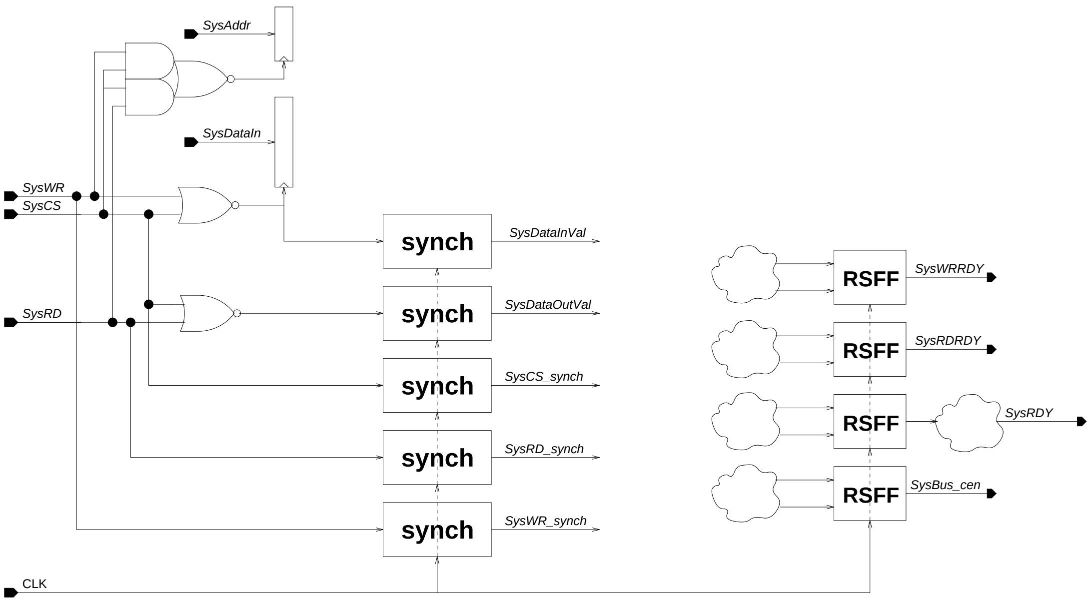


Fig. 4.22: System bus synchronization circuits

the bidirectional data bus. As can be seen from the circuit, the *SysRDY* signal depends on the asynchronous input signals as well as on the synchronized ones since this signal is not used by the internal circuit it can switch state immediately based on the input signals. This occurs, when the *SysRD* or *SysWR* signals are first asserted resulting in the *SysRDY* signal going low immediately. After the signals are synchronized (*SysRD_synch*, *SysWR_synch*) the *SysRDY* signal is deasserted.

4.5.2 DCT-bus Interface

The circuit of the DCT bus interface that contains the small FIFO is shown in Fig. 4.23. The input signals to the DCT interface circuit were presented also in Fig. 4.17. The FIFO is implemented with a dual-port static RAM that contains also two 3-bit counters, one for the read-port and the other for the write-port. During encoding, DCT coefficients are written through the write-port into the FIFO and are read from the internal circuit from the read-port. During decoding, the write-port is written from the internal circuit and the FIFO is read from the external DCT-unit.

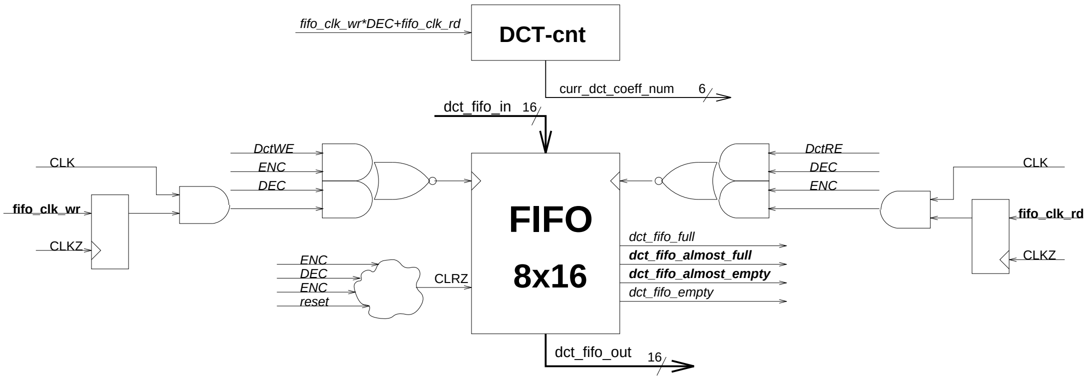


Fig. 4.23: DCT bus interface

Since the clocking of the FIFO is asynchronous, the clock inputs of the dual-port RAM depend both on the external and the internal signals. The writing of the FIFO is naturally performed synchronously to the internal circuit. In each cycle, the respective flag (full) is checked and if a DCT-coefficient is available it is written into the FIFO at the next cycle. The same holds also when the FIFO is read from the internal circuit. A problem which arise is that the flag signals provided by the FIFO, which is also generated by the megacell compiler in the ES2 library, are also asynchronous to the internal circuit since they depend on the read and write operations performed by the external circuit. For example, if the FIFO is empty, the respective *fifo_empty* flag will be set. When the external circuit writes a new DCT coefficient into the FIFO, this flag is cleared almost immediately. However the timing of the transition of the *fifo_empty* signal from high to low may occur at an arbitrary point in the duration of the internal clock period. Therefore, it can't be read directly by the control unit. The flags that the generator includes indicate only if the FIFO is empty or full. We have added another two signals that signify if the FIFO is almost full (75%) or almost empty (25%). These last two flags, together with the initial two, are used to interface safely the FIFO, as will be described in the following paragraphs. The detailed timing of the FIFO is described in [21]. Additionally to the FIFO, the DCT interface unit also contains a 6-bit counter that is reset before each coding process and it counts the DCT coefficients that are written into the FIFO, either from the external or from the internal circuit.

DCT bus synchronization

The signals that need to be synchronized from the DCT interface are the flag signals *dct_fifo_empty* and *dct_fifo_full*. These signals are more difficult to synchronize properly than the system

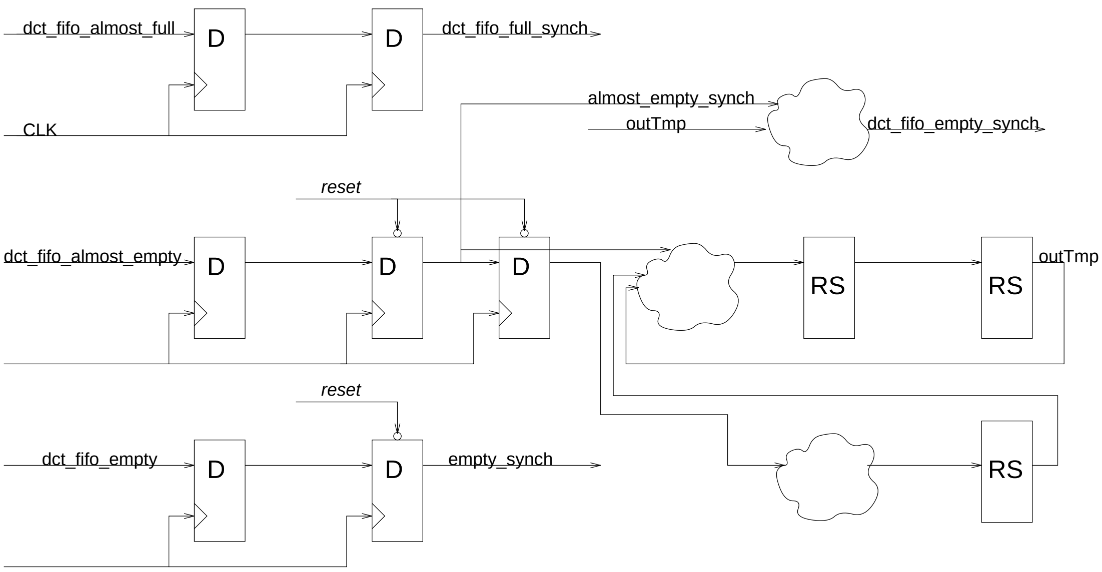


Fig. 4.24: DCT bus interface synchronization circuits

bus asynchronous signals, because during the clock-periods that the signals remain in the synchronization circuit, the internal circuit may assume wrong values that will overflow the FIFO during writing and reading. For example, if the flags take two clock cycles to synchronize and on each clock cycle a new value is read from the FIFO when the last DCT coefficient is read, it will take another two clock cycles until the control unit is informed that the FIFO is empty and should not try to read any more values. Unfortunately, during these two cycles, the control sees that the FIFO is not empty and will try to read another DCT coefficient resulting in a wrong FIFO read counter. This problem can be easily solved in the case that the internal circuit is writing into the FIFO, during decode mode. Then, it is possible to synchronize the *almost* full signals (75%) and when this is set, no more DCT coefficients are permitted to be written. Because it takes two cycles to synchronize the respective flag, the maximum number of DCT coefficients that can be written above the 75% marker are only two, which may only result in filling the FIFO. When the FIFO is emptied about to the half, two cycles less than the 75% marker, the internal circuit may start writing again into the FIFO.

The respective synchronization circuit is shown in Fig. 4.24. The actual problem is synchronizing the *dct_fifo_empty* signal. In this case, we can't use the *almost_dct_fifo_empty* signal because we have to read all the DCT coefficients. When the *almost_dct_fifo_empty* mark is reached, then if we continue reading, we will read the last two until the FIFO is empty. Should the external circuit write at that moment the last DCT coefficient, the *almost_dct_fifo_empty* signal will be set and the DCT coefficient will not be read by the internal circuit. Therefore, the synchronization circuit is a little more complex than the synchronization for the *fifo_full* signal. To solve this problem we have to synchronize also the *fifo_empty* signal and the first time when we pass through the low mark we set a respective RS flip-flop after the number of clock cycles that are needed for the synchronization of the *fifo_empty* signal. Then, we can safely use the synchronized *fifo_empty* signal in the place of the *almost_fifo_empty* signal. After we pass the low mark in the upper direction, the respective flip-flop is cleared, and we can use again the *almost_fifo_empty* signal. This circuit enables us to read all the remaining DCT coefficients in the FIFO.

4.6 JTAG Unit

The JTAG Unit embedded in our design is based on [17–19]. The ES2 library includes a JTAG controller and various boundary scan cells that can be used to execute the various instructions specified by the JTAG standard. The supplied JTAG controller implements 16 instructions of which 7 are the obligatory and the optional instructions specified by the standard, and the other 8 are user specified internal scan registers. Therefore, the instructions are coded using 4 bits. The reader is referred to the above specifications for the exact description of the testing architecture used by JTAG and the various procedures that such a unit has to implement in order to conform to the standard. We will briefly describe the JTAG FSM controller in the next subsection, through which also the probability estimation PLAs are tested for faults using the corresponding RUNBIST instruction as was described in sec. 4.3.3.

4.6.1 JTAG I/O signals

The JTAG standard specifies that four primary pins of the chip will be reserved for the JTAG controller and for the input and output test vectors which are applied in test mode. These pins are:

TCLK the JTAG internal circuit has its own clock that is applied to all the internal JTAG registers and to the controller.

TMS this signal is used by the controller to initiate the state transitions. The JTAG controller, as described in the next subsection, is an FSM and from each state only two transitions are possible. These transitions are controlled by this signal.

TDI this is the data input pin, through which input test vectors are serially shifted into the JTAG data and instruction registers.

TDO this is the data output pin, from which the output of a test is serially shifted out from a JTAG data register.

TRST* this is an optional active-low signal, used to bring the JTAG controller asynchronously to the reset state. The JTAG controller can be also brought synchronously to the reset state by applying clock signals to TCLK when TMS is set to 1. After a specific number of clock signals the JTAG controller will be in the reset state, independently from the initial state.

As mentioned above, two main categories of registers are used by JTAG —data registers and instruction registers. JTAG data registers can be any of the normal registers in the circuit, which are modified to function as shift registers during test mode. Since many such data registers can be used, the register currently active is selected by the instruction currently processed by the JTAG controller. Therefore, instruction registers are also in use, which select the appropriate data register and generate the necessary control signals for the various units. The data registers are used as scan-path test registers and the reader in referred to the literature [17–19] for the corresponding scan-path test architecture. We note here that all JTAG compliant ASICs have to include boundary scan registers at their primary pins in order to implement instructions that test the PCB interconnections between JTAG compliant chips on a PCB. In the next section we will introduce the JTAG controller states and briefly describe its use in order to test the PLAs in the probability estimation unit.

4.6.2 JTAG FSM controller

The JTAG controller is a 16 state FSM and it is presented in Fig. 4.25. Two transitions are possible from each state, which are controlled by the TMS signal. The states of the JTAG controller and the instruction currently selected are used to generate the JTAG control signals, which are applied to the data and instruction registers. Initially, as can be seen from the state transition diagram, when TMS is set to 1 and clock cycles are applied, the state machine will be in the `Test-logic-Reset` state after a maximum of six cycles. It remains in this state as long

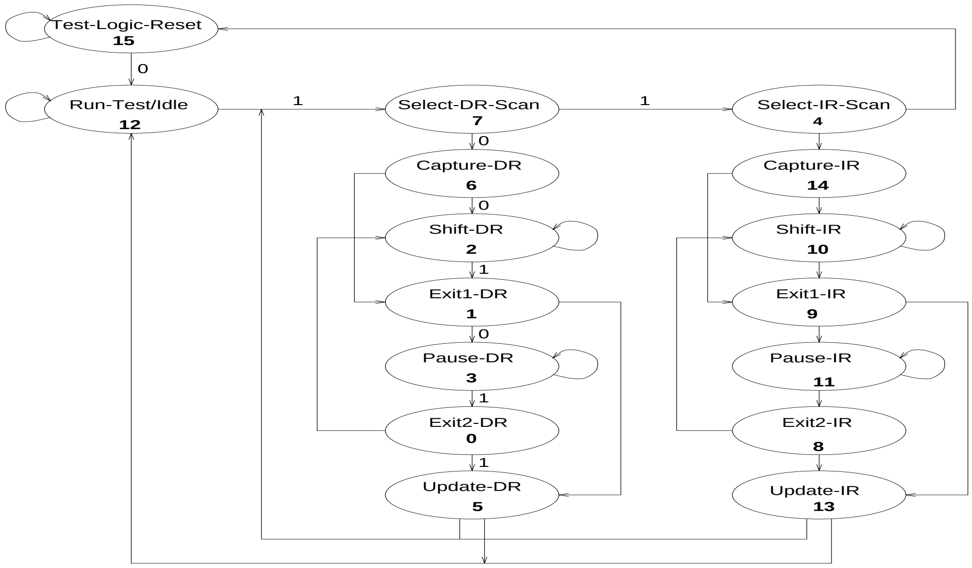


Fig. 4.25: JTAG Unit

as TMS is set. When the JTAG pins are in the high-Z position the internal circuit guarantees that the signals will be in the high-1 position and therefore the JTAG circuits will be inactive. The FSM is positive edge triggered, that is, the state transition occur at the positive edge of the clock. The actions of the test circuit occur at the negative edges of the TCLK signal. The most significant control signals generated by the JTAG controller are the following three for each of the data registers:

ClockDR this is the clock signal applied to the test data register. Since the actions occur at the negative edge of the TCLK signal, the low to high transition of this clock signal occurs at the negative edge of the TCLK signal.

ShiftDR this signal controls the data registers inputs. When this signal is active the test register functions as a shift register and a value is inserted into it through the TDI pin.

UpdateDR this is used to update the test register either with the value shifted in from the TDI input pin or from the internal circuit. The test data registers are configured in a way which permits a new value to be shifted in without disturbing the current value at the output of the register. This is accomplished by using two flip-flops instead of one. The first contains the registers output and the second the test-vector which is currently shifted in. When the UpdateDR register signal is asserted the first flip-flop receives the contents of the second and this value is applied to the internal circuit.

As can be inferred from the above signals, the states in the JTAG FSM are used to control the capture of internal data and the shifting of the test-vectors into the test registers from the TDI pin and out of them to the TDO pin.

The instructions that have to be supported by a JTAG compliant test circuitry are:

EXTEST is used to test through the boundary scan cells the test board interconnections by applying a specific test vector to the output pins and sampling the corresponding test vector from the input pins. This presupposes that all the chips on the board are JTAG compliant.

SAMPLE, PRELOAD the SAMPLE instruction is used to take a snapshot of the signals flowing through the boundary scan cells, both at the input and output pins in the `Capture--DR` controller state, and then shifting the corresponding data out to the TDO pin in the

`Shift-DR` controller state. The PRELOAD instruction is used to load a specific value into the boundary scan register without interfering with the normal operation of the circuit in the `Shift-DR` state. Then, the controller enters the `Update-DR` state and the specified value is applied to the internal circuit. The PRELOAD instruction is used prior to the EXTEST instruction in order to ensure that all signals driven out of the chip are defined as soon as the EXTEST instruction is selected.

IDCODE, USERCODE instructions are used to supply an internal hardwired bit-string identifying the manufacturer and the part number.

The following two instructions are optional.

INTEST this instruction is used to test the internal circuitry by shifting into the test-registers a specific test-vector and shifting out the respective output of the circuit which is sampled in the Capture-DR state.

RUNBIST this instruction is used to test any BIST circuits in the chip.

The RUNBIST instruction is used to generate the appropriate control signals for the BIST circuits of the PLAs in the probability estimation unit. The use of this instruction by the JTAG controller is as follows: The RUNBIST instruction is one of the dictated instructions and is shifted into the JTAG controller by passing through the following JTAG states: `Run-Test/Idle`, `Select-DR-Scan`, `Select-IR-Scan`, `Capture-IR`, `Shift-IR`. It remains in the `Shift-IR` state until the 4 bits of the RUNBIST instruction are shifted in and then, it returns to the `Run-Test/Idle` state. On entering in this state the respective *BIST_TEST* signal of the BIST circuit is asserted and the BIST is started. After 280 cycles the BIST has finished and the two 16-bit signatures of the two PLAs have been shifted in the 32-bit signature register as described in sec. 4.3.3. Next, the JTAG state machine is brought to the `Shift-DR` state and the contents of the signature register is shifted out to the TDO pin, since the *RUNBIST* is currently active and the signature register is implicitly selected. The state machine remains in the `shift-DR` state for 32 cycles and then it is deactivated by entering the state `Test-Logic-Reset`. The contents of the signature register remains untouched and can be read as many times as necessary.

4.7 Global Control Unit

The Global Control Unit is responsible for initializing the internal circuits and contains the FSM that generates the appropriate control signals for the I/O Unit and for the respective local controls units.

An overview of the control unit is given in Fig. 4.26. It includes the FSM, five counters which are used to keep track of the DCT coefficients coded in each DCT block and the number of DCT blocks coded, and also various other D-type flip-flop used to save the state information across the FSM states. The FSM uses a five bit index to specify the current state for a total of 18 states. The combinational part of the FSM is described in *BDS* format, as will be presented in the forthcoming chapter on the Design Environment (sec. 5.1.2). In order to reduce the number of input signals to the FSM —resulting in less minimization and mapping synthesizing time— some of the combinational logic, which in most cases calculates if some of the counters are zero or have a specific value, is implemented outside of the FSM and only the results are used by the latter. Next we will describe the operations which are performed in each state. A reduced state transition diagram is shown in Fig. 4.27. Regardless of the current state, the next state will be the *ST_INIT* state if the *reset* signal is encountered. Also, when the *CODE* flag in the status register 0 is set, the state transitions in the FSM are freezed, that is the FSM remains at its present state until the *CODE* flag is cleared.

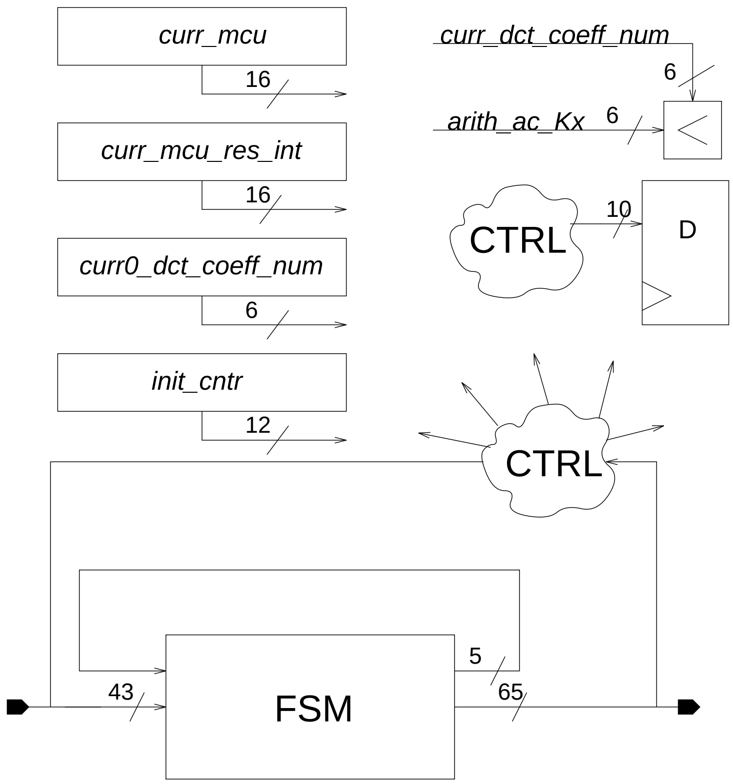


Fig. 4.26: Overview of the Control Unit.

ST_INIT This is the state in which the FSM enters on encountering the *sreset* (synchronized reset) in its active state, independent from the current state of the FSM. The input *reset* signal is synchronized and fed to the control unit. As long as the resulting synchronized signal is high, the FSM remains in the *ST_INIT* state. The only action performed in this state is to clear the 2 status registers. Upon deasserting of the *sreset* signal, the FSM goes to the *ST_WAIT* state.

ST_WAIT This state is entered after the *reset* signal is deactivated; in this state the various registers can be read and written in order to set-up a coding process. When the *CODE* flag in the first status register is set according to an external write, the next state is either *ST_INITENC* or *ST_INITDEC* according to the *E* and *D* flags of the status register 0. During the transition to the later stages, the initialization counter, the MDU/MCU counter, the MCU counter and the MCU restart interval counter and various other registers are cleared or loaded with an appropriate value. In case that the *EXT_ST_RAM_LD* flag is set, used in test mode to read and write the internal static RAM, the next state will be the corresponding *ST_EXT_ST_RAM_LD* state. If neither of the previous flags are set, the FSM remains in the current state. This holds true for the all the FSM states as well. The default transition is the current state.

ST_EXT_ST_RAM_LD In this state, the FSM monitors the *bytein_val* signal and upon it's activation the corresponding data value is written into the static RAM or the value read from the static RAM is written into the *DOUT* register. When the *EXT_ST_RAM_LD* signal goes low the FSM returns to the *ST_WAIT* state.

ST_INITENC This state is used to initialize all the necessary registers and counters in order to start an encoding process. The initialization counter, which was reset in the *ST_WAIT* state, is used to count from 0 to 640 in order to clear the internal context static RAM. After

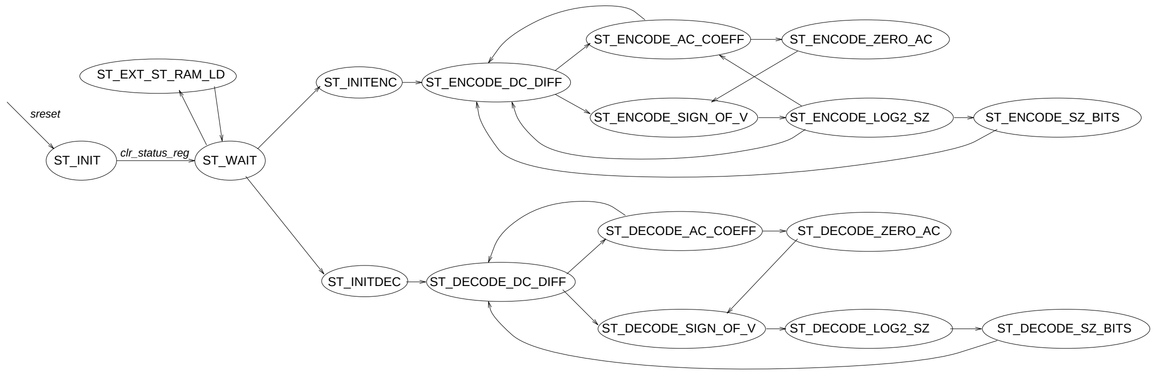


Fig. 4.27: FSM state transitions.

that, the Sz and the M register in the statistical model unit are cleared and the next transition is to the state *ST_ENCODE_DC_DIFF*, in which the encoding of the first DC DCT coefficient begins.

ST_ENCODE_DC_DIFF The first DCT coefficient is read from the DCT FIFO. If the FIFO is empty the FSM remains in the current state until a DCT coefficient is written into the FIFO. In case that the DC difference value is zero the next state entered will be the *ST_ENCODE_-AC_COEFF*, in the other case the *is_C0_dc* (Fig. 4.15) binary decision is coded and the next state entered is *ST_ENCODE_SIGN_OF_V*.

ST_ENCODE_AC_COEFF In this state a series of zero AC DCT coefficients are coded —when a non-zero DCT coefficient still remains in the 64 DCT block. In the case that all the rest of the AC coefficients are zero, an EOB binary decision is coded. In order to find out if a non-zero DCT coefficient still remains in the 64 block, the zero DCT coefficients are not written into the DCT FIFO but they increment a DCT coefficient zero counter which is cleared after each non-zero DCT coefficient. In the case that the last of the 64 DCT coefficients is encountered, an EOB decision is coded in the position in which the last non-zero DCT coefficient were encountered. When a non-zero coefficient is found before the DCT block is finished, the following actions are performed: First the zero DCT coefficients, that have incremented the respective counter are coded and then the coding of the non-zero AC coefficient is performed. The zero DCT coefficients are coded in the state *ST_ENCODE_-ZERO_AC*. If an EOB decision is coded, the next state will be the *ST_ENCODE_DC_DIFF* state for the next block of DCT coefficients.

ST_ENCODE_SIGN_OF_V This state is entered either for a DC or an AC coefficient. In the first case an *is_SIGN_dc* binary decision is coded and in the second case an *is_SIGN_ac* binary decision is coded. The M register is initialized and the next state will be the *ST_ENCODE_-LOG2_SZ* state.

ST_ENCODE_LOG2_SZ In this state the magnitude category of the DCT coefficients is coded. For this purpose the M register is shifted left until its value exceeds the value in the Sz register. Various possibilities have to be taken into account, such as coding an Sz register with value 1 or 2, coding with a negative or positive context (*is_NSZ1*, *is_PSZ1*) for DC coefficients and the binary decision *is_SZ12* for AC coefficients. Depending on the outcome of the coding procedure, the next state may be the *ST_ENCODE_AC_COEFF* state, when the absolute value of the DCT coefficient is 0 and the coding of the magnitude bits is not necessary or the *ST_ENCODE_DC_DIFF* state if the 64 block of DCT coefficients has been coded or in the most probable case the state *ST_ENCODE_SZ_BITS* in order to code the magnitude bits.

ST_ENCODE_SZ_BITS In this state the magnitude bits of the DCT coefficients are coded. During each cycle the M register is shifted to the right until it becomes zero. The corresponding bit of the Sz register, which is in the same position as the bit which is set in the M register, is coded providing the *is_B2* binary decision.

ST_ENCODE_ZERO_AC In this state a series of zero DCT coefficients are coded and the the corresponding counter is decremented until it reaches zero. Since another non-zero DCT coefficient follows, the next state will be the *ST_ENCODE_SIGN_OF_V* state.

ST_INITDEC This state is similar to the corresponding *ST_INITENC* state. The next state after the initializations will be the *ST_DECODE_DC_DIFF* state. In order to decode the first binary decision in the *ST_DECODE_DC_DIFF* state, the context is already supplied to the probability estimation unit from the current state. This is also the main difference between the encoding and the decoding states. In the decoding states the corresponding decoded binary decision have to be issued in the previous state in order to pass though the pipeline stages.

ST_DECODE_DC_DIFF In this state, the binary decision issued in the *ST_INITDEC* state is decoded and the next state may be the *ST_DECODE_AC_COEFF* state or the *ST_DECODE_SIGN_OF_V* state depending if the *DC_DIFF* value is zero.

ST_DECODE_AC_COEFF The next states from this state may be the *ST_DECODE_DC_DIFF* in case that the block of 64 DCT coefficients is coded or the state *ST_DECODE_ZERO_AC* if either a sequence of zero DCT coefficients or a non-zero DCT coefficient is decoded.

ST_DECODE_SIGN_OF_V The sign of the DCT coefficient is decoded. The next state is *ST_DECODE_LOG2_SZ*.

ST_DECODE_LOG2_SZ The magnitude category is decoded in this state. The magnitude category is predicted to be as large as possible in order to fully utilize the pipeline. The outcome of the last binary decision, which provides the magnitude category of the DCT coefficient, will cancel the previous three binary decisions, which were issued in the pipeline with the assumption that the maximum magnitude category has not been reached. The next state is *ST_DECODE_SZ_BITS*.

ST_DECODE_SZ_BITS In this state the magnitude bits are decoded. Since no prediction is possible for the magnitude bits or equivalently, they are predicted with 0.5 probability each, the magnitude bits are decoded sequentially.

ST_FLUSH_BITS This state is used to end the encoding process by sending the *final_bits* signal to the arithmetic coding unit and waiting for the last three coded bytes. After that, the encoding process has finished and the corresponding *SYS_ECS* flag in the status register is set and the *CODE* flag is cleared.

Chapter 5

Design Environment and Conclusions

In this chapter, we will shortly present the CAD environment that was utilized in the design of the chip and the possible uses of such a VLSI chip in image compression systems.

5.1 Design flow

As was mentioned in the implementation chapter, the software used was the Cadence Edge Design framework with the SOLO2030 design kit from ES2. The design flow using the specific design kit is shown in Fig. 5.1. All the schematics, except the control units, were designed manually using the standard cell library. The control units were synthesized using the *SIS* [27,28] synthesizing software —the exact procedures will be described in the following paragraphs— and then imported into the Cadence environment. A large number of the soft-cells provided with the ES2 design kit were not adequate for our purposes, since they are ported from discrete component schematics. Also, since we decided to use only synchronous clocked registers, without any asynchronous set/reset signals, many types of registers, counters, etc., from the soft library were redesigned as needed, with various bit-sizes.

As can be seen from the diagram, each schematic was designed, netlist extracted for Verilog[1], simulated and timing analyzed. Fanout and timing violations from the Verilog simulator, which can be shown graphically in the Cadence schematic editor, were eliminated. The design process continued with the final top level schematic, since a hierarchical design methodology was used, being flattened and following that, the standard cells were placed and routed in the Cadence design environment. After the final layout generation, exact parasitics delays of the routed design were extracted and included in the Verilog simulations for accurate timing analysis.

5.1.1 Simulation environment

We used the Verilog [29] simulator for our logic simulations. Verilog is also provided together with the Cadence Edge environment and has the ability to include delays in its logic simulations. The delays are of two kinds. First a constant intrinsic delay is assigned to each gate and then a variable delay, according to the fanout of each gate, is calculated and added to the intrinsic delay. In order for Verilog to include the first type of delay, only the Verilog library of the standard cells has to be provided. However, the fanout delay, which involves calculations that depend on the circuit, is foundry specific and therefore C library routines, which compose

[1] initially we were using the Silos2 simulator, but as the Verilog simulator become available, we switched over to Verilog, since it includes more powerful simulation constructs in its language.

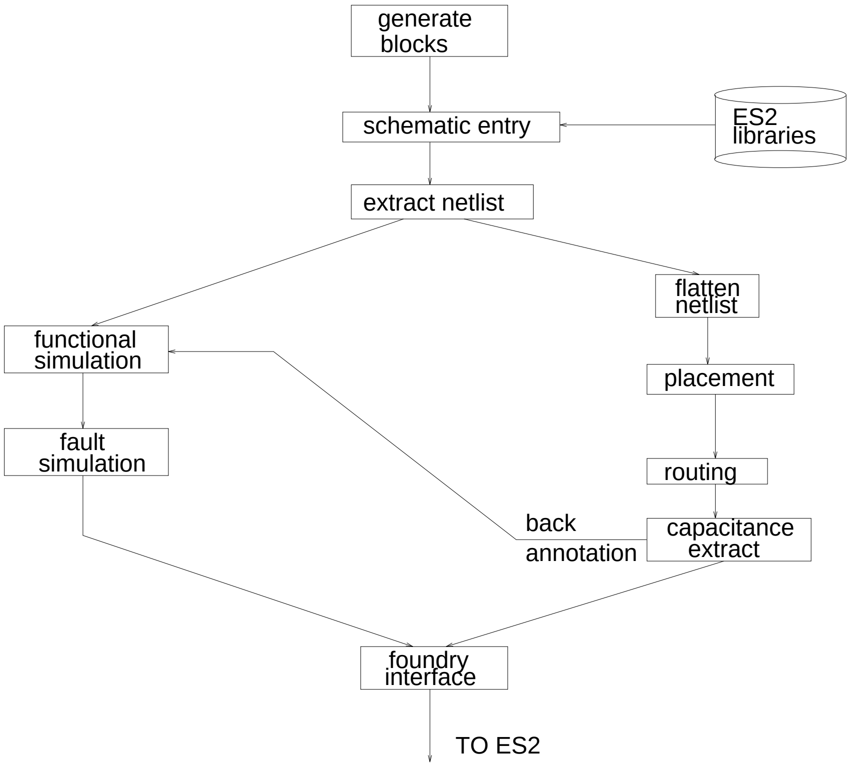


Fig. 5.1: SOLO2030 Design flow

the delay calculator, have to be linked into the Verilog simulator. Also, the Megacells generated with the compilers from the ES2 library make use of such C routines in order to provide the functionality of the regular structures.

5.1.2 Control Unit designs

The control units were designed using the SIS [27,28] synthesis package developed at the University of Berkeley. The respective design flow is shown in Fig. 5.2. The control unit is described in a high level language (BDS), that includes construct like "if then else", "while", "for" and is used to describe combinational circuits. Through bdsyn the BDS description is translated into Berkeley Logic Interchange Format (BLIF) [30] and then read into the SIS environment. Following that, various minimizations like boolean minimizations, common cube extractions, etc.,

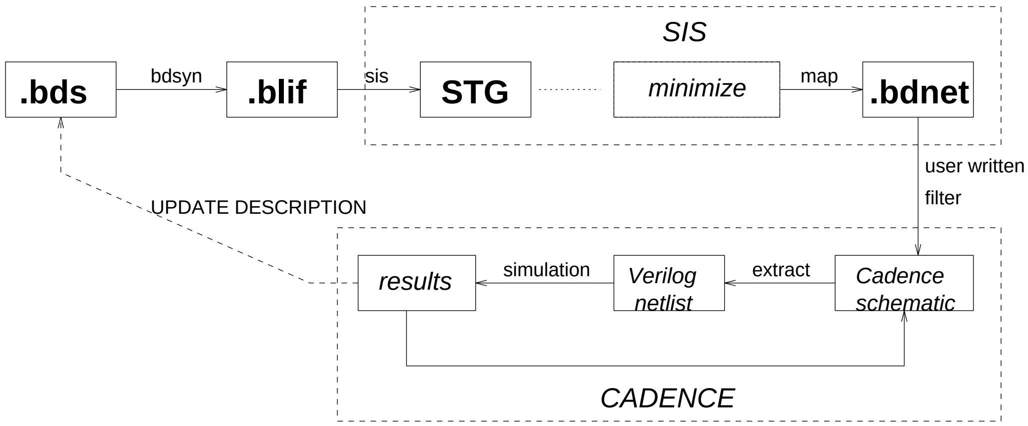


Fig. 5.2: SIS synthesis design flow

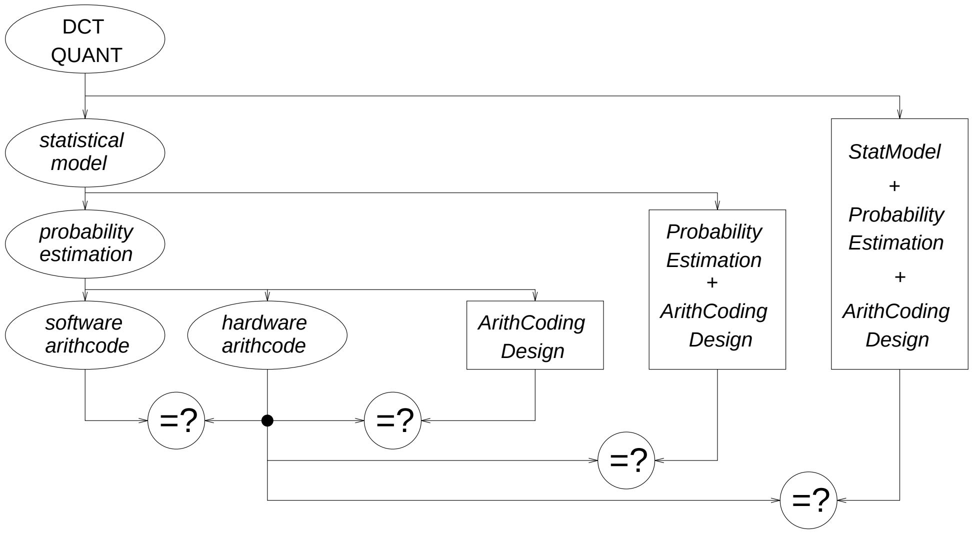


Fig. 5.3: Verification procedure.

are performed, most of which are user controlled. A library file with the standard cells boolean functions and timing characteristics, which we wrote from the ES2 specifications, is supplied to SIS and a mapping of the specified technology is performed. The resulting output, contains the circuit implemented with the ES2 standard cells. Various parameters can be set during the mapping process, and they control the amount of speed versus area tradeoffs performed by SIS. The result is in textual *bdnet* format and general characteristics of the circuit generated, like total area and timing delays, are provided. Since the bdnet file is in textual format we have written a small filter to generate a primitive Cadence Edge schematic from the text, the result of which we used to extract a Verilog netlist and to perform the necessary simulations. The result of the simulations prescribed the corrections that were necessary to be performed to the initial description and after the modifications, the overall process was repeated again, until the circuit performed according to the specifications.

5.1.3 Verification procedure

The procedure used to verify the correct operation of our hardware design is shown graphically in Fig. 5.3. A free available software implementation [2] of the baseline JPEG system with Huffman coding was used to generate the test-vectors for our design for a number of test images. First, arithmetic coding software routines were incorporated into the JPEG baseline system, and following that, the hardware algorithm was implemented in software in order to verify its code-stream against the code-stream produced by the software algorithm. Afterwards, a Verilog behavioral description of the hardware algorithm was written in order to simulate the coding pipeline. The software was modified to generate appropriate test-vectors, as the design of the various units was completed. First, test-vectors for the arithmetic coding unit were generated and the respective output of the circuit was verified against the expected code-stream. The same process was repeated for the probability estimation unit and for the statistical model unit. At the end, DCT-coefficients and code-stream test-vectors generated from the software were verified against the output from our circuit. The I/O-unit was tested independently from the other units and verilog behavioral modules were written that modeled an external system bus circuit and the external DCT-bus circuit. These modules provided our circuit with the necessary test-vectors, generated in software, and also performed the necessary verification. In

[2] Independent JPEG Group's free software, available via anonymous ftp from ftp.uu.net in directory: /graphics/jpeg, file: jpegsrc.v4.tar.Z

addition to the test-vectors generated from real images, artificial images, random DCT coefficients or specific patters, were fed both into the circuit and to the software implementation and the output from the latter was verified against the output from the hardware simulations.

5.2 Chip characteristics

The final placed and routed chip is shown in Fig. 5.4. The die is almost a square of 7.079 mm

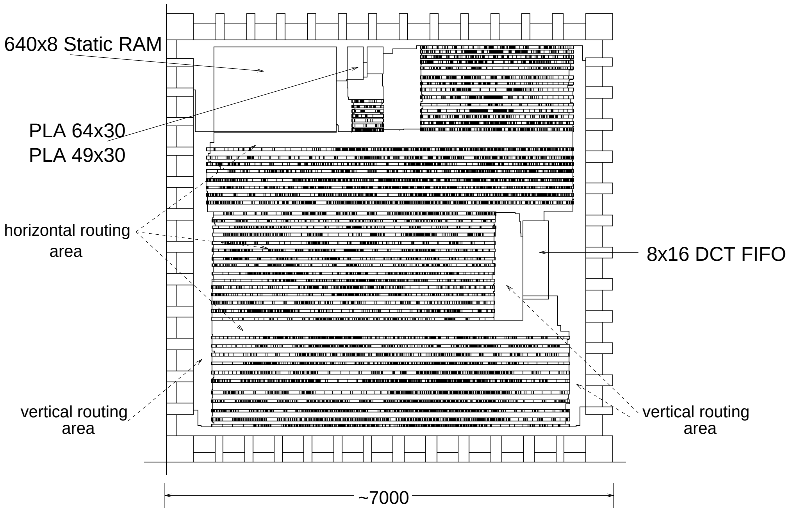


Fig. 5.4: Placed & routed chip.

$\times$ 7.130 mm with an area of 50.47 mm^2. It contains 8192 standard cells, one static RAM, two PLAs and one dual port RAM. The equivalent number of gates is 22257. It was simulated to operate correctly, as described above, for a clock cycle time of approximately 53 ns and for a clock frequency of 18.9 MHz. The simulations included minimum and maximum delays with a clock duty cycle range of 40% to 60%. It was submitted for fabrication trough the EUROCHIP Esprit program to ES2 in the 1.0 μm process.

The RAM and PLAs occupy the upper left part of the die and the FIFO is placed in the middle right position. The system bus interface pads are placed on the left side of the die and the DCT bus interface to the right. The six power and supply pads are positioned at the four corners of the die. Three main standard cell areas were defined for a total of 48 standard cell rows. After the place and route phase has been performed, parasitics capacitances from the nets were extracted and incorporated into loaded Verilog simulations. The estimation of the wire parasitics during the Verilog simulation of the schematics is based mainly on the number of transistors of the cells that are driven by the gate which also drives the net. Each standard cell in the Verilog library contains also a transistor count which is used for such purposes. This estimation is conservative in most of the cases and the actual wire delays will be smaller, after placement and routing is performed. Megacell generated cells contain a small number of routing pins in regard to the area that they occupy and therefore, they are assigned a small number of transistors, which does not correspond to the number of transistors used to implement the circuit. The estimation process fails to predict the wiring load capacitances, in cases where gates drive a large number of other cells, like clock signals and buses with a large number of listeners. Also, if they drive gates in various units which are not placed nearly topological, the wiring load capacitances exceed the output and the input capacitances of the gates and are larger than predicted.

During the initial first place and route iterations, the nets with fanout violations were assigned high routing priorities during the consequent place and route iterations. Critical nets resulting in high capacitance values, even after high priorities were assigned to them, were routed manually in the interactive routing environment of the Cadence Edge routers. Two such routers are provided, a tree based router and an expansion based router. The first is used to route signals with large fanouts, like supply lines and clock signals, and the second is used to route signals in the routing areas between standard cell rows.

According to [23], separate power and ground pads should be provided for the core and the I/O ring. One pair of power and ground pads is needed for every 40 mm^2 of core area and each pair of power and ground pads corresponds to 18–20 I/O cells. Since our chip contains 42 I/O cells (excluding the power and the ground pads), two such pairs were used. Also, standard cell rows which are clocked with a frequency of 20 MHz, can be supplied from only one side with power and ground up to a row length of 4.5 mm. If they are supplied from both sides, the length of the rows can be as large as 7 mm. Since the standard cell rows in our design exceed the 4.5 mm margin, they were supplied manually from both sides. Automatically the tree router supplies the standard cell rows only from one side and therefore, either each row has to be split in the middle or the power and ground nets have to be routed manually. Also, the minimum track length of the power lines, for the given operating frequency, is 20 μm for a 10000 gate design. We used larger tracks of 60 μm, since the gate count of our design is much larger.

5.3 Design Environment Evaluation

In this section we will briefly present our experiences during the design period with the CADENCE Edge software and the respective ES2 design-kit. As was pointed out previously, the design-kit provided by ES2 is mainly targeted for ASIC design environments. The Cadence Edge software on the other hand is a complete design framework for semi and full-custom VLSI design. Two design-kits were provided to us through EUROCHIP. The first is the "comercially" SOLO2030 ASIC design-kit from ES2 and the second is a full-custom design-kit, developed by EUROCHIP for the 1.5 μm process of ES2. Since we used the SOLO2030 design-kit, we will concentrate mainly on this package. The digital standard library provided by this design-kit, is a subset of the library included in the SOLO1400 software, which is a low to medium complexity design package developed by ES2. Also, the soft-library included in the Cadence design-kit contains only a small number of the cells, which are included in the SOLO1400 software. The following remarks can be made about the standard and the soft-cell library.

- Although three different driving strength of inverters are sufficient for the majority of design requirements, as was pointed out in the beginning of the implementation chapter, the inclusion of different kind of driving strength of logic gates in the standard library would result in smaller standard cell areas and less routing requirements. With the present combination of standard cells, inverters or buffers have to be inserted at the outputs of the logic gates, in cases where the fanout limit is surpassed.

- The inclusion of cells with more than one inputs is provided only for multiplexors and registers and only for inputs of four. A much larger variety of such cells, which are designed as soft-cells, would also help in both aspects, area and speed.

- The soft-library, as it is implemented using schematics from the discrete TTL and CMOS components, although it may seem familiar to a hardware designer, does not function always optimal, using the existing standard cell library. Therefore, this soft-library was also eliminated in the last version of the respective design-kit.

- Although the standard cell library contains fall-through latches, their use is limited due to the use of a one phase clocking scheme. Unfortunately, two-phase clock generators, primarily used in CMOS design, are not included in the library and therefore the much larger edge triggered flip-flops have to be used. The design of such two-phase generators from standard cell components is not possible.

5.3.1 Circuit design environment

The inclusion of generators for regular structures like RAMs, ROMs, PLAs, multipliers, etc., is very positive and also the availability of a datapath compiler in the $1.0\ \mu$m helps reducing the amount of area spend for achieving a specific functionality. The design and simulation environment provided, is very limited. Concerning the design tools available to the designer we can notice that:

- the symbols used by the schematic editor are not parametric —the *modulemaker* package is used for such purposes in Cadence Edge, but was not included in the framework available to us by EUROCHIP— and therefore, different symbols for different bit-lengths for registers and for sizes of 1 to 32 bits have to be generated. Although a minor limitation, it is a very annoying one.

- A more important shortcoming of the design software is the absence of the ability to integrate circuits described in textual form, like a Hardware Description Language (HDL) or even in simple boolean equations. Although these functions are provided by large synthesis packages like Synopsis, even the SOLO1400 medium complexity design software from ES2 provides the ability to include combinational circuits, described in the so called MODEL language [20], into existing designs. The manual design of schematics of random combinational logic where the use of a PLA is not affordable could also be performed automatically.

5.3.2 Simulation environment

Concerning the simulation environment, two logic simulators were provided, SILOS and Verilog. Initially, since only the SILOS simulator included the delay calculator for the ES2 library we had to use it until the Verilog delay calculator was also supported. The main limitation of SILOS lies in that it doesn't contain procedural constructs in its simulation language, which makes the generation and verifications of test-vectors very difficult. Verilog, on the other hand, contains such constructs in its language, but it is much slower than SILOS, especially if it has to save the values of each net for the Cadence Edge waveform display environment. In many cases Verilog performed from 10 to 100 times slower then SILOS, which consequently prescribed the use of a limited number of nets that would be saved in each simulation run. The main limitation of the simulation environment can be summed up into:

- The Verilog delay calculator provided was only for the flat netlister of Cadence Edge. This means that all the design must be netlisted in order to be simulated. In small designs this doesn't matter, but as the design gets more complex and is build up hierarchically, the modification of a schematic should not affect the netlist of the other schematics which are not modified. The result using the flat netlister is that the final design of 22000 gates has to be netlisted in its entirety in order to simulate it, even if only one gate has been modified. The other approach is to use a special Verilog netlist for each schematic and as the final cell is build up hierarchically from already netlisted schematics, the resulting netlist time is very small. This would be possible if the delay calculator for the hierarchical netlister were supported. A way in between is to manually split the design into a few large parts, which are netlisted independently and manually combined to perform the simulation. We followed the last approach since the hierarchical netlister was not supported.

- No timing analysis software was supported, although the Veritime and the Timing Analyzer from Cadence were available. The Veritime software has the ability to run both static and dynamic (hybrid) simulations of the circuit and to present timing analyzing information, like longest, shortest paths with minimum and maximum delays, distribution of the paths and slack times with regard to the clock cycles. Unfortunately, Veritime requires a different kind of delay calculator than Verilog, because in Verilog each delay associated with a gate contains values for the six possible transitions (01, 10, 0X, X0, 0Z, Z0), but Veritime uses 18 transitions, each one of the previous six, combined with minimum, typical and maximum delays, because it has to perform timing analysis. It was not possible for us to modify the Verilog delay calculator for this, because the Veritime version available to us [31] did not include the necessary programming interface for such a task. Only after the design was send for fabrication we received an updated version. Therefore, only static and hybrid simulations with the intrinsic delays of the gates have been performed. The delays added due to fanout were not incorporated in such timing analysis.

5.4 Conclusions

We have presented in this work the design of an alternative entropy coder in DCT based JPEG still image compression. The alternative entropy coder is specified in the JPEG specifications, and performs better than the usual Huffman entropy coder. Furthermore, it achieves the maximum entropy compression limit. Since the arithmetic entropy coder is not included in the baseline system, its use is in special purpose applications that implement as a basic function the Huffman coder and also provide the additional capability to code or to trancode according to the arithmetic entropy coder.

The circuit, which performs the arithmetic operations, including the probability estimation state machine but excluding the statistical areas which are application dependent, form a soft-megacell of approximately 2000 standard cells and 5000 gates that can also be used in implementations of the JBIG specifications for lossless compression of bi-level images. Since the circuit is designed using standard cell technology some performance penalty is introduced, but it may use future fabrication processes, from the same foundry and with minor changes the fabrication with a different library is also possible.

Possible extension or future enhancement that may be incorporated are: The elimination of the additional renormalization cycles ($\approx 20\%$) in the arithmetic coding unit by providing an additional path to the input multiplexors of the C and A registers with their values shifted left by 2. This would lengthen the cycle time, but it may prove more efficient, depending on the implementation. The use of a dual port RAM in the place of the static RAM which contains the context storage locations. In the present implementation, two context areas are used. A respective dual-port RAM for one context area occupies about the same silicon area as the two context areas in the present implementation. Therefore, either only one context area may be used in order to remain in the present area occupancy or two context areas can be used with additional $10\,\mathrm{mm}^2$ of silicon area. The use of a dual port RAM would eliminate all the data dependencies concerning stages 3 and 5 of the pipeline. Unfortunately, the dependencies from stages 4 and 5 can only be eliminated when two probability estimation state machines are used. The two state machines can be used to retrieve the probability estimation on the MPS and the LPS path during the cycle, in which in stage 5, the arithmetic coding operation is performed and the new access in stage 4 is the same as that in stage 5. When an update is necessary, the values retrieved from the appropriate (MPS or LPS) alternative state machines can be used. These enhancements would eliminate all the data dependencies in the encoding pipeline, resulting in a encoding rate of 1 bit/cycle. The same rate could also be achieved during decoding when successive binary decisions, which depend on previous decoding operations still in the pipeline, are predicted correctly. When these predictions are wrong a penalty of three additional cycles has to be taken into account.

Appendix A

ES2 Standard Cell Library

The 1.0 μm ES2 Standard Cell Library is comprised from the cells shown in the following two Tables (A.1, A.2). Foreach cell, its name, size, fanin, fanout, typical and maximum delays are given. For cells with more than one input, the fanin, typical and maximum delays are given for the worst cases, that is, for the largest fanin and the maximum rise and fall delays. Detailed delay characteristics of each cell can be found in [23].

cell	height = 47.5 μm	fanin	fanout	rise	rise-fanout	fall	fall-fanout
name	length (μm)	pF/inp.	max. pF	ns	ns/pF	ns	ns/pF
nand2	12.5	0.052	0.690	0.520	3.140	0.450	2.330
nand3	17.2	0.062	0.650	0.740	3.080	0.610	2.830
nand4	20.0	0.061	0.610	0.860	3.100	0.710	3.640
nor2	12.0	0.049	0.420	0.500	4.960	0.620	2.250
nor3	27.1	0.039	0.420	0.500	4.960	0.620	2.250
and2	16.8	0.036	1.120	1.070	2.130	1.320	1.930
and3	21.5	0.045	0.910	1.010	2.570	1.690	1.840
and4	25.0	0.058	1.080	1.020	2.250	1.650	1.820
or2	17.2	0.039	0.910	1.050	2.530	1.190	1.860
or3	21.5	0.048	0.860	0.750	2.680	2.010	2.140
aoi22	19.8	0.053	0.410	0.850	4.660	1.340	4.430
aoi21	19.8	0.053	0.440	0.650	4.570	1.060	5.190
oai22	20.5	0.054	0.260	1.010	5.580	1.280	4.530
oai21	17.4	0.048	0.340	0.730	5.560	0.980	5.020
inv	9.0	0.063	0.910	0.290	2.570	0.350	1.840
inv3	16.8	0.150	3.310	0.270	0.750	0.270	0.700
inv6	24.5	0.344	7.220	0.260	0.340	0.230	0.270
buf39	32.0	0.112	6.630	0.710	0.380	0.890	0.310
mux21	29.2	0.070	0.870	1.570	2.630	2.060	1.910
mux41	62.4	0.127	0.780	2.740	2.940	2.910	2.600
xor2	27.5	0.052	0.960	1.710	2.440	2.090	1.83
xnor2	27.5	0.046	0.990	1.750	2.370	2.400	1.900

Table A.1: ES2 $1.0\,\mu$m combinational standard cell library

The combinational part of the standard cell library also includes tristate inverters and buffers (NTRI, NTRI3, NTRI6, TRI3, TRI6, 4TRI3) and adders (ADD2, FADD1, FADD2). The next table presents the basic fall-through latches and edge triggered flip flops. In addition to the cells shown in Table A.2 the library includes edge-triggered scan flip-flops (SFF, SFFR, SFFS, SFFRS)

cell name	height = 47.5 μm length (μm)	fanin pF/inp.	fanout max. pF	rise ns	rise-fanout ns/pF	fall ns	fall-fanout ns/pF
LAT	49.5	0.043	0.920	2.530	2.510	2.070	1.910
LATR	55.0	0.034	0.970	3.350	2.330	4.120	1.600
LATS	56.2	0.034	0.930	2.800	2.510	4.270	1.600
LATRS	53.1	0.036	1.020	4.070	2.120	4.500	1.370
DFF	57.8	0.036	1.000	3.170	2.030	3.170	1.520
DFFR	69.0	0.063	0.990	2.870	2.330	2.900	1.460
DFFS	62.5	0.053	0.970	3.330	2.090	3.180	1.260
DFFRS	74.5	0.067	0.980	2.910	2.330	3.350	1.450

Table A.2: ES2 $1.0\,\mu$m storage standard cell library

and a four bit edge-triggered register (REG4).

Appendix B

Programmers Manual

In this chapter we will present the necessary operations that have to be performed in order to test the internal static RAM and the two PLAs for faults and also the procedures that have to be carried out prior to the coding process. A sample example for each coding mode (encoding/decoding) will be given that should work without any modifications if an appropriate external circuit interfaces correctly to the VLSI chip. The only procedures, which are assumed to be supplied by the interface circuit, are the procedures that read and write all the internal system-bus registers and the 16-bit DCT coefficients from the DCT-bus interface. The examples cover the following procedures during normal and test mode operation.

1. reading and writing an internal system-register.
2. reading and writing an DCT-coefficient from the internal DCT-fifo.
3. setting up encoding mode.
4. setting up decoding mode.
5. reading and writing the internal context area RAM.
6. testing the internal context RAM for faults.
7. JTAG EXTEST instruction.
8. JTAG RUNBIST instruction to test the two internal PLAs.

B.1 Definitions of Procedures and Constants

The internal addressable registers were described in Sec. 3.3.1. Fig. 3.4 contains a layout of these registers. They are repeated here together with their assigned symbolic names. A `I_` in front of the name signifies a register which is used in normal operation only for input. Respectively `O_` is used for output registers and `IO_` for registers that are used both ways. All registers can also be read from the system bus. Active low signals are identified with a star (*) suffix. In the third column, each of the bits of the two status registers are given with their symbolic names. A respective flag, which is used to generate the 8-bit values that have to be written into the status registers, is given in the last column. A value that will be written into one of the status registers therefore has only to perform a logic *or* of the corresponding flags. Some of the register names are assigned to the same internal address. This is done in order to differentiate the use of the specific register in normal mode from test mode.

REGISTERS	**Addr.**	IO_ST0	**status flags**	**M_XXX**
IO_ST0	0	I_ENC	0	0x01
IO_ST1	1	I_DEC	1	0x02
I_DIN	2	I_CODE	2	0x04
O_DOUT	3	O_SYS_WRRDY*	3	0x08
I_EXT_ST_RAM_A	4	O_SYS_RDRDY*	4	0x10
I_MCU_RES_INT_L	4			
I_MCU_RES_INT_H	5	O_DCT_WRRDY*	5	0x20
I_NUM_MCU_L	6	O_DCT_RDRDY*	6	0x40
I_NUM_MCU_H	7	O_SYS_ECS*	7	0x80
I_MDU_MCU_POS	8	IO_ST1	**status flags**	**M_XXX**
I_MCU_MEMBER	8	I_DCT_SIGN_EX	0	0x01
I_MCU_MEMBER+1	9	I_EN_RES_INT	1	0x02
I_MCU_MEMBER+2	10	I_EN_MDU_MCU	2	0x04
I_DAC_TBL_IDX	11	I_EXT_ST_RAM_CLR	3	0x80
I_DC_TBL	12	I_EXT_ST_RAM_LD	4	0x10
I_DC_TBL+1	13	I_EXT_ST_RAM_WE	5	0x20
I_AC_TBL	14	I_EXT_ST_RAM_A8	6	0x40
I_AC_TBL+1	15	I_EXT_ST_RAM_A9	7	0x80

Table B.1: Internal addressable registers and the corresponding status flags for the status registers IO_ST0 and IO_ST1

B.2 Setting up encode mode

The following read and write operations — given as verilog tasks — have to be performed prior to the encoding process. st0_reg and st1_reg are external temporary variables that contain the status bits to be written into the internal status registers. MS_XXX is the mask that sets the bit XXX in the status registers. Their values are shown in Table B.1. The procedures write_byte_sys and read_byte_sys are the write and read procedures to the system bus. The other two variables which are necessary are:

opt_mdu_mcu: The number of MDU in each MCU.

opt_num_mcu: The number of MCU in the image.

It is obvious that *opt_mdu_mcu* $*$ *opt_num_mcu* should sum up to the number of MDU in the image. The operations in the following task have to be performed prior to the supply of the DCT coefficients. The last write into status register 0 enables encoding. Since practically a YCbCr *luminance-chrominance* color space is used in most of the cases, and furthermore the chrominance components are subsampled with respect to the luminance component, a MCU is comprised from six (6) MDUs, of which the first four (4) belong to the *luminance* component Y, the fifth (5) belongs to the *chrominance* component Cb and the last (6) belongs to the *chrominance* component Cr. In most cases the *luminance* component uses one of the context areas and the two *chrominance* components use the other. The *MCU_MEMBER* parameter specifies the component (1 to 4) to which the 10 possible MDUs belong. Therefore, a value of b10010000 is written into location *MCU_MEMBER+1* specifying that MDU 2 and 3 belong to the component b00 (Y), MDU 4 to component b01 (Cb) and MDU 5 to component b10 (Cr). The component, to which MDUs 0 and 1 belong, is specified by the 4 high order bits of the register *MCU_MEMBER*, which is zero by default. Each of the 1 to 4 possible components is assigned to a context area. Since in our implementation we decided to use only two such areas, the *luminance* component uses the first context area and the *chrominance* components the other two. Therefore, a value of b00 should be written into bit positions 0 to 1 of register *DAC_TBL_IDX*, a value of b01 into bit positions 3 to 2 and a value of b01 into bit positions 5 to 4. Actually only the first bit of these two bit values are used in our implementation and the other is ignored. The write operations into the registers *DC_TBL* and *AC_TBL* define the L, U and Kx values, which are by default 0, 1 and 5 respectively. Since these values can differ for each of the two context areas, two such writes are necessary. The corresponding data structure is can be found in Fig. 3.9.

```
`define DEFAULT_MASK_ST1 (`MS_DCT_SIGN_EXT | `MS_EN_MDU_MCU)
`define DEFAULT_MASK_ST0 `MS_ENC
`define BYTE_SZ 8
`define WORD_SZ 32

task set_encode_mode;
reg [`BYTE_SZ-1:0] byte;
begin
    st0_reg = `DEFAULT_MASK_ST0 | `MS_ENC;
    st1_reg = `DEFAULT_MASK_ST1 | `MS_EXT_ST_RAM_CLR;
    write_byte_sys(`ST0, st0_reg);
    write_byte_sys(`ST1, st1_reg);
    read_byte_sys(`ST0, st0_reg);
    write_byte_sys(`MDU_MCU_POS, opt_mdu_mcu);
    write_byte_sys(`NUM_MCU_L, opt_num_mcu&16'h00FF);
    write_byte_sys(`NUM_MCU_H, (opt_num_mcu&16'hFF00) >> 8);
    write_byte_sys(`MCU_RES_INT_L, opt_mcu_res_int&16'h00FF);
    write_byte_sys(`MCU_RES_INT_H, (opt_mcu_res_int&16'hFF00) >> 8);
    write_byte_sys(`DAC_TBL_IDX, 8'b00111100);
    write_byte_sys(`DC_TBL, 8'h10);
    write_byte_sys(`DC_TBL+1, 8'h10);
    write_byte_sys(`AC_TBL, 8'h05);
    write_byte_sys(`AC_TBL+1, 8'h05);
    write_byte_sys(`MCU_MEMBER+1, 'b10010000);
    st0_reg = st0_reg | `MS_CODE;
    write_byte_sys(`ST0, st0_reg);
end
endtask
```

B.3 Normal encoding operation

During the encoding operation, DCT coefficients are supplied to the DCT bus and the code stream is read from the system bus. The system bus busy waits for the data available bit (*SYS_RDRDY*) to be set in status register 0. When it is set, one byte of coded data is read from the *DOUT* register. This is accomplished in the main while loop, in which also the *SYS_ECS* bit is checked. When this bit is set the encoding process has finished. The other for-loop (*sys_cycles_to_wait*) is used only for inserting a delay between the point in time in which the ready flag is seen to be active and the reading of the coded byte.

```
task encode_sys;
input [31:0] bytes;
begin
    forever begin
        read_byte_sys(`ST0, st0_reg);
        if (st0_reg[`SYS_ECS]) begin
            £display("ECS received");
            disable encode_sys;
        end
        while (st0_reg[`SYS_RDRDY_] !== 0) begin
```

```
            read_byte_sys(‘ST0, st0_reg);
            if (st0_reg[‘SYS_ECS]) begin
                £display("ECS1 received");
                disable encode_sys;
            end
        end
        for (i=0; i<sys_cycles_to_wait; i=i+1) begin
            @(negedge clk) ;
        end
        read_byte_sys(‘DOUT, SysBus_reg);
        verify_sys(SysBus_reg);
    end
end
endtask
```

B.4 Setting up decode mode

The read and write operations before the decoding process are exactly the same, except that this time the corresponding decoding flag is set and the encoding flag is cleared.

```
task set_decode_mode;
begin
    st0_reg = ‘DEFAULT_MASK_ST0 & ~‘MS_ENC;
    st0_reg = st0_reg | ‘MS_DEC;
    st1_reg = ‘DEFAULT_MASK_ST1 | ‘MS_EXT_ST_RAM_CLR;
    write_byte_sys(‘ST0, st0_reg);
    write_byte_sys(‘ST1, st1_reg);
    read_byte_sys(‘ST0, st0_reg);
    write_byte_sys(‘MDU_MCU_POS, opt_mdu_mcu);
    write_byte_sys(‘NUM_MCU_L, opt_num_mcu&16'h00FF);
    write_byte_sys(‘NUM_MCU_H, (opt_num_mcu&16'hFF00) >> 8);
    write_byte_sys(‘MCU_RES_INT_L, opt_mcu_res_int&16'h00FF);
    write_byte_sys(‘MCU_RES_INT_H, (opt_mcu_res_int&16'hFF00) >> 8);
    write_byte_sys(‘DAC_TBL_IDX, 8'b00111100);
    write_byte_sys(‘DC_TBL, 8'h10);
    write_byte_sys(‘DC_TBL+1, 8'h10);
    write_byte_sys(‘AC_TBL, 8'h05);
    write_byte_sys(‘AC_TBL+1, 8'h05);
    write_byte_sys(‘MCU_MEMBER+1, 'b10010000);
    st0_reg = st0_reg | ‘MS_CODE;
    write_byte_sys(‘ST0, st0_reg);
end
endtask
```

B.5 Normal decoding operation

The main while loop during decoding is the same as during encoding. The flag which signifies that another byte has to be written into the *DIN* register is *SYS_WRRDY*. The second for-loop is used to supply additional zero bytes after the coded bytes have been absorbed. The final signal which signifies that the decoding process has finished (*decoding_finished*) comes from the external DCT unit when the image has been decoded.

```
task decode_sys;
input [31:0] bytes;
begin
    while (curr_byte < bytes) begin
        read_byte_sys(‘ST0, st0_reg);
        while (st0_reg[‘SYS_WRRDY_] !== 0) begin
```

```
            read_byte_sys(`ST0, st0_reg);
        end
        £display("\nTesting sys-data stall for #%0d cycles\n",
            sys_cycles_to_wait);
        for (i=0; i<sys_cycles_to_wait; i=i+1) begin
            @(negedge clk) ;
        end
        £display("\nSys-data stalled for #%0d cycles\n", sys_cycles_to_wait);
        write_byte_sys(`DIN, cdata[curr_byte]);
        £display("%m: writing byte #%0d of %0d = %0h", curr_byte, bytes,
            cdata[curr_byte]);
        curr_byte = curr_byte+1;
    end
    forever begin
        if (decoding_finished)
            disable decode_sys;
        read_byte_sys(`ST0, st0_reg);
        while (st0_reg[`SYS_WRRDY_] !== 0) begin
            read_byte_sys(`ST0, st0_reg);
            if (decoding_finished)
                disable decode_sys;
        end
        write_byte_sys(`DIN, 'h00);
        curr_byte = curr_byte+1;
    end
end
endtask
```

B.6 Internal static RAM test for faults

The internal test for faults is as follows: A RAM with n locations has an address space from 0 to $2^w - 1$ where $w^{w-1} < n \leq 2^w$. The test vectors that should be applied to the RAM, which are also the test vectors generated by the BIST, are [25]:

1. Let $a = 2^w$. for each address i from 0 to $a - 1$ a 0 is written into location i.

2. for each address i from 0 to $a - 1$ if $i \leq n - 1$ the previous written 0 is read from location i, a word with all bits set is written into location i and immediately read from location i. If $n \leq i \leq a - 1$ a word with all bits set is written into location i.

3. for each address i from $a - 1$ to 0, when $i \leq n - 1$ a word with all bits set is read from location i, a 0 is written into location i and a 0 is read from location i. When $i > n - 1$ a 0 is written into location i.

4. for each address i from $a - 1$ to 0, when $i \leq n - 1$ a 0 is read from location i.

The above sequence is used to test the ability to write both a 0 and a word with all bits set into each position in the RAM and also to test the transitions of each bit in the words from 0 to 1 and from 1 to 0. Since the internal static RAM uses an address of 10 bits the following tasks are used to read and write the internal static RAM, in one location, and are utilized to generate the above test sequence.

```
task wr_ext_st_ram;
input [7:0] addr;
input [7:0] data;
begin
    read_byte_sys(`ST0, st0_reg);
    while (st0_reg[`SYS_WRRDY_] !== 0)
```

```
        read_byte_sys(`ST0, st0_reg);
    write_byte_sys(`EXT_ST_RAM_A, addr);
    write_byte_sys(`DIN, data);
end
endtask

task rd_ext_st_ram;
input [7:0] addr;
output [7:0] data;
begin
    read_byte_sys(`ST0, st0_reg);
    while (st0_reg[`SYS_WRRDY_] !== 0)
        read_byte_sys(`ST0, st0_reg);
    write_byte_sys(`EXT_ST_RAM_A, addr);

    read_byte_sys(`ST0, st0_reg);
    while (st0_reg[`SYS_RDRDY_] !== 0)
        read_byte_sys(`ST0, st0_reg);
    read_byte_sys(`DOUT, data);
end
endtask

task set_ext_st_ram_a98;
input [9:0] addr;
begin
    if (addr == 255) begin
        st1_reg = st1_reg | `MS_EXT_ST_RAM_A8;
        write_byte_sys(`ST1, st1_reg);
    end else if (addr == 511) begin
        st1_reg = st1_reg & ~`MS_EXT_ST_RAM_A8;
        st1_reg = st1_reg | `MS_EXT_ST_RAM_A9;
        write_byte_sys(`ST1, st1_reg);
    end else if (addr == 767) begin
        st1_reg = st1_reg | `MS_EXT_ST_RAM_A8;
        write_byte_sys(`ST1, st1_reg);
    end else if (addr == 1023) begin
        st1_reg = st1_reg & ~`MS_EXT_ST_RAM_A8 & ~`MS_EXT_ST_RAM_A9;
        write_byte_sys(`ST1, st1_reg);
    end
end
endtask
```

B.7 JTAG controlled testing procedures

The VLSI chip is JTAG (*IEEE standard 1149.1*) compliant, which means that a JTAG controller is included and four primary pins are dedicated for it. Except for the boundary scan register, only one additional internal data register was implemented. The internal data register with address 0 contains the 32-bit signature generated by the internal BIST circuit of the two PLAs. Detailed description of the JTAG standard, the JTAG FSM controller can be found in the references. Some information with specific implementation details can also be found in sec. 4.6. Description of the BIST circuit and the resulting 32-bit signature for correct operation can be found in sec. 4.3.3.

The test procedures use the following utility task, which changes the current state depending on the value of TMS. The JTAG controller is designed in such a way, in which only two transitions are possible from each state. In many states, one of the transitions is the current state. The following task is used to generate such transitions. The task *Next_JTAG_controller_state* is actually a set of such similar tasks, which in place of *JTAG_controller_state* contain the actual name of the current state, as shown in the following listing.

```
`define JTAGCLK_CYCLE     10000 // in 0.1ns units
`define JTAGCLK_DUTY      0.5

task Next_JTAG_controller_state;
input x;
begin
    @(posedge TCLK) #(`JTAGCLK_CYCLE*0.25) TMS = x;
end
endtask

task Next_Test_Logic_Reset;
task Next_Run_Test_Idle;
task Next_Select_DR_Scan;
task Next_Select_IR_Scan;
task Next_Capture_IR;
task Next_Exit1_IR;
task Next_Update_IR;
task Next_Capture_DR;
task Next_Shift_DR;
task Next_Exit1_DR;
task Next_Update_DR;
```

The BIST for the internal PLAs is performed through the included JTAG controller which provides the necessary internal control signals. For this purpose, the JTAG controller is first loaded with the RUNBIST instruction and the JTAG FSM is brought to the state `Run/Test-Idle`, in which it remains for the rest of the BIST. The duration of the BIST sequence for the specific internal PLAs is set to 280 cycles. This is the minimum required number of clock cycles. Additional cycles can be inserted as necessary. They are ignored by the internal circuit and do not alter the signature value. After that period, the signature register is selected and its contents is shifted out through the TDO pin. The 32-bit value contains the two 16-bit signatures of the internal PLAs and should match the following values:

megacell	signature
PLA 64 × 32	1010110011110000
PLA 49 × 32	1101010011010101

```
`define JTAG_RUNBIST 4

task jtag_runbist;
output [31:0] signatures;
integer i;
begin
    £display("STARTING JTAG-RUNBIST SEQUENCE\n");
    display_jtag_state = 1;
    TRST = 1;
    @(posedge TCLK) #(`JTAGCLK_CYCLE*0.25) TMS = 0;

    jtag_load_IR(`JTAG_RUNBIST);
    for (i=0; i<280; i=i+1) begin
        @(posedge TCLK) #(`JTAGCLK_CYCLE*0.25) TMS = 0;
    end
    Next_Run_Test_Idle(1);
    Next_Select_DR_Scan(0);
    Next_Capture_DR(0);
    for (i=0; i<31; i=i+1) begin
        Next_Shift_DR(0);
        signatures[i] = TDO;
    end
end
```

```
endtask
```

B.8 Summary of test procedures

A simple test sequence is shown in the following task. First the internal registers are written and read with a specific and variable value which test the capability of the system bus. Then the internal static RAM is written and read with a specific value that tests the capability of the global internal FSM to write and read the specified RAM. A more extensive test of the internal RAM for faults is followed as was described previously. After that a BIST is initiated to test the internal PLAs and then specific encoding and decoding procedures are performed. These encoding and decoding procedures are interleaved in order to test the correct termination and initialization between the coding periods.

```
task test_procedures;
begin
    test_registers;
    test_ext_st_ram;
    test_fault_ext_st_ram;
    test_jtag;
    test_encode; test_encode;
    test_decode; test_decode;
    test_encode; test_decode; test_encode; test_decode;
end
endtask
```

Bibliography

[1] William B. Pennebaker and Joan L. Mitchell, *JPEG Still Image Data Compression*, Van Nostrand Reinhold, New York, 1993.

[2] K. R. Rao and P. Yip, *Discrete Cosine Transform*, Academic Press, New York, 1990.

[3] E. Feig and E. Linzer, "Discrete Cosine Transform Algorithms for Image Data Compression", in *Proceedings Electronic Imaging '90 East*. Boston, MA, Oct. 1990, pp. 84–87.

[4] A. Ligtenberg and M. Vetterli, "A discrete Fourier-cosine transform chip", *IEEE Journal on Selected Areas in Communication*, vol. 49, no. 4, pp. 49–61, Jan. 1986.

[5] R. W. G. Hunt, *Measuring Colour*, Halsted Press, New York, 1987.

[6] F. I. Van Nes and M. A. Bouman, "Spatial Modulation Transfer in the Human Eye", *Journal of the Opt. Soc. of Am.*, vol. 57, no. 3, pp. 401–406, Mar. 1967.

[7] Jr. Glen G. Langdon, "An introduction to Arithmetic Coding", *IBM Journal Research Development*, vol. 28, no. 2, pp. 135–149, March 1984.

[8] I. H. Witten, Radford M. Neal, and John G. Cleary, "Arithmetic coding for Data compression", *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, June 1987.

[9] W. B. Pennebaker, J. L. Mitchell, Jr. G. G. Langdon, and R. B. Arps, "An overview of the basic principles of the Q-Coder adaptive", *IBM Journal Research Development*, vol. 32, no. 6, pp. 717–726, November 1988.

[10] W. B. Pennebaker and J. L. Mitchell, "Probability estimation for the Q-coder", *IBM Journal Research Development*, vol. 32, no. 6, pp. 737–752, November 1988.

[11] J. L. Mitchell and W. B. Pennebaker, "Software implementations of the Q-coder", *IBM Journal Research Development*, vol. 32, no. 6, pp. 753–774, November 1988.

[12] R. B. Arps, T. K. Truong, D. J. Lu, R. C. Pasco, and T. D. Friedman, "A multi-purpose VLSI chip for adaptive data compression of bilevel images", *IBM Journal Research Development*, vol. 32, no. 6, pp. 775–795, November 1988.

[13] CD 11544 ISO/IEC JTC1/SC2/WG9, *Progressive Bi-level Image Compression, Revision 4.1*, Sept. 1991.

[14] Stephen J. Urban, "Review of standards for electronic imaging for fascimile systems", *Journal of Electronic Imaging*, vol. 1, no. 1, pp. 5–21, Jan. 1992.

[15] International Standard DIS 10918-1, *Digital Compression and Coding of Continuous-Tone Still Images. Part 1: Requirements and Guidelines*.

[16] International Standard DIS 10918-2, *Digital Compression and Coding of continuous-tone still Images, Part 2: Compliance Testing*.

[17] IEEE Standard 1149.1, *Test access port and boundary scan architecture*, 1990.

[18] C.M. MAUNDER and R.E. TULLOSS, "An introduction to the Boundary Scan Technique. ANSI/IEEE Std 1149.1", *Journal of Electronic Testing, Theory and Applications.*, vol. 2, pp. 27–42, March 1991.

[19] Colin M. Maunder and Rodham E. Tulloss, Eds., *The Test Access Port and Boundary Scan Architecture*, IEEE Computer Society Press, 1990.

[20] European Silicon Structures (ES2), Berkshire, UK, *SOLO1400 Design Manual*, Jan. 1990.

[21] European Silicon Structures (ES2), Berkshire, UK, *SOLO2030 Users Manual*, 2 edition, Aug. 1990.

[22] John L. Hennessy and David A Patterson, *Computer Architecture, A Quantitative Approach*, chapter A, Morgan Kaufmann Publishers, Palo Alto, CA, 1992.

[23] European Silicon Structures (ES2), Rousset, France, *ES2 ECPD10 Library Databook*, Mar. 1993.

[24] European Silicon Structures (ES2), Rousset, France, *ES2 Engineering Application Notes*, June 1992.

[25] ES2 (European Silicon Structure), Rousset, France, *ES2 Asic Design Guidelines*, June 1992.

[26] L. Glasser and D. Dobberpuhl, *The Design & Analysis of VLSI Circuits*, chapter 6, pp. 360–365, Addison-Wesley, 1985.

[27] Russell N. Segal, *BDSYN Users Manual*, Unversity of California, Berkeley, March 1990.

[28] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert L. Brayton, and Alberto Sangiovanni-Vincentelli, *SIS: A system for Sequential Circuit Synthesis*, University of California, Berkeley, May 1992.

[29] Gateway Design Automation Corporation, Lowell, MA, *VERILOG-XL*, Sept. 1989.

[30] *Berkeley Logic Interchange Format*, University of California, Berkeley, May 1992.

[31] Gateway Design Automation Corporation, Lowell, MA, *VERITIME Version 1.1a4*, Sept. 1989.