SIMULATION OF DIGICIPHER™, AN HDTV SYSTEM PROPOSAL

A THESIS
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Levent Onural
November 1991

SIMULATION OF DIGICIPHER^{TM}, AN HDTV SYSTEM PROPOSAL

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL AND

ELECTRONICS ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCES

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Levent Öktem

November 1991

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Levent Onural(Principal Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Erdal Arıkan

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Enis Çetin

Approved for the Institute of Engineering and Sciences:

Prof. Dr. Mehmet Baray
Director of Institute of Engineering and Sciences

ABSTRACT

SIMULATION OF DIGICIPHER^{TM}, AN HDTV SYSTEM PROPOSAL

Levent Öktem
M.S. in Electrical and Electronics Engineering
Supervisor: Assoc. Prof. Dr. Levent Onural
November 1991

In this thesis, the digital video encoder–decoder parts of an American HDTV system proposal, DigiCipher^{TM} is simulated in an image sequencer, based on the system description sheets. Numerical and subjective performances are tested, by observing and making calculations on the decoder outputs of the system simulation. The performance tests show that the image quality does not have HDTV quality. Considering the very good picture quality in the demonstrations of the designer company (General Instruments), it is suspected that the description sheets do not mention all of the data compression methods used in the system.

Keywords : HDTV, Data Compression, Digital Video Encoder–Decoder

ÖZET

YÜKSEK TANIMLAMALI TELEVİZYON SİSTEM ÖNERİLERİNDEN DIGICIPHER'IN SİMÜLASYONU

Levent Öktem
Elektrik ve Elektronik Mühendisliği Bölümü Yüksek Lisans
Tez Yöneticisi: Doç. Dr. Levent Onural
Kasım 1991

Bu çalışmada, Amerikan yüksek tanımlamalı televizyon (YTTV) sistem önerilerinden DigiCipher^{TM}'in sayısal video kodlayıcı– kod çözücü bölümünün bilgisayar simülasyonu, sistem hakkında tasarımcı firma tarafından çıkarılan tanımlama raporuna dayanılarak yapılmıştır. Kod çözücü çıktıları üzerinde gözlem ve hesaplamalar yoluyla nümerik ve öznel performans testleri uygulanmıştır. Performans testleri, görüntü kalitesinin YTTV kalitesinde olmadığını göstermiştir. Tasarımcı firmanın (General Instruments) yaptığı demonstrasyonlardaki görüntü kalitesinin çok iyi olduğu gözönüne alındığında, tanımlama raporunda sistemde kullanılan tüm bilgi sıkıştırma yöntemlerinden bahsedilmediği kuşkusu uyanmaktadır.

Anahtar kelimeler : YTTV, Video kodlayıcı-kod çözücü, Bilgi sıkıştırma.

ACKNOWLEDGMENT

I would like to thank to Assoc. Prof. Levent Onural for giving me the opportunity to study at Tampere University of Technology, Finland, for one year, where I made this study; and for guiding me in the stage of writing the thesis.

I am indebted to Prof. Yrjö Neuvo for his supervision of my studies in Finland.

I want to express my special thanks to Vesa Lunden and Roy Mickos for their helps in different stages of my work;and to Mehmet Gencer, with whom I started this study.

I would also like to thank to F. Levent Değertekin, to Zafer Gedik, to Satılmış Topçu and to Özlem Albayrak for their helps in the stage of typing this thesis.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

1.1 HDTV

HDTV is a new TV standard which has much higher resolution than the current systems (roughly twice more horizontal and vertical resolution).

It has a wide-screen aspect ratio, 16:9, where the aspect ratio for the conventional systems is 4:3.

In conventional systems, the viewing distance is about seven times the picture height. From closer distances, the patterning due to limited resolution becomes visible. In HDTV, since the resolution is higher, a closer viewing distance, about three times the picture height, is allowed. This corresponds to a horizontal viewing angle of 30° , where for the conventional systems this angle is 10°. This is depicted in Figure 1.1.

The higher quality image is accompanied by new audio capabilities. Sound is digital, and its quality is comparable with CD sound quality. Multichannel and surround sound capabilities are also available.

The video bandwidth of an uncompressed HDTV signal is almost four times as much as a conventional color TV signal. For compatibility with the bandwidth of already allocated channels, the HDTV signal must be intensively compressed by advanced signal processing methods. Since the video data is the one which takes most of the bandwidth, the greatest efforts of compression are focussed on the video data ([1] , [2] and [3]).

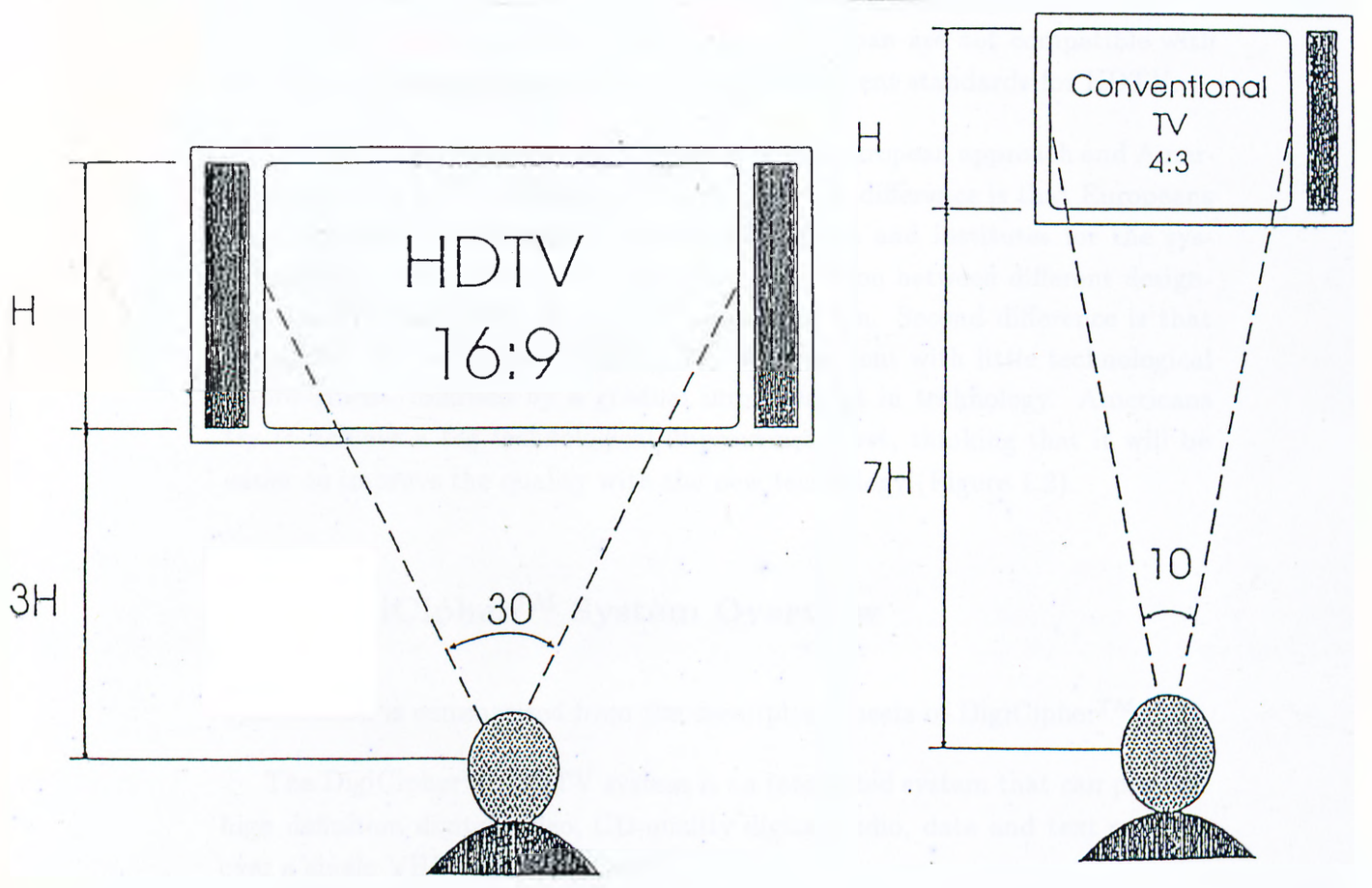


Figure 1.1: Comparison of HDTV and conventional TV

1.2 HDTV Efforts in USA, Japan and Europe

There has been big efforts in USA, Japan and Europe to develop an HDTV standard. Both in Europe and in the United States, the change into HDTV is planned to be gradual. When the incompatibility of the Japanese system was realized in the middle 1980's, the Europeans and the Americans started to develop their own standards. One of the important objectives of developing a new standard is the compatibility with already existing systems. Since the already existing systems in Europe, USA and Japan are not compatible with each other, it seems that there will be three different standards for HDTV.

There are two fundamental differences in the European approach and American approach to the HDTV system design: First difference is that Europeans have organized a cooperation among many firms and institutes for the system design, where Americans preferred competition between different designers, among which they will choose the best design. Second difference is that Europeans try to achieve a big quality improvement with little technological improvement, followed by a gradual improvement in technology. Americans try to achieve a big technological improvement first, thinking that it will be easier to improve the quality with the new technology (Figure 1.2).

1.3 DigiCipher^{TM} System Overview

This section is summarized from the description sheets of DigiCipher^{TM} ([4])

The DigiCipher^{TM} HDTV system is an integrated system that can provide high definition digital video, CD-quality digital audio, data and text services over a single VHF or UHF channel.

Figure 1.3 shows the overall system block diagram. At the HDTV station, the encoder accepts one high definition video and four audio signals and transmits one 16-QAM modulated data stream.. The control computer can supply program related information such as program name, etc. At consumer's home, The DigiCipher^{TM} HDTV receiver receives the 16-QAM data stream and provides video, audio, data, and text to the subscriber. On screen display can be used to display program related information.

Figure 1.4 shows the block diagram of the encoder. The digital video encoder accepts YUV inputs with 16:9 aspect ratio and 1050-line interlaced

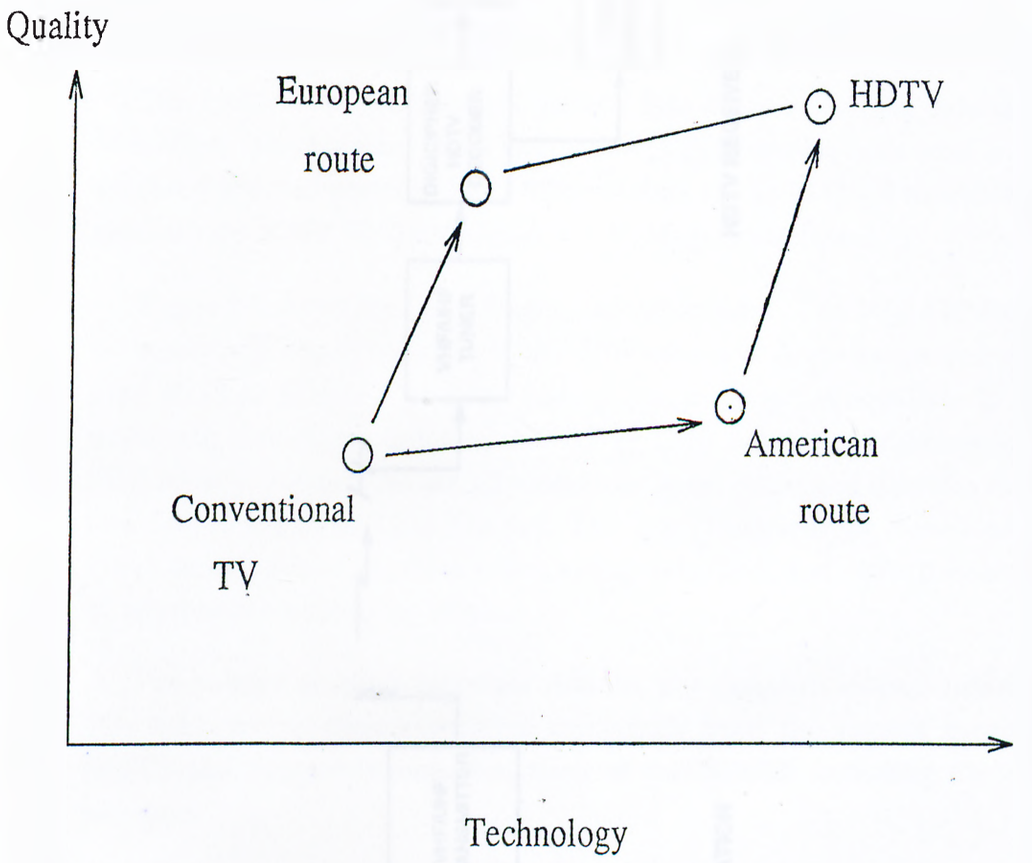


Figure 1.2: American and European routes to HDTV

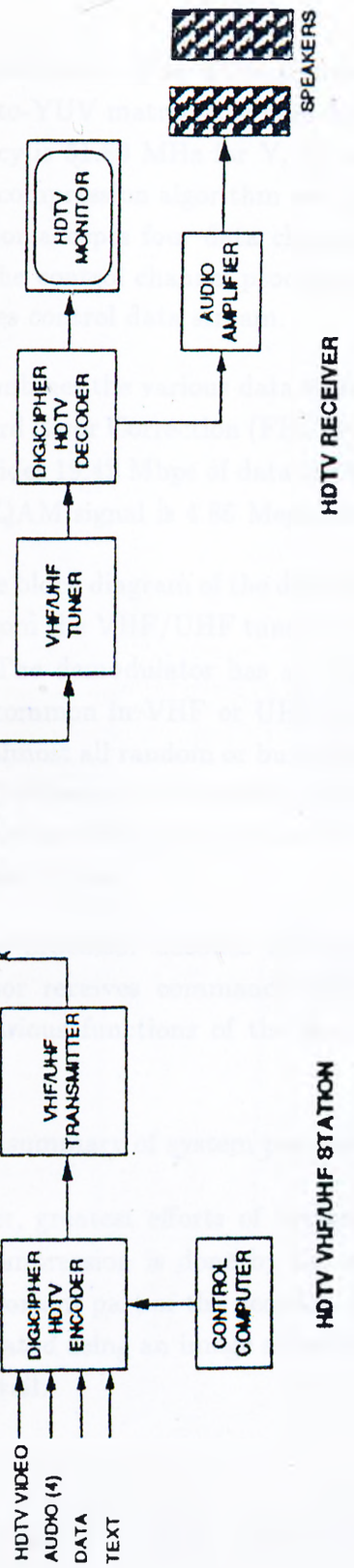


Figure 1.3: Overall System Block Diagram

(1050/2:1) at 59.94 field rate. The YUV signals are obtained from analog RGB inputs by RGB-to-YUV matrix, low pass filtering, and A/D conversion. The sampling frequency is 51.80 MHz for Y, U, and V. The digital video encoder implements the compression algorithm and generates video data stream. The data/text processor accepts four data channels at 9600 baud and generates a data stream. The control channel processor interfaces with the control computer and generates control data stream.

The multiplexer combines the various data streams into one data stream at 15.8 Mbps. The Forward Error Correction (FEC) encoder adds error correction overhead bits and provides 19.42 Mbps of data to the 16-QAM modulator. The symbol rate of the 16-QAM signal is 4.86 Megasymbols/sec.

Figure 1.5 shows the block diagram of the decoder. The 16-QAM demodulator receives IF signal from the VHF/UHF tuner and provides the demodulated data at 19.42 Mbps. The demodulator has an adaptive equalizer to combat multipath distortions common in VHF or UHF terrestrial transmission. The FEC decoder corrects almost all random or burst errors and provides the error-free data to the Sync/Data selector. The Sync/Data selector maintains overall synchronization and provides video, audio, data/text, and control data streams to appropriate processing blocks.

The control channel processor decodes the program related information. The user microprocessor receives commands from the remote control unit (RCU) and controls various functions of the decoder including the channel selection.

Table 1.1 shows the summary of system parameters.

As mentioned earlier, greatest efforts of compression are focussed on the video data, and this compression is done by the encoder. Hence, the video encoder is the most important part of the decoder. In this study, the video encoder is computer-simulated using an image sequencer. Next chapter describes the video encoder in detail.

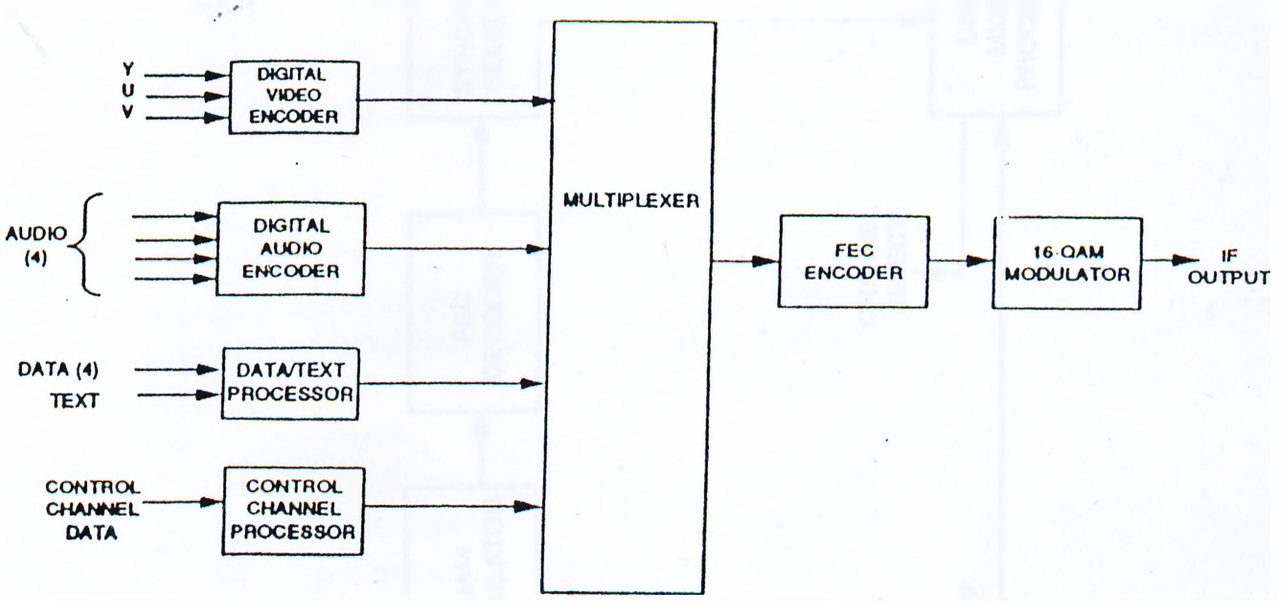


Figure 1.4: Encoder Block Diagram

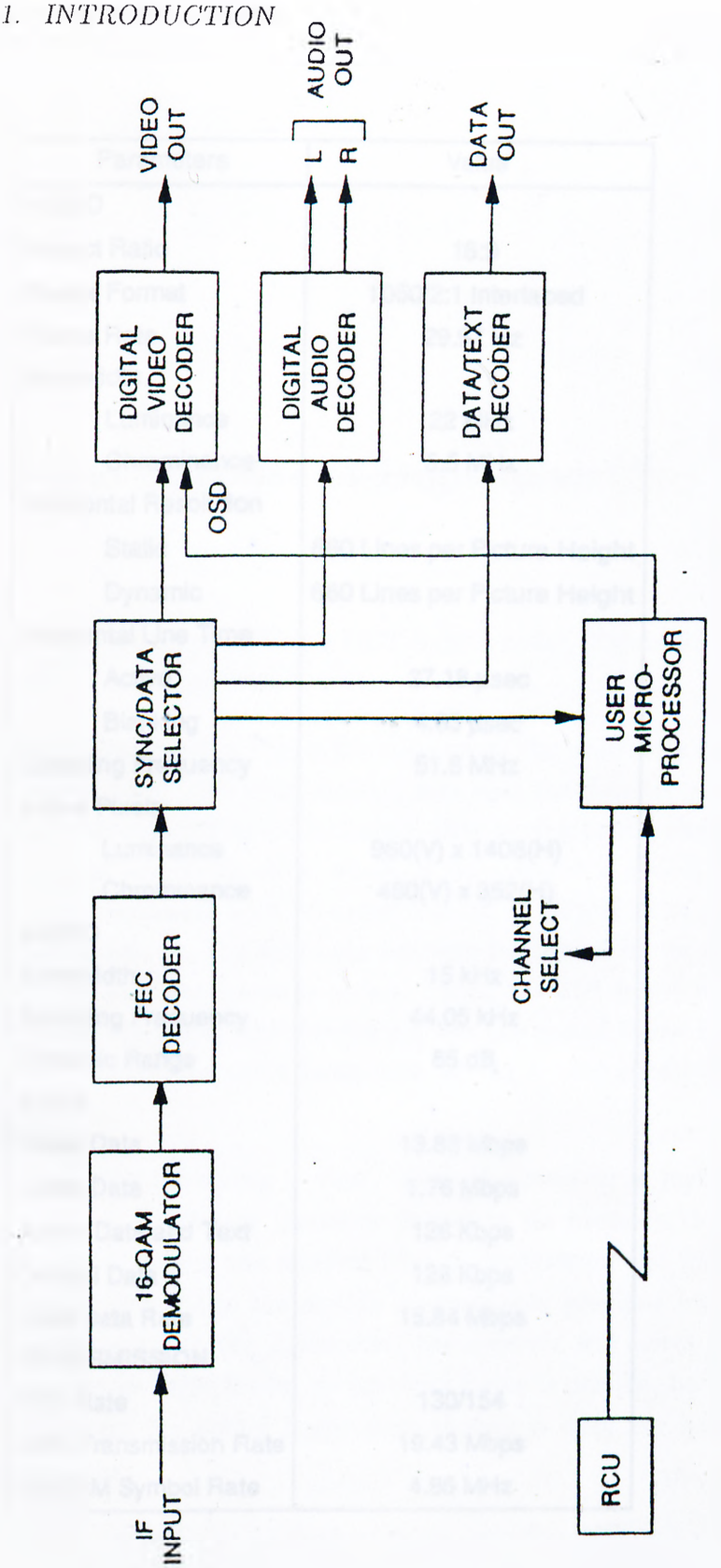


Figure 1.5: Decoder Block Diagram

Parameters	Value
VIDEO	
Aspect Ratio	16:9
Raster Format	1050/2:1 Interlaced
Frame Rate	29.97 Hz
Bandwidth	
Luminance	22 MHz
Chrominance	5.5 MHz
Horizontal Resolution	
Static	660 Lines per Picture Height
Dynamic	660 Lines per Picture Height
Horizontal Line Time	
Active	27.18 µsec
Blanking	4.63 µsec
Sampling Frequency	51.8 MHz
Active Pixels	
Luminance	960(V) x 1408(H)
Chrominance	480(V) x 352(H)
AUDIO	
Bandwidth	15 kHz
Sampling Frequency	44.05 kHz
Dynamic Range	85 dB
DATA	
Video Data	13.83 Mbps
Audio Data	1.76 Mbps
Async Data and Text	126 Kbps
Control Data	126 Kbps
Total Data Rate	15.84 Mbps
TRANSMISSION	
FEC Rate	130/154
Data Transmission Rate	19.43 Mbps
16-QAM Symbol Rate	4.86 MHz

Table 1.1: System Parameters

Chapter 2

Digital Video Encoder of DigiCipher^{TM}

The video encoder of DigiCipher^{TM} is a DCT-hybrid coder. Figure 2.1 shows the block diagram of the video encoder.

The 'refreshing' mentioned in Figure 2.1 means that the prediction frame is periodically forced to zero. This is for making sure that the decoder will have the same memory content with encoder shortly after tuning to the channel or after any transmission errors. In other words, if the refreshing period is, say, 20 frames, then the first frame is PCM coded instead of DPCM. The next 19 frames are DPCM coded, i.e. the difference between the actual frame and the motion compensated previous frame is coded. Then, the 21^{st} frame is again PCM coded. Not the difference with prediction, but the actual frame is input to the DCT-coder. This is the same thing with having a 'zero prediction' by default.

The reason for using refreshing is this: When the receiver is just tuned to the channel, it has a different 'previous frame' in its memory than the encoder has. In DPCM mode, the receiver tries to reconstruct the frame by adding the received difference to the motion compensated previous frame. If the encoder always transmits the differences, the receiver will never obtain the actual frame. But in PCM mode, no previous frame is needed, hence the receiver can reconstruct the actual frame even though it does not have the correct 'previous frame'.

The compression process can be broken down into five different subprocesses:

1. Chrominance Preprocessor

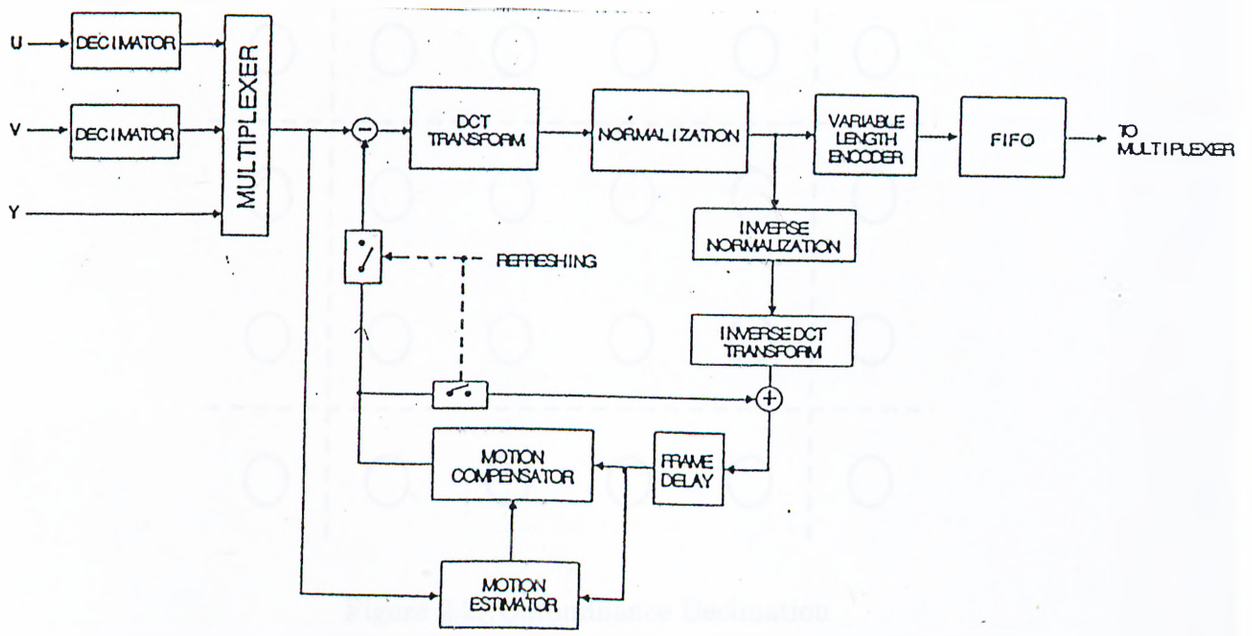


Figure 2.1: Digital Video Encoder block diagram

2. Discrete Cosine Transform

3. Coefficient Quantization (Normalization)

4. Huffman (Variable Length) Coding

5. Motion Estimation and Compensation

2.1 Chrominance Preprocessor

Human eye is less sensitive to color changes (both temporal and spatial) than the light intensity changes ([5]). To make use of this fact, The YUV color space is used. The Y component (luminance) is the light intensity, U and V (chrominance) are the color data. The relation between RGB and YUV representations is

$$Y = 0.30R + 0.59G + 0.11B \quad (2.1)$$

$$U = 0.493(B - Y) \quad (2.2)$$

$$V = 0.877(R - Y) \quad (2.3)$$

The digital RGB data from the camera is converted to YUV using (2.1) – (2.3) before being input to the encoder. In the chrominance preprocessor of the

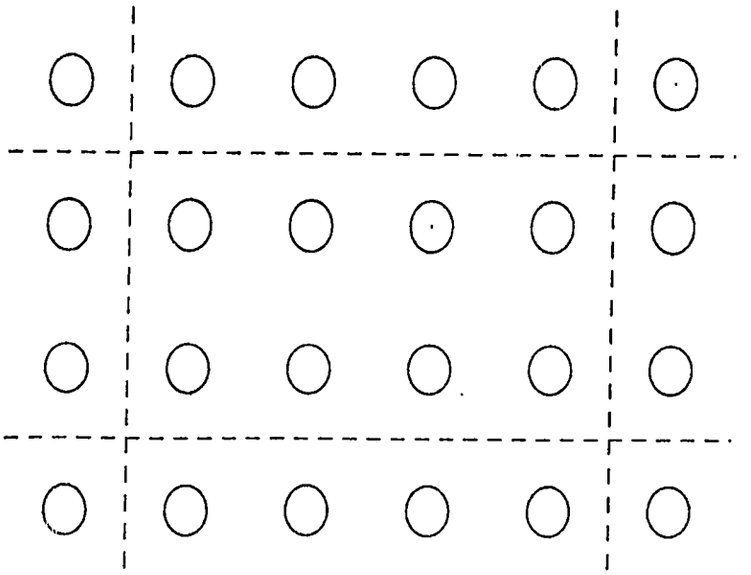


Figure 2.2: Chrominance Decimation

encoder, U and V components are decimated by a factor of four horizontally, and two vertically. Decimation is done by averaging the eight pixels (Figure 2.2). So for one frame there are 352(horizontal) by 480(vertical) points for U, 352(h) by 480(v) points for V, and 1408(h) by 960(v) points for Y after chrominance decimation. Each U and V point represents the color data of 8 luminance points.

On the decoder, U and V components are interpolated back to full resolution.

2.2 Discrete Cosine Transform (DCT)

The Discrete Cosine Transform (DCT) transforms a block of pixels into a new block of transform coefficients ([6]). Recovery at the decoder is done by applying the inverse transform. If $f(i,j)$ represents pixel intensity as a function of horizontal position, and $F(u,v)$ represents the value of each coefficient after transformation, then the equations for the forward and inverse transforms are

$$F(u,v) = \frac{4C(u)C(v)}{N^2}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} f(i,j)cos\frac{(2i+1)u\pi}{2N}cos\frac{(2j+1)v\pi}{2N} \quad (2.4)$$

$$f(i,j) = \sum_{u=0}^{N-1}\sum_{v=0}^{N-1} C(u)C(v)F(u,v)cos\frac{(2i+1)u\pi}{2N}cos\frac{(2j+1)v\pi}{2N} \quad (2.5)$$

54	59	64	68	70	70	70	70
58	63	65	67	68	67	68	69
65	70	71	68	72	73	74	72
74	72	75	76	75	75	73	72
74	73	74	74	73	73	73	71
77	76	76	76	74	73	72	70
79	77	75	75	73	72	72	69
81	80	80	78	76	74	73	70

Table 2.1: Example spatially-correlated block

where

$$C(w) = \begin{cases} 1/\sqrt{2} & \text{for } w = 0 \\ 1 & \text{for } w = 1, 2, ..., N-1 \end{cases}$$

and N is the dimension of the square block. N is chosen to be 8 because efficiency of the method does not improve very much beyond this size, while complexity grows substantially.

The advantage of this method is that most of the signal energy is compacted into a small number of transform coefficients ([3]). DCT is a very common method for compression, since it makes a very efficient use of the spatial correlation among pixel values of a typical image. In DigiCipher^{TM}, the difference between the actual pixel value and the pixel value from the motion compensated previous frame (instead of the actual pixel value) is transform coded.

The compaction of DCT can be best described by an example. Table 2.1 shows a block of data with high spatial correlation.

After DCT, the given block is transformed to the coefficient block in table 2.2.

It can be seen from table 2.1 and table 2.2 that most of the signal energy is compacted into a few coefficients (the ones on the upper left part of table 2.2) via Discrete Cosine Transform. In this example, it is not quite clear yet how this energy compaction results in more efficient coding. This will be demonstrated in the following parts, using the same sample data.

143.44	-0.14	-1.75	-0.15	-0.44	0.25	-0.15	0.18
-6.88	-5.55	-1.20	-0.83	-0.24	-0.66	-0.05	0.05
-2.34	-0.77	-0.56	-0.39	0.01	-0.21	-0.28	-0.03
-1.53	-0.57	-0.38	-0.05	0.45	0.26	-0.06	-0.27
1.19	-0.14	-0.61	0.05	0.31	0.34	0.32	-0.04
0.78	-0.14	-0.10	0.37	0.08	0.16	0.12	-0.04
1.04	-0.29	-0.16	0.01	-0.48	-0.22	0.06	0.20
-0.47	-0.59	0.44	0.13	-0.50	-0.17	-0.20	0.18

Table 2.2: DCT coefficients of the block in table 2.1

2.3 Coefficient Quantization (Normalization)

Coefficient quantization introduces small changes into image to improve coding efficiency. It rounds DCT coefficients to a limited number of bits. ([7])

In the description sheets of DigiCipher^{TM}, the method of rounding is described as *"... by shifting a coefficient from left to right, spilling the least significant bits off the end of its register."* Though this statement claims truncation instead of rounding, it is assumed that the quantizer rounds the coefficient to the nearest quantization step; because in the description sheets, there is an example about quantization, and in that example the coefficients are rounded to the nearest integer. (There is an obvious contradiction in the description sheets about the choice between rounding and truncation.)

Human eye is more sensitive to the lower spatial frequencies ([5]). Considering this fact, finer quantization is done for the DCT coefficients corresponding to lower spatial frequencies. Low frequency coefficients are the ones at the upper left part of each DCT block.

Table 2.3 shows a convenient distribution of the number of bits used for quantizing each coefficient. Sign bit is not included in the given number of bits.

If the data in table 2.2 is to be quantized according to the distribution in table 2.3, the result is as in table 2.4.

Here, the dynamic range is assumed to be -512 to 512. Hence, quantization to 9 bits (excluding sign bit) corresponds to rounding to nearest integer. Similarly, quantization to 8 bit means rounding to nearest even number. But the quantizer output is the quantization step index, not the value of that step.

9	9	8	7	6	5	4	3
9	8	7	6	5	4	3	3
8	7	6	5	4	3	3	3
7	6	5	4	3	3	3	3
6	5	4	3	3	3	3	3
5	4	3	3	3	3	3	3
4	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3

Table 2.3: A convenient distribution for number of bits

143	0	-1	0	0	0	0	0
-7	-3	0	0	0	0	0	0
-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Table 2.4: Data in table 2.2 quantized with the bit allocation as in table 2.3

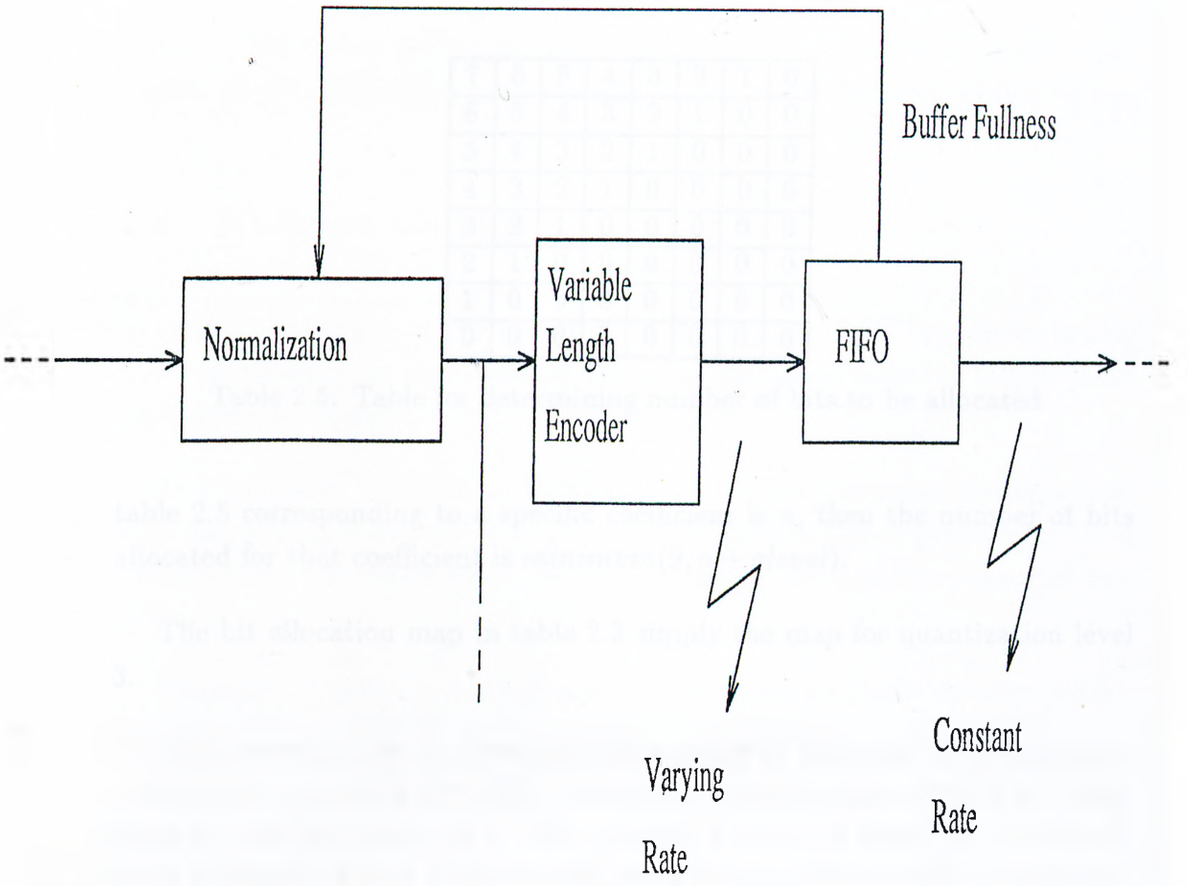


Figure 2.3: Adaptation of Quantization Level

This is some kind of implicit normalization. For example, in table 2.2, the coefficient at third row, first column is -2.34. The bit allocation map in table 2.3 suggests that it should be quantized to 8 bits. So, it will be rounded to nearest even number, that is -2. The quantization step is -2, but its index is -1. Hence, the quantizer output for -2.88 for quantizing to 8 bits is -1.

For maintaining a constant bit rate on the average, adaptive quantization is done. The total number of bits used is adjusted according to the buffer fullness (Figure 2.3)

In the best case, the encoder allocates 9 bits (not including the sign bit) for each coefficient. This is when the system is operating at maximum level on a performance scale ranging from 0 to 9 (the *"quantization level"*). If the targeted bit rate is exceeded, then the quantization level is decremented to 8 before encoding the next block.

Table 2.5 is used to determine the number bits assigned to each coefficient of an 8 by 8 block as a function of the quantization level. If the number in

7	6	5	4	3	2	1	0
6	5	4	3	2	1	0	0
5	4	3	2	1	0	0	0
4	3	2	1	0	0	0	0
3	2	1	0	0	0	0	0
2	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Table 2.5: Table for determining number of bits to be allocated

table 2.5 corresponding to a specific coefficient is n, then the number of bits allocated for that coefficient is $minimum(9, n + qlevel)$.

The bit allocation map in table 2.3 simply the map for quantization level 3.

As an example, let us determine the number of bits used to quantize the coefficient in the third row, fifth column for a quantization level of 4. From table 2.5, we find that n is 1. We compare 9 with $n + qlevel = 1 + 4 = 5$. Since $minimum(9, 5) = 5$, the system uses 5 bits to quantize $F(3, 5)$ when the quantization level is 5.

Then, let us determine the number of bits used to quantize the same coefficient when quantization level is 9. Now, $n + qlevel$ yields $1 + 9 = 10$. Since $minimum(9, 5) = 9$, the system uses 9 bits to quantize $F(3, 5)$ when the quantization level is 9.

The most objectionable artifact of excessive quantization is claimed to be the *blocking effect* ([10]). This artifact is caused by processing each block seperately. The amplitude of coefficients more or less change from block to block. So, the error introduced by quantization of a particular coefficient is different for two neighboring blocks.

Example: Let block A and block B be two neighboring blocks. For block A, $F(0, 1) = 7.1$. For block B, $F(0, 1) = 8.9$. Let the quantization level be 2, so $F(0, 1)$ is to be rounded to the nearest even number. In this case, the quantization error of $F(0, 1)$ is 0.9 for block A, and -0.9 for block B. There is a difference of 1.8.

These quantization errors are distributed rather evenly within the blocks, since each coefficient corresponds to a particular spatial frequency within the

block. This makes the quantization error jump on the block border more visible. Hence, the border between neighboring blocks becomes visible by eye.

2.4 Huffman Coding

In order to make use of the compression done by the DCT transform coding and quantization, an algorithm for assigning variable number of bits to these coefficients is required.

In DigiCipher^{TM}, Huffman Coding is used. It is a statistical coding procedure which assigns shorter code words to events with higher probability. It is optimal when the events have probabilities which are negative powers of two ([8]).

Example: Generation of Huffman codes for a simple finite-alphabet coder.

Assume that there are four events to be coded. Let these events be symbolized by letters a, b, c, and d. The probability of occurence of a is 0.5, of b is 0.25, of c and d are 0.125. Figure 2.4 shows the Huffman tree. To form this tree, the events are put to the bottom as leaves of the tree. Then the two events with least probabilities are connected together to form a node. The total probability of the two events are assigned as the probability of the node. On later iterations, this node is considered as an event, instead of considering the two leaves connected to this node as two seperate events. Iterations are continued until the root, the node with probability 1, is reached.

For assigning the code words, the path to be taken from the root to each leave is considered. Each move to a left node is a 0, and each move to a right node is a 1. So, the code words assigned to the events in this example are as follows:

a : 0

b : 10

c : 110

d : 111

Here is a typical observation of 16 events: *aacadadbbcbaaaba*. The given Huffman coder codes this event as 0011001110111101011010000100.

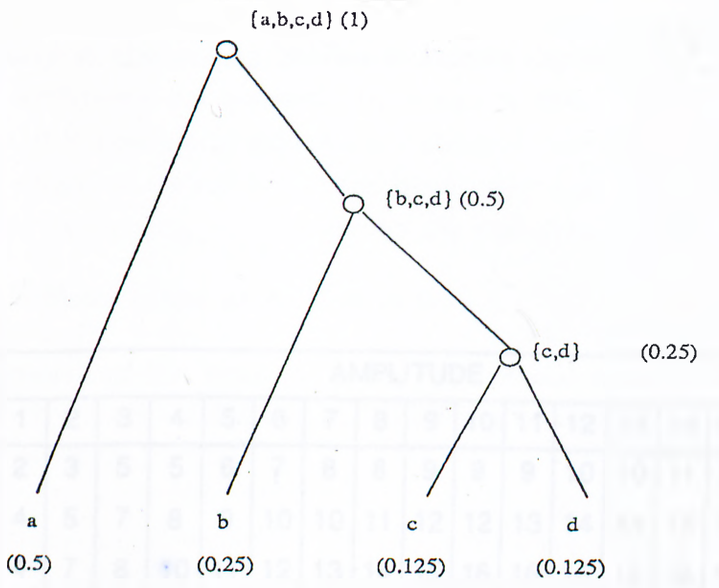


Figure 2.4: Huffman tree for the given example

Note that there is no code word which is identical with the first bits of a longer code word. This is to ensure that the decoder will have no ambiguities in interpreting the bits sent by the encoder. The decoder has the same code book with the encoder.

In order to apply Huffman coding to this application, the 8 by 8 coefficients are serialized into a sequence of 64, and *amplitude/runlength* coded. Scanning the sequence of 64, an event is defined to occur each time a nonzero coefficient is encountered. A code word is then assigned indicating the amplitude of the coefficient and the number of zeros preceding it (runlength). A special code word is reserved for informing the end of the block.

The encoder compares the length of the code words and the number of bits required to directly code the coefficients. When it is more efficient, it codes the coefficients directly. When direct coding is applied, a special code word is sent to inform the decoder about this.

DigiCipher^{TM} uses a Huffman code book which had been formed using the event probabilities obtained by making experiments on many image sequences. In description sheets of DigiCipher^{TM}, a table of bit lengths for each Huffman code word is given (Table 2.6). The bit length does not include the sign bit. If the amplitude or runlength is larger than 15, a special code word is generated to inform the decoder about this, and then the amplitude and runlength are sent uncoded.

	AMPLITUDE															
RUNLENGTH	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	2	3	5	5	6	7	8	8	9	9	9	10	10	11	11	11
1	4	5	7	8	9	10	10	11	12	12	13	14	14	15	15	16
2	4	7	8	10	11	12	13	14	15	16	16	16	18	18	19	19
3	5	8	10	11	13	14	15	16	17	18	18	19	19	19	21	21
4	6	9	12	14	15	17	18	18	20	21	20	22	28	29	29	29
5	7	10	13	16	18	19	22	21	21	29	29	29	29	29	29	29
6	7	11	14	17	18	19	19	17	20	21	28	28	28	28	28	28
7	8	12	16	18	19	22	20	28	28	28	28	28	28	28	28	28
8	9	14	17	21	28	28	28	28	28	28	28	28	28	28	28	28
9	9	15	19	28	28	28	28	28	28	28	28	28	28	28	28	28
10	10	16	20	28	28	28	28	28	28	28	28	28	28	28	28	28
11	11	18	28	22	28	28	28	28	28	28	28	28	28	28	28	28
12	11	17	28	22	28	28	28	28	28	28	28	28	28	28	28	28
13	11	17	28	22	28	28	28	28	28	28	28	28	28	28	28	28
14	12	20	28	22	28	28	28	28	28	28	28	28	28	28	28	28
15	13	20	28	22	28	28	28	28	28	28	28	28	28	28	28	28

Table 2.6: Number of bits used for each code word of two-dimensional Huffman code book

The efficiency of this coding process is heavily dependent on the order in which the coefficients are scanned. By scanning from high amplitude to low amplitude, it is possible to reduce the number of runs of zero coefficients typically to a single run at the end of the block. Any long run at the end of the block would be represented efficiently by the *end-of-block* code word.

Example: Huffman coding of the block in table 2.4

The serialization of the block to a one dimensional array of length 64 yields $[143, 0, -7, -1, -3, -1, 0, 0, \ldots, 0]$. According to the above definitions, the events are as follows:

Event #	Amplitude	Runlength
1	143	0
2	7	1
3	1	0
4	3	0
5	1	0

Calculation of number of bits used to code the block:

For event 1: 4 (For informing direct coding mode) + 9 (For coding amplitude) + 6 (For coding runlength) = 19.

For event 2: 10 (From table 2.6).

For event 3: 2 (From table 2.6).

For event 4: 5 (From table 2.6).

For event 5: 2 (From table 2.6).

For end-of-block code: 3 (Assumed).

For signs of non-zero coefficients: 5 (Since there are 5 non-zero coefficients).

Total: $19 + 10 + 2 + 5 + 2 + 3 + 5 = 46$.

So, 64 coefficients has been coded with 46 bits. If direct coding was to be applied, 339 bits (sum of the numbers in table 2.3 + sign bits) should have been used.

2.5 Motion Estimation–Compensation

Motion compensation is a method for improving the prediction of the current frame using the previous frame. It is often the case that some part of the current picture is very highly correlated with some part of the previous frame. The aim of motion estimation is finding out 'which part of the previous frame is most correlated with the specific part of the current frame'.

There are several methods of motion estimation–compensation. For the object oriented method, for example, the 'specific part' mentioned above is an object. This is intuitively best method in finding out maximum correlations, but it is very difficult to design a very fast object-recognizing system, and it needs tremendous amount of computation and complex circuitry. A much easier method is *block matching*. Block matching is the most popular method used nowadays. DigiCipher^{TM} also uses this method.

In DigiCipher^{TM}, the current luminance frame is divided into blocks of 32(horizontally) by 16(vertically). Because of chrominance decimation, the block size is 8 by 8 for chrominance frames. So in a frame, there are 44 by 60 blocks for each of Y, U , and V components. For each block, its neighborhood at the previous frame is searched for finding a section of 32 by 16 which minimizes the difference with the block being handled. The spatial distance between the upper left corner of the difference-minimizing section is generated as the *motion vector*. The difference-minimizing section is used as the prediction for the block, and the motion vectors (one for each block) are transmitted so that the decoder will be able to have the same prediction.

Example: Motion estimation-compensation by block matching

The block size used in this example is 4 by 2 instead of 32 by 16, since it is the same idea, but it is much easier to demonstrate the idea by a small size.

Figure 2.5 shows a region in the current frame, and figure 2.6 shows the region with the same spatial location in the previous frame. The block on which motion estimation–compensation is being done in this example has been marked in figure 2.5.

Now, the region in the previous frame is searched to find a section which minimizes Σ *absolute errors*. If there were no motion estimation–compensation, the section to be used as prediction would be the one having the same spatial location as the block that is being operated on. In this case, the difference

130	128	137	133	127	123	120	118	117	118	120	122
131	127	136	136	130	126	122	119	116	117	122	121
129	126	135	143	79	81	83	116	114	119	121	120
134	130	136	140	73	76	115	117	115	120	122	120
133	129	134	137	130	123	119	115	116	119	120	120
137	133	135	137	135	128	123	119	118	119	118	117

Figure 2.5: Region in the current frame

130	128	137	133	127	123	120	118	117	118	120	122
131	127	136	136	130	126	122	119	116	117	122	121
129	126	135	143	137	130	127	116	114	119	121	120
134	130	84	85	90	128	115	117	115	120	122	120
133	129	80	73	123	125	122	115	116	119	120	120
137	133	135	137	135	128	123	119	118	119	118	117

Figure 2.6: Region in the previous frame

block would be

-58	-49	-44	0
-17	-52	0	0

Then,

Σ *absolute errors* $= abs(-58) + abs(-49) + abs(-44) + abs(0) + abs(-17) + abs(-52) + abs(0) + abs(0) = 220.$

When motion estimation-compensation comes into action, the prediction can be improved. The section marked in figure 2.6 yields minimum mean absolute error in the region, hence it is a better prediction.

When the error-minimizing section is used as prediction, the difference block is

-5	-4	-7	-12
-7	3	-8	-8

Then,

Σ *absolute errors* $= abs(-5) + abs(-4) + abs(-7) + abs(-12) + abs(-7) + abs(3) + abs(-8) + abs(-8) = 54$.

The motion estimator finds out which section minimizes the error, and generates its relative location to the block being processed as motion vector. In this example, the motion vector is [2 (horizontal) , -1 (vertical)]. By convention, positive horizontal component denotes a motion towards right, and positive vertical component denotes a motion downwards. So the block has moved to its current location by moving two pixels right and one pixel up.

Chapter 3

Simulations

3.1 Equipment

The simulations were done in a DVSR VTE–100 image sequencer. Figure 3.1 shows the block diagram of the image sequencer. The host computer was a SUN–3.80 Workstation. The simulation program was written in C programming language. The functions which supply the interaction with the image sequencer were available in the program library.

3.2 Procedure

A highly modular C program was written to simulate the system. Assumption list was rather long, so the program was written in a highly modifiable way.

3.2.1 Assumption List

Here are the assumptions made when the system was simulated:

—Zero order interpolation is done for missing chrominance components. In other words, each chrominance component is repeated eight times (4 times horizontally and 2 times vertically) to get equal number of chroma points as luminance points.

—Motion vectors have integer components. This assumption is stated here explicitly, because in some systems (e.g. MPEG [9]) the motion vectors may

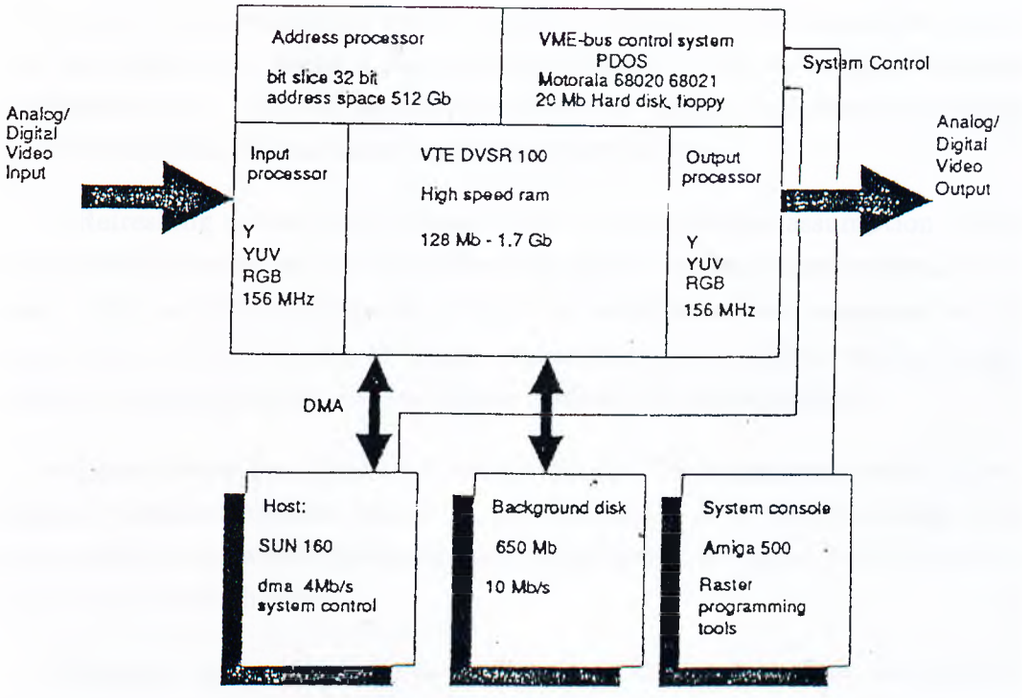


Figure 3.1: The block diagram of the simulation system, DVSR VTE–100

have non-integer components which are integer multiples of 0.5. A non-integer component denotes that the sections corresponding to motion vectors with nearest two integer components have been averaged to obtain the prediction. For example, the motion vector $[2, 1.5]$ means that the sections from the previous frame corresponding to the motion vectors $[2, 1]$ and $[2, 2]$ are averaged to get the prediction. Similarly, $[3.5, 4.5]$ means that the sections corresponding to the motion vectors $[3, 4], [3, 5], [4, 4]$ and $[4, 5]$ are averaged to get the prediction.

—Maximum motion vector lengths are 25 horizontally and 15 vertically. In other words, each block are searched for an error-minimizing section in a region of dimensions 82(horizontally) by 46(vertically). This is not a critical assumption, and it was made for computational convenience.

—Minimization cost function is mean absolute error. This was assumed because it is widely used in the motion estimators. This probably is not a critical assumption.

—Only luminance signal is taken into account for motion estimation. The generated motion vectors are used for both luminance and chrominance motion compensation. The motion vectors are scaled for chrominance blocks so that

the chrominance decimation would be taken into account. This may be a critical assumption, i.e. doing it this way or another way may have big differences in performance. This was assumed because it is logical, and there is nothing in the description sheets which implies another method.

—Refreshing period is 19 frames. This is also a critical assumption. This assumption is made because the refreshing period in some other systems is in that order, and this assumption allowed the selection of test sequences to be made from a richer library of image sequences. (It is difficult to find large variety of long sequences due to limited memory of the sequencer).

—Quantization level update is once a frame. The experiment results show that the assumed update period is not too long. If it were too long, the quantization level would fluctuate fastly from frame to frame, but it did not happen out to be the case.

—Dynamic range of transform coefficients is -512 to 512. This is a critical assumption, and it was made to guarantee avoiding overflow in PCM mode frame (The first frame after refreshing). It would be clever to use a different dynamic range in PCM mode, but there was no sign of this in the description sheets, hence constant dynamic range was assumed.

—The length of Huffman code word of each amplitude/runlength pair was given in a table in the description sheets. This table (Table 2.6) was used to calculate the number of bits needed to encode each block. Special code words, whose lengths have not been given in the description sheets, were assigned arbitrary lengths. Assigning arbitrary lengths to special code words is not critical for system performance, unless too long codes are assigned.

—Zig-zag scan is applied during the serialization of the 8 by 8 block of quantized coefficients into a sequence of 64 (Figure 3.2). This is a very important assumption, since the scanning pattern affects the coding efficiency very much. The DCT coefficients corresponding to lower spatial frequencies are expected to have higher amplitudes for typical image sequences. The assumed scanning pattern scans the coefficients from the coefficients of low frequencies to coefficients of high frequencies. (It is the same as the scanning pattern used in MPEG [9].)

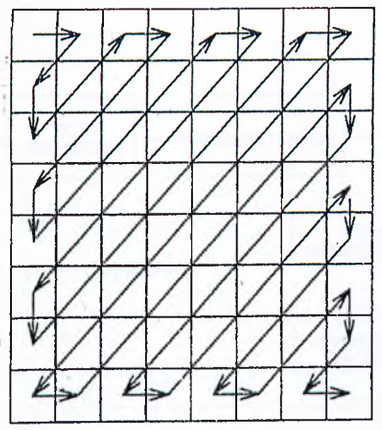


Figure 3.2: Zig-zag scan pattern

3.2.2 Image Sequences

Three image sequences with different characteristics were used. One of them (*Costgirls*) is a slowly moving sequence. It shows three little girls playing with toys in a room. The second sequence (*Car*) is a fastly moving sequence. In this sequence, the camera is panning a car which enters a parking area. The last image sequence is a computer-generated sequence (*Cross*) with still parts, fastly moving parts,, suddenly appearing objects, high special details, etc.

The length of each sequence is 19, and their size is 352 by 288. The actual system has a size of 1408 by 960. So, some of the system parameters (such as transmission rate, buffer size) was scaled by the ratio of these sizes, 0.075.

3.2.3 Relation Between System Blocks and Program Functions

The FIFO buffer was simulated by a counter variable. The variable length encoding is not actually done. Only the number of bits needed for variable length encoding is calculated, and added to the counter. This is done because there was no need to make the actual coding. The adaptation of the quantization level depends only on the queue length, not the content of the queue. Hence, there is no reason to make the actual coding.

One question rises now: How can we observe what the decoder receives (assuming no transmission errors) if we do not actually create the bits which

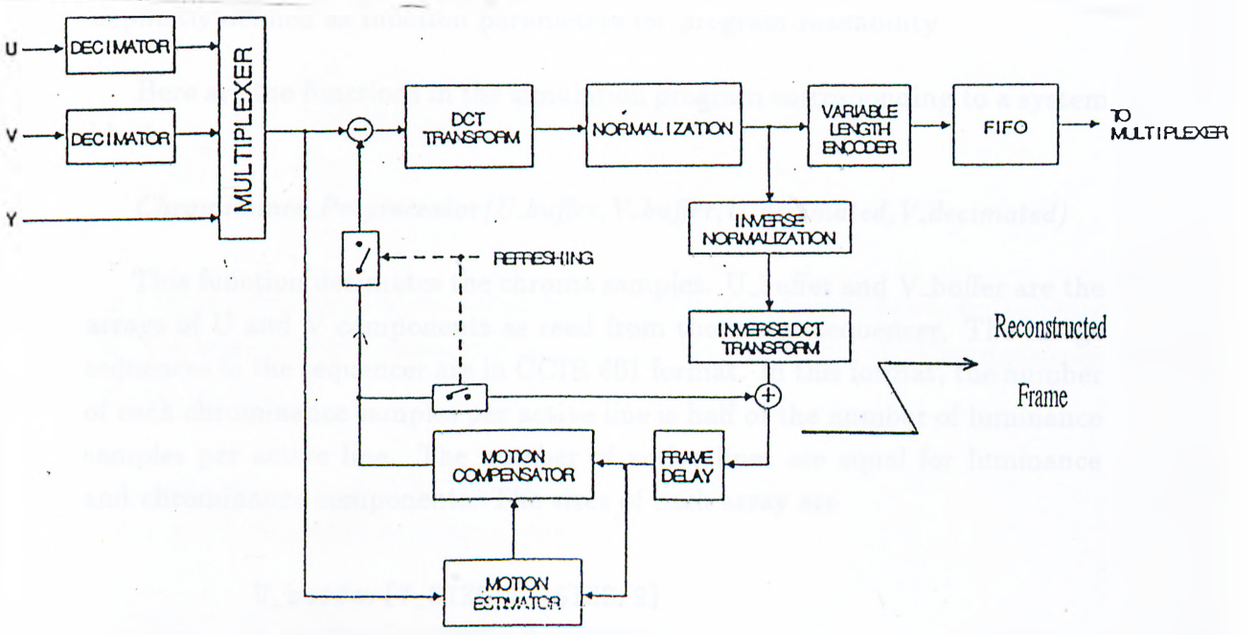


Figure 3.3: Monitoring the reconstructed frame

the decoder needs to construct the image? Before answering this question, it is useful to point out a basic principle of DPCM coding: The encoder and the decoder should have the same prediction so that there will be no cumulative errors. This implies that the predictor in the decoder and encoder should have the same input. In this system, the inputs to the predictor are the motion vectors and the previous frame. So, inside the encoder, the previous frame which the decoder is supposed to have exists as an input to the predictor. This input is an output from a frame delay. Then, the input to this frame delay is exactly the same frame as the current frame which the decoder has, i.e. the reconstructed frame.

Summarizing the above paragraph, to monitor the reconstructed signal in the receiver with no transmission errors, the frame in the encoder which is to be fed to the predictor for the next prediction was examined (Marked in Figure 3.3).

The frame delay was implemented automatically by the program structure. There was a loop in the program in which all the operations for a frame was done. This loop ran from the first frame to the last frame, and each time a new frame was read from the sequencer, it automatically implemented the frame delay.

Other blocks were seperate functions. Inputs and outputs to them are explicitly defined as function parameters for program readability.

Here are the functions in the simulation program corresponding to a system block:

Chrominance_Preprocessor(U_buffer, V_buffer, U_decimated, V_decimated)

This function decimates the chroma samples. U_buffer and V_buffer are the arrays of U and V components as read from the image sequencer. The image sequences in the sequencer are in CCIR 601 format. In this format, the number of each chrominance samples per active line is half of the number of luminance samples per active line. The number of active lines are equal for luminance and chrominance components. The sizes of each array are

```
U_buffer[Y_SIZE][X_SIZE/2]
V_buffer[Y_SIZE][X_SIZE/2]
U_decimated[Y_SIZE/2][X_SIZE/4]
V_decimated[Y_SIZE/2][X_SIZE/4]
```

where Y_SIZE is the number of active lines (288) and X_SIZE is the number of samples per active line for luminance (352).

In other words, the function *Chrominance_Preprocessor()* decimates the chrominance samples by a factor of four horizontally and two vertically. The main loop for this function is

```
for(i=0;i<Y_SIZE;i++)
  for(j=0;j<X_SIZE;j++)
  {
    U_decimated[i][j] = (U_buffer[i*2][j*2] + U_buffer[i*2][j*2+1]
              + U_buffer[i*2+1][j*2] + U_buffer[i*2+1][j*2+1] )/4;
    V_decimated[i][j] = (V_buffer[i*2][j*2] + V_buffer[i*2][j*2+1]
              + V_buffer[i*2+1][j*2] + V_buffer[i*2+1][j*2+1] )/4;
  }
```

Motion_Estimate(Y_Current, Y_Previous, Motion_Vectors)

Y_Current is the array of luminance samples of the current frame, and Y_Previous is the array of luminance samples of the previous frame. Motion_Vectors is the array of motion vectors. The motion vectors are generated

in this function using full search, as described in Chapter 2. The sizes of the arrays are

```
Y_Current[Y_SIZE][X_SIZE]
Y_Previous[Y_SIZE][X_SIZE]
Motion_Vectors[NUM_OF_BLOCKS_X][NUM_OF_BLOCKS_Y][2]
```

The last dimension of the array *Motion_Vectors* denotes the two components for each motion vector, namely x-component and y-component. The main loop for this function is

```
for(i=0;i<NUM_OF_BLOCKS_Y;i++)
  for(j=0;j<NUM_OF_BLOCKS_X;j++)
  {
    min = 1000000;
    for(k=-MAX_MOVE_Y;k<=MAX_MOVE_Y;k++)
      for(l=-MAX_MOVE_X;l<=MAX_MOVE_X;l++)
        if((Error=MAE(i,j,k,l,Y_Current,Y_Previous))<min)
        {
          min = Error;
          Motion_Vectors[i][j][0] = k;
          Motion_Vectors[i][j][1] = l;
        }
  }
```

The function *MAE(i,j,k,l,Y_Current,Y_Previous)* calculates the mean pixelwise absolute error between the 32 by 16 block from the current frame which has its upper left corner at (i,j) and the block from the previous frame which has its upper left corner at (i-k,j-l). When the indices exceed the frame bounds, the pixel from the previous frame which is supposed to be located at the bound-exceeding coordinate is taken to be zero. For example, if $i = 0, j = 0, k = 3, l = 2$, then

$$Y\_Current[i][j] = Y\_Current[0][0]$$

$$Y\_Current[i-k][j-l] = Y\_Current[-3][-2] = 0$$

Hence, the absolute difference between these two pixels is taken to be $Y\_Current[0][0]$. Note that the indices of *Y_Current* never exceed the frame bounds.

Motion_Compensate(Motion_Vectors, Y_Previous, U_Previous, V_Previous, Prediction_of_Y, Prediction_of_U, Prediction_of_V)

This function uses the motion vectors and the previous frame to generate the prediction. For the generation of the prediction of U and V, x-component of the motion vectors are divided by 4, and y-component of the motion vectors are divided by 2. The results are rounded to the nearest integer. The divisions are for scaling the motion vectors by the decimation ratio of the chrominance preprocessing. When the motion-compensated coordinate for a pixel exceeds the frame bounds, the prediction for the value of that pixel is assigned to zero. The main loop of this function is

```
for(i=0;i<NUM_OF_BLOCKS_Y;i++)
  for(j=0;j<NUM_OF_BLOCKS_X;j++)
  {
    MY = Motion_Vectors[i][j][0];
    MX = Motion_Vectors[i][j][1];
    for(m=0;m<16;m++)
      for(n=0;m<32;n++)
        is_off_bound = ((i*16+m-MY)<0) || ((i*16+m-MY)>Y_SIZE)
                    || ((j*32+n-MX)<0) || ((j*32+n-MX)>X_SIZE);
        if(is_off_bound)
          Prediction_of_Y[i*16+m][j*32+n] = 0;
        else
          Prediction_of_Y[i*16+m][j*32+n]
                        = Y_Previous[i*16+m-MY][j*32+n-MX];
        if( ! ( (m%2) || (m%4) ) )
        {
          if(is_off_bound)
          {
            Prediction_of_U[(i*16+m)/2][(j*32+n)/4] = 0;
            Prediction_of_V[(i*16+m)/2][(j*32+n)/4] = 0;
          }
          else
          {
            Prediction_of_U[(i*16+m)/2][(j*32+n)/4]
                    = U_Previous[(i*16+m-MY)/2][(j*32+n-MX)/4];
            Prediction_of_V[(i*16+m)/2][(j*32+n)/4]
                    = V_Previous[(i*16+m-MY)/2][(j*32+n-MX)/4];
          }
```

```
        }
}
```

DCT(Prediction_Error, Transform, DCT_table)

This function takes an 8 by 8 block of prediction errors, i.e. $(Current - Prediction)$ as input, takes its DCT transform according to Equation 2.4, and writes to the 8 by 8 real number array $Transform$. In fact, *Prediction_Error* array has the dimensions of a full frame size, but the DCT function operates in an 8 by 8 sub-block of this array each time it is invoked. The location of the sub-block within the frame is adjusted by adding a proper offset to the beginning address of the *Prediction_Error* array to be passed to the *DCT* function as parameter. For example, if we want to take the DCT of an 8 by 8 sub-block of the array $Prediction\_Error\_Y$, and the coordinate of the upper left corner of the sub-block is $(72, 96)$, then the function call will be

```
DCT(&Prediction_Error_Y[72][96], Transform, DCT_table);
```

Instead of calculating the cosines each time, the function uses a table which holds 32 samples of a full cosine wave. The cosines to be used in *DCT()* are calculated by doing modular arithmetic on the table indices. For example, $cos(\frac{(2i+1)u\pi}{16})$ =DCT_table[((2*i+1)*u)%32].

This is done only for increasing the calculation speed of the simulation program. There are various fast implementations of DCT, but they are not implemented in this program, since they are more complex, and computation time was not a strict limitation. (The simulation of encoding of a sequence of length 19 takes about seven hours of computing time with the SUN 3.80 workstation.)

IDCT(Quantized_Transform, Prediction_Error, DCT_table)

This function is simply the inverse of the function *DCT()*. It takes an 8 by 8 array of quantized transform coefficients from the output of the denormalizer as input, takes the inverse DCT according to Equation 2.5, rounds to the nearest integer, and writes to the proper 8 by 8 sub-block of the array *Prediction_Error*. The size of *Prediction_Error* array is full frame size, and the location of the sub-block is adjusted by adding a proper offset to the beginning address of the array *Prediction_Error* to be passed to the *IDCT()* function as parameter. For example,

```
IDCT(Quantized_Transform,&Prediction_Error_Y[72][96], DCT\_table)
```

stores the results of the inverse DCT operation to the 8 by 8 sub-block of the array Prediction_Error_Y, the coordinates of the upper left corner being (72,96).

The array *Prediction_Error* is used to reconstruct the image by adding with the *Prediction* array.

Quantize(Transform, Normalized_Transform, qlevel, qtable)

This function reads an 8 by 8 real number array of transform coefficients, i.e. the output array of the function *DCT()*, quantizes the coefficients according to the quantization level (*qlevel*) and using the bit allocation map (*qtable*) (Table 2.5), and writes the result to the 8 by 8 integer array $Normalized\_Transform$. The array $Normalized\_Transform$ holds the quantization level indices rather than the quantized levels. For example, if the value of a coefficient is 2.2, and it is supposed to be rounded to the nearest even integer, then the quantized level is 2, but the quantization level index is 1, and this 1 is stored to the array $Normalized\_Transform$. In other words, normalization is also carried out in this function. (This also explains why the name '$Normalized\_Transform$' is given to the output array instead of the name '$Quantized\_Transform$'). The main loop of this function is

```
for(i=0;i<8;i++)
  for(j=0;j<8;j++)
    Normalized_Tansform[i][j] = (int)(0.5+Transform[i][j]
                  / two_to_the_power(max(0,qtable[i][j]-qlevel)));
```

Denormalize(Normalized_Transform, Quantized_Transform, qlevel, qtable)

This function prepares the input to the function *IDCT()*. The 8 by 8 integer array Quantized_Transform is the output of this function. *Denormalize()* denormalizes the normalized transform coefficients so that they will represent the quantized levels rather than the quantization level indices. *qlevel* is the quantization level which had been used in quantizing the original transform coefficients, and *qtable* is the bit allocation map (Table 2.5). The main loop of this function is

```
for(i=0;i<8;i++)
  for(j=0;j<8;j++)
```

```
Normalized_Transform[i][j] = Normalized_Transform[i][j]
                * two_to_the_power(max(0,qtable[i][j]-qlevel));
```

Huffman(Normalized_Transform, qlevel, codebook)

This function calculates the number of bits needed to code the 8 by 8 integer array *Normalized_Transform* by the variable length coding method. *qlevel* is the quantization level that has been used in calculating the number of bits needed to directly code a coefficient. If the amplitude or runlength of a certain coefficient exceeds 16, then the coefficient is directly coded.

codebook is the two-dimensional array which holds the number of bits used to Huffman-code an event if the event has an amplitude less than or equal to 16 and runlength less than 16 (Table 2.6).

First, the 8 by 8 block is serialized to a one-dimensional array of length 64, *serialized*[], according to the scanning pattern given in figure 3.2.

Then, the number of non-zero coefficients and their locations are calculated by the loop

```
for(count=0,i=0;i<64;i++)
  if(serialized[i])
    location[++count]=i;
```

After this loop has been terminated, *count* holds the number of non-zero coefficients and the array *location*[*i*] holds their locartions in the array *serialized*[]. Then, the calculation of the number of bits needed to code the block is done by

```
no_of_bits=0;
if(count>0)
  for(i=1;i<=count;i++)
  {
    if( (abs(serialized[location[i]])>16)
        || ( (location[i]-location[i-1]) > 16) )
      sum += uncoded(location[i],qlevel)+RUNLENGTH_BITS+UNCODED;
    else
      sum += codebook[abs(serialized[location[i]])-1]
                                    [location[i]-location[i-1]-1];
  }
```

Here, *uncoded(x,y)* is a function which calculates the number of bits needed to directly code a coefficient with location x witihin the array *serialized*[] when the quantization level is y. $RUNLENGTH\_BITS$ is the number of bits needed to directly code the runlength, and it is taken to be 6. $UNCODED$ is the number of bits needed for the special code word for informing that direct coding will be done, and it is taken to be 4.

After the above loop has been terminated, we have to add the number of bits needed to inform the end of the block. The length of end-of-block code word is taken to be 5. We also have to add the bits for the signs of the coefficients. Note that we have to send the sign of non-zero coefficients only, hence we need as much bits as the number of non-zero coefficients. (This was not specified in the description sheets, but it seems that this is an efficient way of transmitting the sign bits). So, these additions are done by

```
sum += END_OF_BLOCK + count;
```

and then the resulting sum is sent as the output of the function. In the description sheets of DigiCipher^{TM}, it was mentioned that the system would check the cases when directly coding the block would be more efficient than Huffman coding, and in these cases the block would be coded directly. So, the number of bits needed to code a block $Normalized\_Transform$[] by the encoder is calculated by

```
minimum(dirbit[qlevel],Huffman(Normalized_Transform,qlevel,codebook))
```

where *dirbit*[] is a one-dimensional array of length 10, and it holds the number of bits needed to directly code the block for each quantization level.

Interpolate(U_decimated, V_decimated, U_buffer, V_buffer)

This function is not a sub-block of the encoder. It is a part of the decoder. It is used in the simulation program as a final stage after all other processing of a frame, so that we can obtain a picture which would be identical to the picture that would have been reconstructed in the decoder.

The simulation is started by initializing the system parameters. The reconstructed previous frame, which is to be used in the prediction after motion compensation, is initialized to a constant value of 128 for all pixels. This corresponds to the *refreshing*. Since the sequences used in the simulation have a length of 19 frames each, only one refreshing is supposed to be done during

the simulation of the encoding of a sequence. For getting meaningful results, the simulation must be done for a time period in which the encoder and decoder are synchronized. This is possible by starting the simulation with a refreshing. Summarizing this paragraph, the simulation of the 19-frame-long sequence starts with a refreshing, and there is no other refreshing throughout the simulation.

After the initialization, the program proceeds as

```
for(picture_no=1;picture_no<=19;++picture_no)
{
  read_from_sequencer(picture_no,Y,Y_Current);
  read_from_sequencer(picture_no,U,U_Current);
  read_from_sequencer(picture_no,V,V_Current);

  Chrominance_Preprocessor(U_buffer,V_buffer,U_Current,V_Current);

  Motion_Estimate(Y_Current,Y_Previous);
  Motion_Compensate(Motion_Vectors,Y_Previous,U_Previous,V_Previous,
                Prediction_of_Y, Prediction_of_U, Prediction_of_V);

/* Calculation of prediction error starts */

  for(i=0;i<Y_SIZE;i++)
    for(j=0;j<X_SIZE;j++)
      Prediction_Error_Y[i][j]=Y_Current[i][j]-Prediction_of_Y[i][j];

  for(i=0;i<Y_SIZE/2;i++)
    for(j=0;j<X_SIZE/4;j++)
    {
      Prediction_Error_U[i][j]=U_Current[i][j]-Prediction_of_U[i][j];
      Prediction_Error_V[i][j]=V_Current[i][j]-Prediction_of_V[i][j];
    }
/* Calculation of prediction error ends.*/

/* The procedure of

       DCT -> quantization -> calculation of # of bits needed to
                   |          encode the block
                  \|/
```

```
            denormalization -> IDCT (Reconstruction of
                                     Prediction Errors)
     for each 8x8 block of the frame is started                    */

    for(i=0;i<Y_SIZE; i+=8)      /* increment by DCT block size = 8  */
      for(j=0;j<X_SIZE; j+=8)    /* increment by DCT block size = 8  */
      {
        DCT(&Prediction_Error_Y[i][j], Transform, DCT_table);
        Quantize(Transform,Normalized_Transform,qlevel,qtable);
        queue_length += minimum(dirbit[qlevel],Huffman(
                              Normalized_Transform,qlevel,codebook));
        Denormalize(Normalized_Transform,Quantized_Transform,qlevel,
                                                          qtable);
        IDCT(Quantized_Transform,&Prediction_Error_Y[i][j],
                                                       DCT_table);
        /* IDCT writes the reconstructed prediction error over the
           original value                                          */
      }

    for(i=0;i<Y_SIZE/2; i+=8)
      for(j=0;j<X_SIZE/4; j+=8)
      {
        DCT_UV(&Prediction_Error_U[i][j], Transform, DCT_table);

/* DCT_UV() is a very little modified version of DCT(). The only
   difference is the full frame size. The full frame size has to
   be known for calculation of  the relative positions of each
   pixel of the 8x8 block in the full frame array. The full
   frame size for chrominance is X_SIZE/4 by Y_SIZE/2, i.e.
   88x144; where full frame size for luminance is X_SIZE by
   Y_SIZE, i.e. 352x288.                                         */

        Quantize(Transform,Normalized_Transform,qlevel,qtable);
        queue_length += minimum(dirbit[qlevel],Huffman(
                              Normalized_Transform,qlevel,codebook));
        Denormalize(Normalized_Transform,Quantized_Transform,qlevel,
                                                          qtable);
        IDCT_UV(Quantized_Transform,&Prediction_Error_U[i][j],
                                                       DCT_table);
/* IDCT_UV() is the modified version of IDCT(), the only
```

```
    difference being the full frame size.                          */


       DCT_UV(&Prediction_Error_V[i][j], Transform, DCT_table);
       Quantize(Transform,Normalized_Transform,qlevel,qtable);
       queue_length += minimum(dirbit[qlevel],Huffman(
                              Normalized_Transform,qlevel,codebook));
       Denormalize(Normalized_Transform,Quantized_Transform,qlevel,
                                                          qtable);
       IDCT_UV(Quantized_Transform,&Prediction_Error_V[i][j],
                                                       DCT_table);
     }

     for(i=0; i<Y_SIZE; ++i)
       for(j=0; j<X_SIZE; ++j)
         Y_Previous[i][j] = saturation_arithmetic_addition(
                    Prediction_of_Y[i][j],Prediction_Error_Y[i][j]);

     for(i=0; i<Y_SIZE/2; ++i)
       for(j=0; j<X_SIZE/4; ++j)
       {
         U_Previous[i][j] = saturation_arithmetic_addition(
                    Prediction_of_U[i][j],Prediction_Error_U[i][j]);
         V_Previous[i][j] = saturation_arithmetic_addition(
                    Prediction_of_V[i][j],Prediction_Error_V[i][j]);
       }

     write_to_sequencer(picture_no, Y, Y_Previous);
     Interpolate(U_Previous, V_Previous, U_buffer, V_buffer);
     write_to_sequencer(picture_no, U, U_buffer);
     write_to_sequencer(picture_no, V, V_buffer);

/* The decrementation of queue length by the amount of bits that
   have been transmitted during a frame period.                    */
     if (queue_length > TRANSMISSION_RATE/FRAME_RATE)
       queue_length -= TRANSMISSION_RATE/FRAME_RATE;
     else
       queue_length = 0;    /* underflow protection */

/* Quantization level update */
```

```
    qlevel = 9 - ((queue_length*10)/BUFFER_SIZE);

/* Buffer Overflow Check */
    if ( qlevel < 0 )
    {
      printf('BUFFER OVERFLOW!');
      exit(0);
    }
}
```

3.2.4 Visual Performance Detection

The screen was divided into four (Figure 3.4). The original image was recorded to the upper left part, the prediction to upper right, the reconstructed image to lower left, and the absolute differences to lower right part. The difference between original and reconstructed images, and the difference between original and prediction images were superposed, each having different colors. This is a very convenient way to observe the correlation between the prediction error, which is the signal to be compressed for transmission, and the reconstruction error, which is the error introduced by quantization. Absolute value of two differences were added to get the luminance of each difference pixel. Each difference pixel was painted by thresholding. If the prediction error was above a threshold, and the reconstruction error below threshold, the difference was painted to blue. If the reconstruction error was above the threshold, and the prediction error below, the difference was painted to red. If both errors exceeded the threshold, the difference was painted to magenta as a result of color superposition (Figure 3.5).

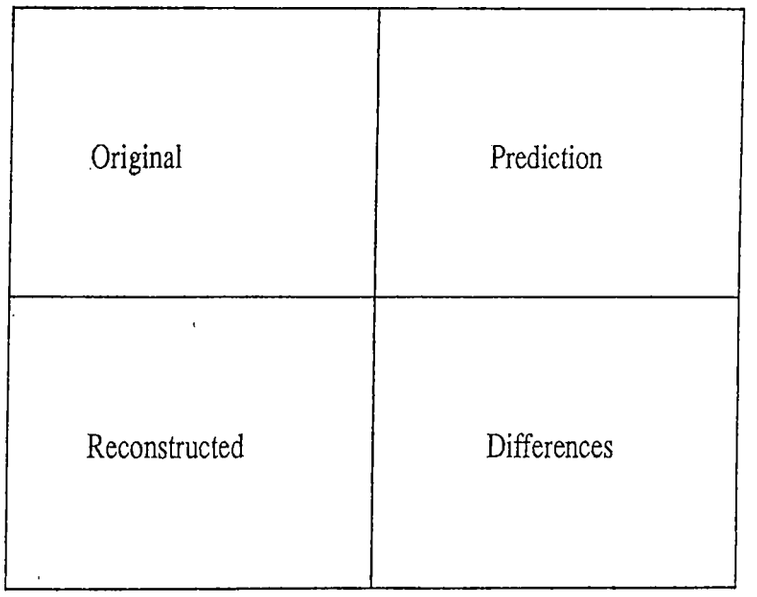


Figure 3.4: Visual performance detection

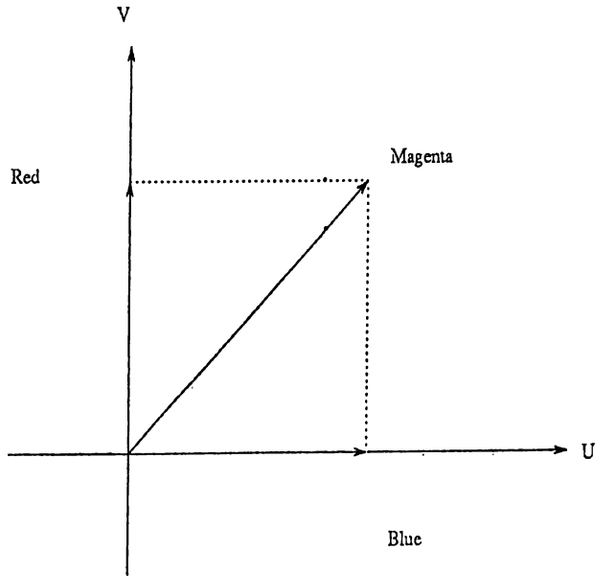


Figure 3.5: Color superposition

Chapter 4

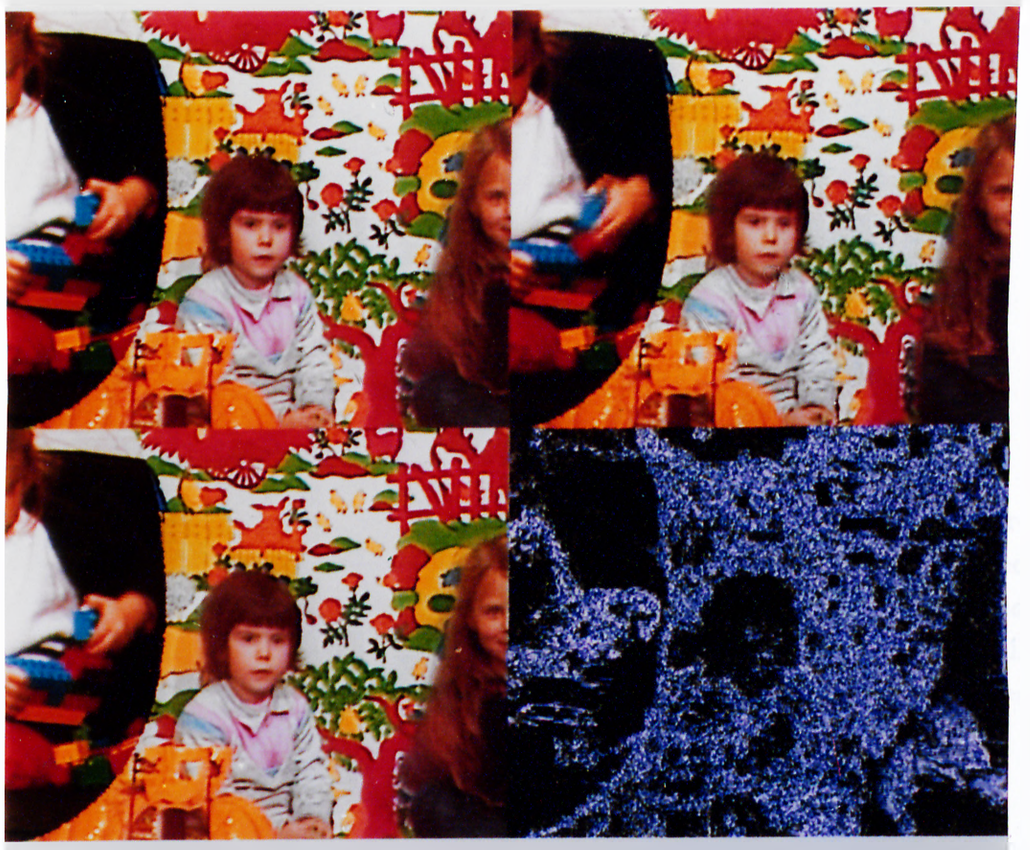
Results

4.1 Visual Performance

An HDTV quality picture with no visible artifacts was expected. It was observed that there were visible artifacts in the picture for quantization levels less than 4.

Figure 4.1 shows the program output as described in section 3.2.4. The first picture has a quantization level of 8, and there are almost no quantization errors introduced. This can be understood from the bottom-right part. There are seldom magenta dots, and no red dots. (In fact, existence of red dots would suggest system design or programming errors, because it corresponds to the events that there is a considerable quantization error while the data to be quantized is small.) The overall luminance level is low, which means low prediction error. The visual performance of the system is acceptably good in this case. There is no visible difference between the original and the recovered frames.

The second picture has a quantization level of 2, and there are considerably high amount of quantization errors. This can be seen easily by observing the reconstructed picture, in which there are visible artifacts, and by observing the bottom right part. The amount of magenta dots is quite high. The picture quality is even worse than the picture quality of a PAL system. The reconstructed image is *sandy* which is especially disturbing in low spatial-detail regions. *Sandiness* means the existence of a texture similar to the surface of a pile of thick sand. This *sandiness* is much stronger than the *blocking effect*, which was claimed by the proposers to be the 'most objectionable artifact'. In fact, blocking effect is not observed in the reconstructed image, possibly



Quantization level 8

Quantization level 2

Figure 4.1: Simulation output for the sequence Costgirls

	SNR			Prediction Gain		
SEQUENCE	Y	U	V	Y	U	V
Costgirls	142.5	397.6	368.5	41.3	175.2	148.1
Car	55.4	726.4	961.6	18.4	299.0	316.8
Cross	23.4	69.3	48.66	15.1	54.4	8.43

Table 4.1: Numerical results of the simulation

because it was shielded by the higher noise of *sandiness*.

The simulation results show that this system *failed* in the visual performance test. 17 people with no professional experience in the field were asked to evaluate the reconstructed picture quality. More precisely, they were asked to evaluate the similarity between the original and reconstructed pictures. All of them gave the same answer: "Good for first pictures (quantization level = 8), bad for second pictures (quantization level = 2).".

4.2 Numerical Results

Table 4.1 shows the simulation results for three different sequences. In the table,

$$SNR = \frac{\Sigma(Original\_Pixel)^2}{\Sigma(Original\_Pixel - Reconstructed\_Pixel)^2}$$

and

$$Prediction\_Gain = \frac{\Sigma(Original\_Pixel)^2}{\Sigma(Original\_Pixel - Prediction\_for\_the\_pixel)^2}$$

Figure 4.2 shows the quantization level versus picture number for the sequence *Costgirls*. Initial quantization level is 3, initial buffer fullness is 0%, and final buffer fullness is 81.4%.

Figure 4.3 shows the quantization level versus picture number for the sequence *Car*. Initial quantization level is 1, initial buffer fullness is 81.4%, and final buffer fullness is 91.4%.

Figure 4.4 shows the quantization level versus picture number for the sequence *Cross*. Initial quantization level is 0, initial buffer fullness is 91.4%, and final buffer fullness is 74.6%.

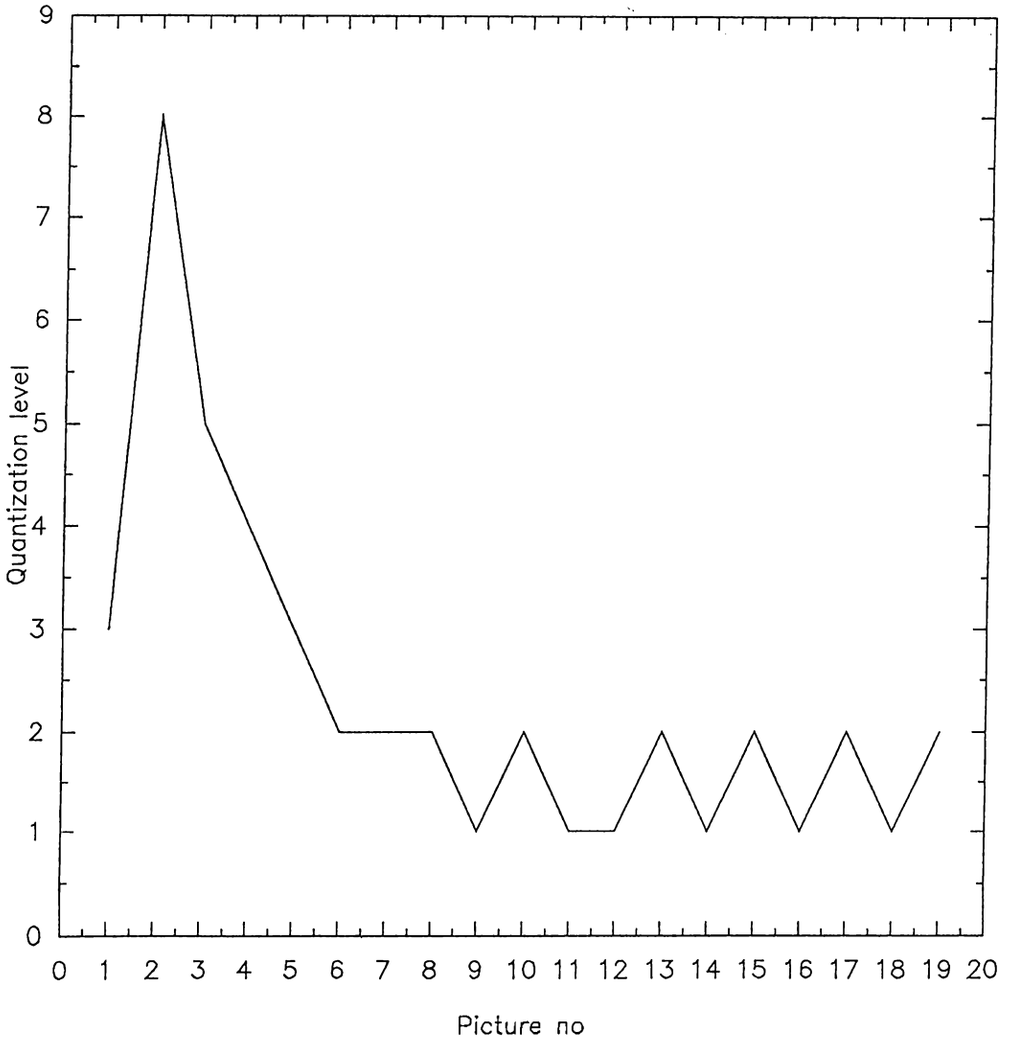


Figure 4.2: Quantization level versus picture number for *Costgirls*

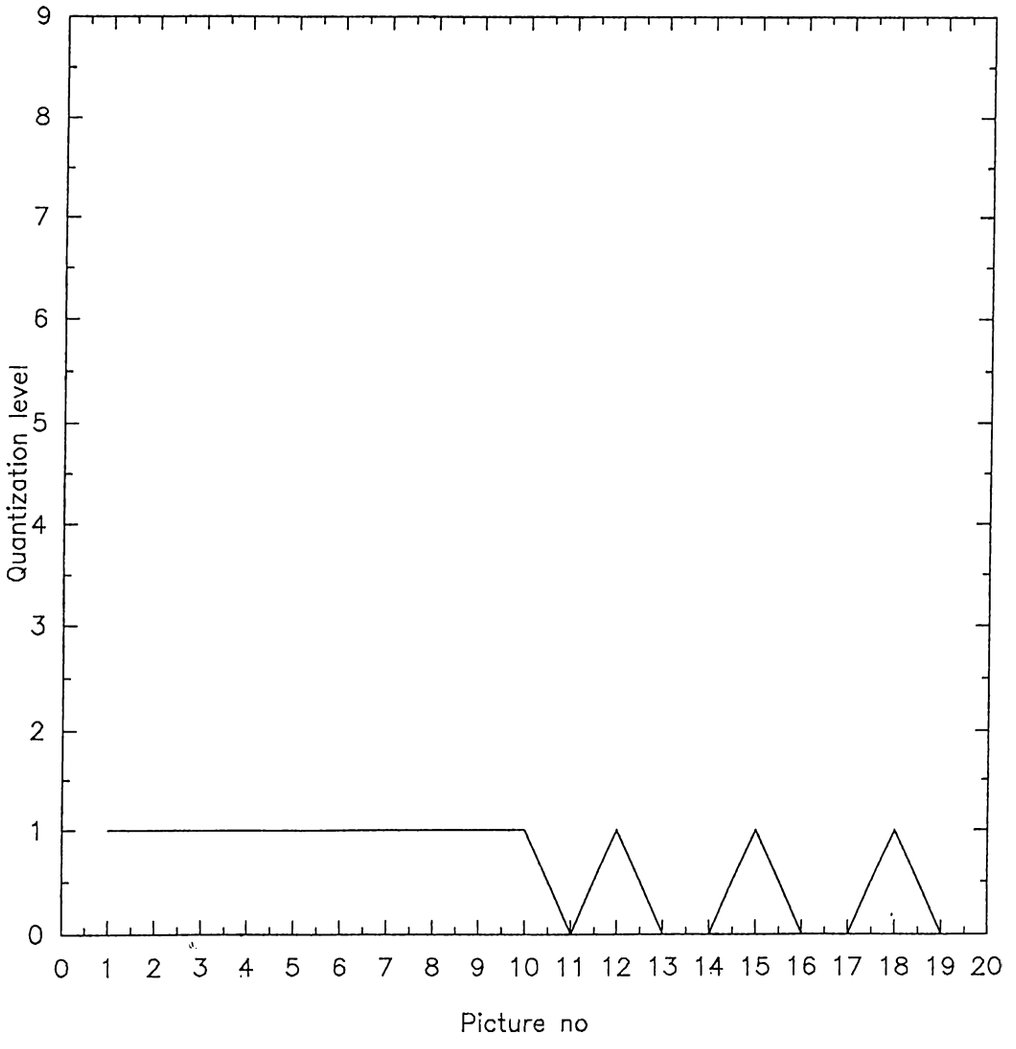


Figure 4.3: Quantization level versus picture number for *Car*

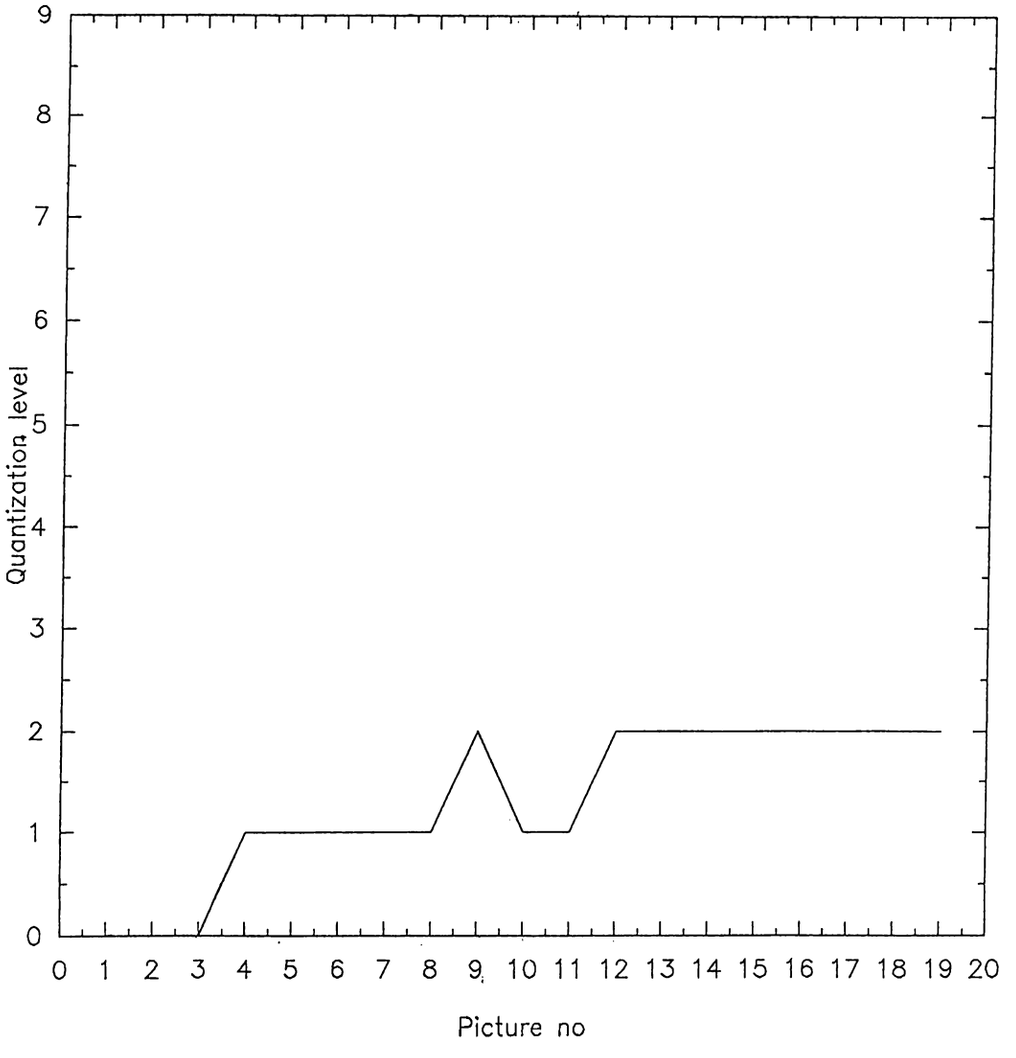


Figure 4.4: Quantization level versus picture number for *Cross*

The simulation results show that the steady state of quantization level occurs at low levels of quantization. The overall picture quality is not satisfactory, both in visual performance test and in SNR calculations. Though there is not a specified standard for acceptible SNR level of an HDTV quality picture, we can say that the calculated SNR figures are not very high. On the average, the noise amplitude is about one ninth of the signal level.

The prediction gain values, however, can be considered satisfactory (Note that the previous frame is used for prediction, and the previous frame does not have a high SNR). On the average, the prediction would save about 2.5 bits for the coding of an 8-bit word, if it was the only compression stage of the system (This result is obtained by taking the 2-based logarithm of the square root of the average prediction gain figures).

In fact, the visual performance test is more than the numerical results, because this system is designed to supply images for consumers, and what the consumer will see as an HDTV picture is more important for him than some numbers which do not make sense for him.

Chapter 5

Conclusions

In this study, the digital video encoder part of DigiCipher^{TM} HDTV system proposal was simulated.

There can be some improvements to this study. Though we have some conclusions now, they depend on some assumptions. There was a long list of assumptions, some of them being critical. In making assumptions, the general trend was followed. It can be interesting to repeat the simulations with several different sets of assumptions. The comparison of the results can show how important those assumptions are. It can also give some ideas which can be useful in designing similar image coding systems.

Another improvement can be the testing of *how typical* the test sequences are. Here, *typical* means the similarity between the statistical distribution of the amplitude–runlength events in the coding of the DCT blocks of the test sequences and that of the training sequences used by the system designers of DigiCipher^{TM}. The method for comparing the statistical distributions can be as follows: The lengths of the Huffman code words can be calculated by making statistical experiments on the image sequences used. These calculated lengths can be compared with the Huffman code word lengths given in the description sheets. The similarity between the lengths have a very close relation with the similarity between the statistical distributions. Another step may be using the calculated code word lengths instead of the ones given in the table for the simulation. This can give an idea about how good the system works for *typical* images.

Here are the conclusions based on the obtained results:

The results were disappointing in terms of picture quality. Both artifacts

observed in the recovered images and the poor signal-to-noise ratio of the reconstructed images suggest that the picture quality is even poorer than the picture quality of conventional TV systems. The most disturbing artifact which was observed is that the picture is *sandy* (The concept of *sandiness* is described in section 4.1). The *blocking effect*, which is the most objectionable artifact of systems involving blockwise DCT, was not even observed, possibly because there was much higher artifacts. One other reason may be the small DCT block size. It is not very likely that the blocking effect is avoided by a convenient quantization bit allocation, since the bit allocation method is rather simple.

The interesting point is that the picture quality in the demonstrations done by General Instruments is quite good. The main reason for the two contradicting results is the lack of clarity in the description of the system. It is not very surprising that the description sheets are not very clear, since there is competition. The new successful methods have to be hidden for getting an advantage in the competition. So it is possible that in DigiCipher^{TM} some effective data compression techniques are applied, but they are not mentioned in the description sheets.

This is not the only simulation of DigiCipher^{TM} system which was made by people working for other institutes or companies. The results of these simulations are not as optimistic as the demonstrations of General Instruments. This strengthens the idea of *hidden techniques.*

The quantizer block seems to be the *black box* of the video encoder. Among the sub-blocks of the video encoder, the quantizer block is the one which has been defined poorest in the description sheets. Furthermore, even the defined methods do not sound professional. There has been extensive study about bit allocation methods of the quantizers in last years ([7]). According to the description sheets, it seems that these studies are not taken into consideration during the design of the quantizer block. If we consider how big and important this project is, this is not very convincing. Considering that the motion-compensated prediction works well according to the simulation results, and that DCT and Huffman coding are well-known data compression methods, it is not illogical to say that the quantizer –as defined in the description sheets– is *guilty* for the failure of the system in these simulations.

References

[1] A. K. Jain "Image Data Compression: A Review" *Proc. IEEE 69*, no. 3 (March 1981): 349–389

[2] A. K. Jain, P. M. Farrelle, and U. R. Algazi "Image Data Compression" in *Digital Image Processing Techniques,* M. P. Ekstrom, ed. New York: Academic Press, 1984

[3] L. D. Davisson and R. M. Gray (eds.) *Data Compression. Benchmark Papers in Electrical Engineering and Computer Science,* Sroudsberg, Penn.: Douden Hunchinson and Ro, Inc. 1976

[4] DigiCipher^{TM} HDTV System, General Instruments, June 22, 1990

[5] Otto E. Mikkelä, Human Vision and Image Quality, EURASIP Digital Video Signal Processing and HDTV course in Tampere University of Technology, May 27–29, 1991

[6] Ahmed,N. , T.Natarajan, and K. R. Rao: Discrete cosine transform, *IEEE Trans.,* vol. c. 23, pp. 90–93, January, 1974

[7] H. Peng, H. Peterson, J. Morgan, and W Pennebaker, "Quantizing RGB Color Image Components in the DCT domain", *Proc. of the 1990 Image Processing and Visualization ITL,* May 1990

[8] D. A. Huffman, "A Method for the Construction of Minimum redundancy codes", *Proc. ERE,* pp. 1098–1101, September 1952

[9] ISO–IEC JTC1/SC2/WG11, Systems Chapter of ISO 11172 (MPEG) CD, 1990

[10] N. S. Jayant and P. Noll. Digital Coding of Waveforms. Englewood Cliffs, N. J.: Prentice–Hall, 1984