Implementation of realtime video compression, conform the MPEG standard, using a Transport Triggered Architecture

M.J.Q. van der Lem

Laboratory of Computer Architecture and
Digital Techniques (CARDIT)
Faculty of Electrical Engineering
Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft,
The Netherlands

March 21, 1997

Delft University of Technology
Faculty of Electrical Engineering

Type	:	Master's Thesis
Number of pages	:	96
Date	:	March 21, 1997

Lab./Dept.	:	Laboratory of Computer Architecture and Digital Techniques (CARDIT)
Codenumber	:	1-68340-44(1994)##
Author	:	M.J.Q. van der Lem
Title	:	**Implementation of realtime video compression, conform the MPEG standard, using a Transport Triggered Architecture**

Supervisor	:	Prof. Dr. Ir. A. J. van de Goor
Mentors	:	Drs. H. Corporaal

Abstract

The MOVE framework is the automatic design of Application Specific Instruction set Processors (ASIP). The goal of this research is to investigate the cost and performance of a MPEG (Moving Picture Expert Group) implementation using the MOVE framework. The MPEG encoder is a standard which compresses a video sequence into a bitstream. The design of ASIPs for the real-time MPEG implementation is processed in two steps. The first step is to design an MPEG encoder written in the high level language C. The MPEG encoder of the Software simulation group was used as a basis. The program produces a qualitative good MPEG bitstream, however the execution time of the program is far too slow to execute in real-time. To optimize the execution time, the motion-estimation and the DCT algorithms of the MPEG encoder program are replaced by optimized algorithms. The second step is mapping the MPEG encoder program into a Transport Triggered Architecture (TTA) using the tools of the MOVE framework. To improve the performance the MPEG program is split up into four threads which run on four separate processors. Further, the ability of using Special Function Units (SFU) are utilized to speed-up the execution time. The result is a design of a qualitative good MPEG encoder program which run on four designed ASIPs. The combination of the MPEG encoder program and the ASIPs perform an almost realtime compressing of a CIRR video signal.

MPEG multi-threading ASIPs

Contents

List of Figures

List of Tables

1 Introduction

One aim of the MOVE project of the CARDIT group at the faculty of electrical engineering at Delft, is a proposal for implementing the MPEG codec algorithm within the MOVE framework. The goal of this report is to give an evaluation of performance and cost, and a representation in a high level language (C) of the MPEG encoding algorithm. Hereafter an implementation for the MPEG encoding algorithm is proposed on a architecture level, by using the MOVE framework.

1.1 Research overview

During recent years more interest is shown in video applications, like multi-media, videophone and video conference. The video applications, store video signals on storage devices like hard disks and transmit them over communication channels. Using video signals this way two disadvantages exist, first the throughput and second the data capacity of these devices are lower than required for video signals. For example, a video signal of 720×576 pixels at a rate of 25 frames per second and 1.5 bytes per pixel requires a bit rate of 125 Mbit/sec. For video with a higher resolution of the frames an even a higher bitrate is required. This discrepancy is solved if the bandwidth of the digital video signal is reduced by compressing it. The MPEG (Motion Picture Experts Group) is a standard for compressing digital video signals into a compressed bitstream with a constant bitrate. The standard consists of the components shown in figure 1.1. The decoder compresses the video signal into a bitstream. This bitstream can be transported over a communication channel or stored on a storage device. The encoder decompresses the bitstream into a video sequence. This video sequence differs from the original sequence because the MPEG standard allows the information to be distorted.

Three important parameters of the standard are the bitrate (B) of the bitstream, the quality of the reconstructed video (Q_v) and the quality of the encoder (Q_e). The bitrate is the number of bits per second the encoder will produce, and send to the decoder over the communication channel. The quality of the reconstructed video (Q_v) is a measure for the difference between the original and reconstructed video signal. This difference can be calculated and expressed as a signal to noise ratio (SNR), but a better measurement is to use the Human Visual System (HVS) since the MPEG codec uses the HVS as a reverence for compression techniques. Between the bitrate (B) and the quality of the reconstructed video (Q_v) a proportional relation exists. The quotient of these parameters is a measure for the quality of the encoder (Q_e), which indicates how good the encoder compresses the video sequence. In formula notation this is $Q_e \Leftrightarrow \frac{Q_v}{B}$. Two other parameters which classify the encoder are the time it takes the encoder to compress a frame of the video sequence which is denoted as (T_d) (i.e. delay), and the number of video frames it compresses in a certain amount of time, denoted as T_e (i.e. throughput). For real-time video compression (T_e) is equal to the inverse of the framerate.

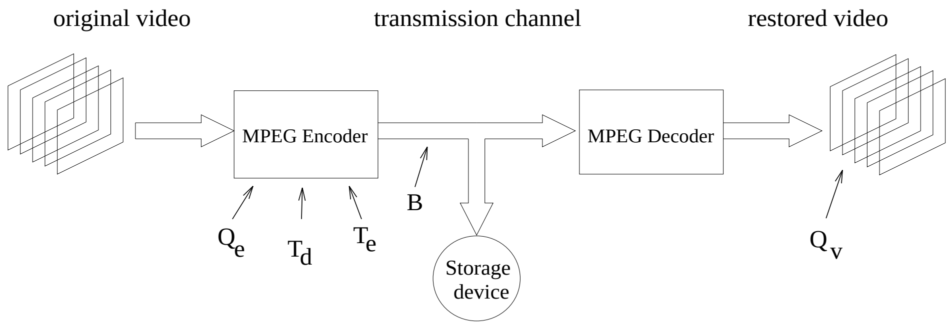

Figure 1.1: The MPEG encoder is used to compress a video sequence and transmit or transport it.

The MPEG standard uses several techniques to compress the video signal ;they are categorized as a lossy and a lossless compression. The lossless compression compresses the data without information loss. Some of these compression techniques are:

- Motion estimation.
- Runlength coding.

The second category is formed by the lossy compression technique. Here the data compression rate is high but after decompression the information of the data is distorted. Some lossy compression techniques used in the MPEG coding algorithm are:

- Sub-sampling Color component.
- Quantization of Discrete Cosine Transform (DCT) coefficients.

To execute these coding algorithms the encoder needs to performe an excessive amount of computations. The first step in this research is to reduce the number of these computations. Second these algorithms with these computations should be mapped into a VLSI design, using the MOVE framework. To implement the MPEG coding algorithm with the MOVE framework the algorithm must be written in a high level language. In this report the 'C' program of the **MPEG Software Simulation Group** was used as a basis, this program is called the **SSG MPEG** in this report. This software was written to produce a qualitative good MPEG bitstream (i.e. the encoder has a high Q_e), but was not designed for short execution time (i.e. the encoder has a lowT_e).

To use this program for a realtime implementation using the MOVE framework, the code optimized in such a way that the execution time was decreased significantly, with some reduction of the encoder quality (Q_e), which resulted in the Fast MPEG program. The speedup in execution time is achieved by utilizing the **fast Discrete Cosine Transform** and a **three dimensional recursive search motion estimation algorithm** (3D-RS), which was designed by [7]. Furthermore many other algorithms were optimized. The Fast MPEG program is designed using the procedure described in figure 1.2. The original program is evolved into the Fast program using the path of optimizing the source program, or by changing the machine description file to support special function units (FUs) used by the optimized Fast MPEG program. It should be noticed that there is no good criterion which optimice path to take. Most of the decision are made by the designer using common sense and intuition. The amount of time spent optimizing the HLL application took about 80% of all the research time.

If the Fast MPEG program and its machine description file are determined these are used by the MOVE framework to produce a VLSI layout. More information about the MOVE framework and the machine description file are given in chapter 6.

1.2 Report overview

The structure of this report is as follows: The next chapter describes the general properties of the MPEG coding algorithm and gives an overview of the specific MPEG encoding algorithm used in this report. In section 3 the profiling information of the original MPEG encoder program is presented. It is concluded that there are two major procedures which consume most of the execution time of the program. These procedures are the DCT and the motion estimation.

The time consuming DCT algorithm is discussed in chapter 2. First some general properties and optimized algorithms are summarized. In section 4.2.1 a one-dimensional and in section 4.2.3 a two dimensional fast DCT is proposed. Section 4.3 discusse the aspects of realizing the proposed algorithm in a low complexity architecture.

In chapter 5 the motion-estimator is discussed, in section 5.1 an overview of the motion-estimation algorithm is given. Section 5.2 describes the full-search motion estimation algorithm. This algorithm is used in the original Software Simulation Group MPEG encoder program. The disadvantage of this algorithm is that it does not produce true motion vectors and has an excessive computation time. In section 5.3 the 3D-RS motion-estimation is proposed. The 3D-RS produce true motion vectors and has a low execution time. In section 5.4 the aspects of implementing the 3D-RS algorithm in the

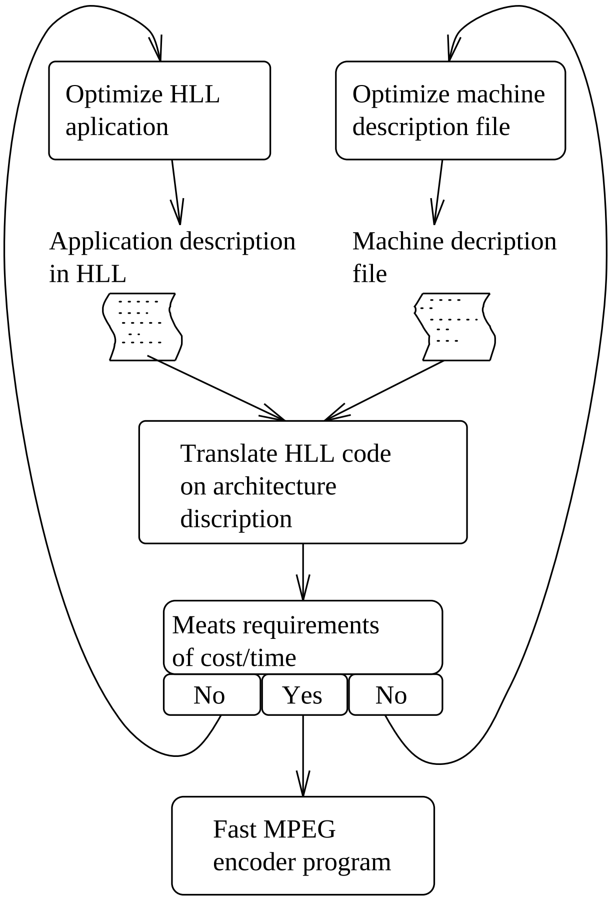

Figure 1.2: The design scheme of the MPEG encoder.

MPEG scheme is discussed and in section 5.5 the cost and performance of the discussed motion-estimators are presented.

In chapter 6 a short overview of the MOVE framework is presented, in section 6.2 the compiler of the MOVE framework is discussed. Section 6.3 discusses the "machine description" file which describes the architecture of a processor. In section 6.4 the scheduler is discussed. The scheduler maps the move code produced by the compiler onto a processor described by the machine description file. In section 7 the results of mapping the several versions of the MPEG encoding program onto different processors are presented.

2 The MPEG coding trajectory

The MPEG is a standard for compressing digital video signals and is designed by the ISO standard group. In [1] the MPEG coding is described in detail. The standard only prescribes the semantics of the output of the encoder, which is called the bitstream, so the algorithms to perform the (de)compressing are not part of the standard. Furthermore provides the standard many options which can be used to improve the quality of the encoder but are not necessary to bring off a MPEG compatible bitstream. The purpose of discribing the semantics of the bitstream is that the algorithms can be optimized by the designeri, and only those algorithms are implemented which are useful for a particular encoder. In the next sections more detailed information about compressing video signals and the MPEG coding algorithm is explained. In section 2.5 a specific MPEG algorithm that is used in this report is presented.

2.1 Description of video signals.

The last years more interest is shown in representing analogue signals in the digital domain. The advantage of digital signals is that the operations like transporting, storing, and processing can be more easily performed than analogue signals. Digital signals can also be stored and transported with no addition of noise. Some examples of products which use these advantages are, the compact disc and the DCC (Digital Compact Cassette). The compact disc uses the property that digital signals are stored easily in large quantities, while the DCC compresses the audio data into a lower bitrate so it is possible to record the data on a magnetic tape. Digital signals can also be used in moving pictures (e.g. in Video); only the bitrate of TV signals and the necessary storage capacity

Signal	frame size	Bits/sample	Frame rate	Bitrate	Storage/sec
			Hz	Mbis/sec	Mbyte
Telephone	-	8		0.064	-
CD audio	-	16		0.704	-
CIF	172*144	12	25	7.4	1
SIF	352*288	12	25	30.4	3.8
CCIR	720*576	12	25	124.4	15.6
HDTV	1920*1250	12	50	1440	180

Table 2.1: Bitrate of different signals before compressing.

are very high. As indicated in table 2.1 the bitrate for high definition television (HDTV) is about 22500 times the bitrate of telephone.

Signals representing moving pictures are three dimensional. If $V(x, y, t)$ is a moving picture signal then this signal can be interpreted as a two dimensional picture, the space (x, y) component which changes in time, the t component. In figure 2.1 the signal

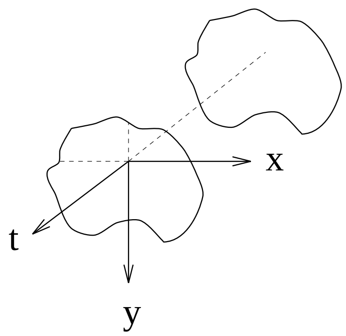

Figure 2.1: A video signal is a three dimensional signal.

$V(x, y, t)$ is drawn, for clarity the t component is shown only for two discrete times, but in reality the signal is continuous. Each of these three components of signal V have redundancy.

The three dimensional signal $V(x, y, t)$ is practically not used because it's difficulty to transport over communication channels. A practical solution is representing moving pictures as a one dimensional signal by sampling the three components of signal $V(x, y, t)$. First, the time component of $V(x, y, t)$ is not represented by a continious signal but only on some time intervals. This is possible because the HVS interprets pictures which are shown at a rate of at least 25 Hz as moving pictures. Second, the two dimensional picture component of $V(x, y, t)$ is divided into discrete intervals by representing the picture as a finite two dimensional array of picture elements. These arrays are scanned line by line, so the two dimensional picture is represented as a finite one dimensional data sequence, which result in a signal $V(x, y, t)$ represented by a one dimensional analogue signal, see figure 2.2. Now the video signal is represented as a one-dimensional signal, the analogue values are quantized to produce a digital video signal.

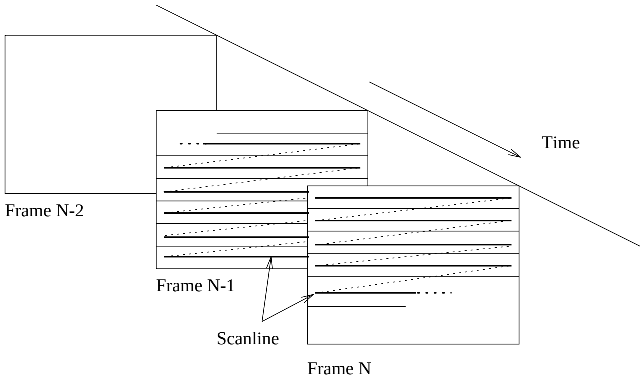

Figure 2.2: Video described by one-dimensional scanline

When you want to compress the high bitrate of a digital video sequence it is possible to compress only the one dimensional data sequence. A disadvantage of this technique is that only the one dimensional data sequence is compressed, since it is difficult to compress the y and t components of the one dimensional representation of the video signal. As a result only the correlation of the x component of the signal $V(x, y, t)$ is compressed which leads to a low compression ratio. A much better compression ratio is established if the video signal is compressed in three dimensions, which is used in the MPEG standard. The different compression techniques used in the MPEG standard, are divided into two classes, these are:

lossy compression With this process the irrelevant data is filtered out of the data sequence. This process is irreversible so some information of the data will be lost, but a high compression rate is established.

lossless compression This technique decorrelates the signal by removing the redundancy information. Most data sequences have some redundancy because some codes of bits occur more often then others. When representing the codes which occur often by less bits then the less occurrence codes a compression is established. With this method there exist a reverse operation so the original data can be restored.

Although the compressing techniques were divided into a lossy and a lossless technique, there is another orthogonal way to divide the compressing techniques, of a video signal, these are:

- The temporal compression
- The spatial compression

The **temporal compression** is the class that decorrelates the time component of the signal V. Correlation means that the possibilities of differences of a point in the frame is not high when the time is small, so this compression is classified as lossless compression. The temporal compression can be separated in two types. By the first type the whole previous frame is used as a prediction for the next frame. The second type identifies part of the frames which are moved between two frames, this technique is called motion estimation.

- Most of the time between two frames only a little scene-change takes place, as an example the background of a scene. MPEG coding will not send the new frame completely but only the difference between the old and the new frame. At the decoder the new frame is computed by adding, the received difference information, with the old frame.

- If there are moving objects in the scene the above method will not work efficiently because every time some part of the frame (i.e. an object in the scene) moves the old information of the space left by that object and the new position of the object should be transmitted. If you can identify the moving object and its displacements you can send only the displacement information to the decoder. This process of identifying moving objects in a scene is called Motion estimation and is also a temporal compression technique used in MPEG. For more information about motion estimation/compensation see [9]

The **spatial compression** in the MPEG coding is only used at one frame at the time. The spatial coding itself uses three compression techniques. Two compression techniques use the fact that some frame data is perceptive irrelevant for the HVS. The compression techniques are:

- *lossy compression* of the luminance/Chrominance blocks. The HVS is less sensitive for the Cb and Cr chrominance than for the luminance. The MPEG coding use this property to allocate more data space for the luminance than for the Cb and CR chrominance data. In figure 2.4. it is shown that there are 4 blocks for the luminance 1 for the CB and 1 for the CR chrominance blocks.

- *lossy compression* in the frequency domain of each block. The HVS is also less sensitive for high frequencies in each Cb, Cr and luminance block. The DCT transforms the time domain data block into a frequency domain block. Note that with this transformation no data compression is established only the low and high frequencies are separated. The quantizer (Q) is the element which reduces the data because it allocates more bits for the low and less for the high frequencies.

- *lossy compression* with the variable length coding. With the variable length coding a set of N bits with high occurrence rate are coded with less then N bits and low occurrence are coded with more then N bits.

In the next section some MPEG terminology and their algorithms for compression are presented.

2.2 MPEG overview.

The MPEG bitstream semantic consists of several layers:

- Video sequence

- Group of pictures (GOP)
- Picture (field or frame type)
- Slice
- Macroblock
- Block

Besides the Macroblock and Block layers, all layers are identified in the MPEG bitstream by unique start codes, so the decoder can re-synchronize after every slice if an error occurred in the transmission. The first layer is the video sequence layer, at this layer the information about picture size, framerate and bitrate are defined. The video sequence consists of a number of GOPs which consists of a number of pictures. The GOP provides a starting point for random access of the MPEG bitstream and always starts with an I-frame in send order (i.e. the order in which the pictures are transported over the transmition channel) and ends with an I or P picture in display order (i.e. the order in which the pictures are displayed), more information about the send and display order of a sequence is discussed later. The pictures in the sequence are divided into three types depending on the type of temporal compression technique used for that picture. The pictures types are:

- Intra coded picture (I)
- Predicted picture (P)
- Bidirectional predicted picture (B)

As shown in figure 2.3 an I-picture uses no temporal prediction, so there is no temporal data reduction. The I-pictures provide a random access point since there is no information needed from other pictures the reconstruct a I-picture. A P-picture is encoded as a reference to the past (in time) I or P-picture. The B pictures are encoded as a reference to the past and/or future I or P- picture. To reconstruct a B-picture the information from the I or P-picture from the past and future is necessary. Before the decoder can reconstruct the B-picture it should already have the future I or P-picture. To solve this discrepancy the sequence is rearranged by the encoder, so that the P-picture is send to the decoder before all B-pictures which reference to this P-picture. For example the display order of figure 2.3:

$$I_1\ B_2\ B_3\ P_4\ B_5\ B_6\ P_7 ...$$

is rearranged into:

$$I_1\ P_4\ B_2\ B_3\ P_7\ B_5\ B_6 ...$$

The pictures are build up of slices and these consists of a number macroblocks. The macroblocks are formed by 16×16 pixels, every macroblock consists of four luminance, one Cb chrominance and one Cr chrominance blocks which makes a total of six blocks per macroblock. The blocks consist of 8×8 elements. For the luminance blocks every

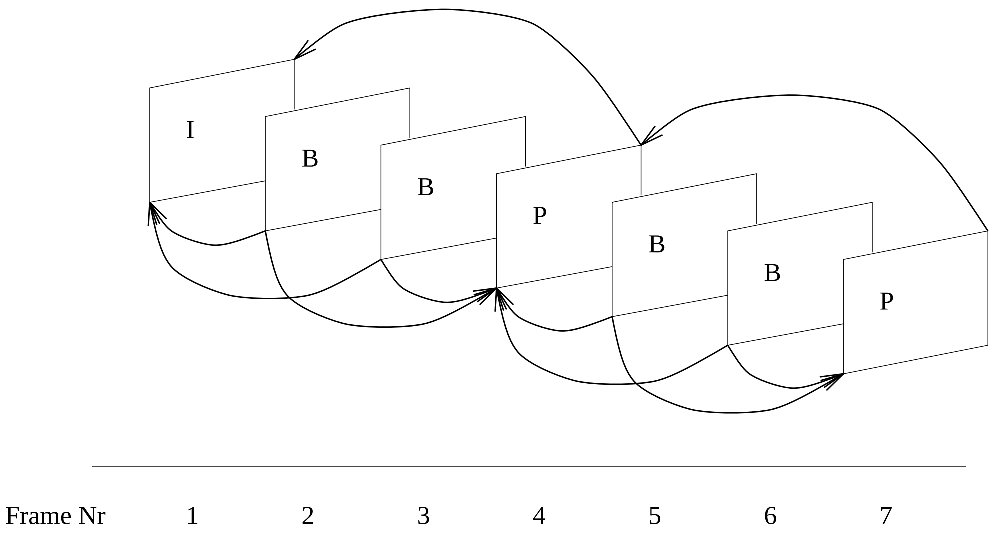

Figure 2.3: A sequence of pictures in display order.

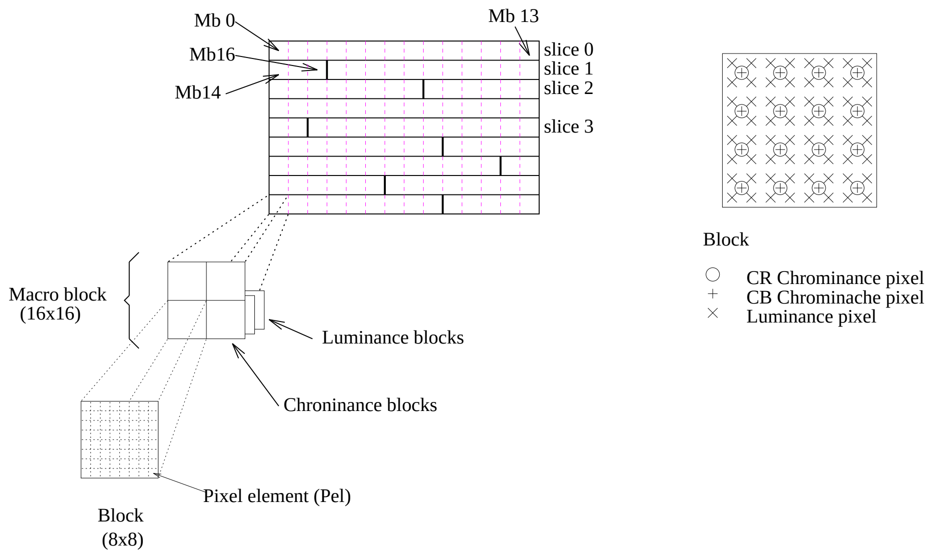

Figure 2.4: Structure of a picture, slices, macroblocks and blocks.

element corresponds with one pixel, for the Cb and Cr every element corresponds with four (2×2) pixels. Figure 2.4 gives an overview of the structure of a frame, here the frame consists of 14 horizontal and 9 vertical macroblocks. Since every macroblock consists of 16×16 pixels the picture size is 224×144 pixels. The frame consists of 8 slices which should not have the same number of macroblocks, as in our example the first slice start at macroblock 0 and ends at macroblock 16.

Motion compensation The macroblock is used as the entity to performs the motion estimation and compensation at. The motion estimator finds the motion vectors of two pictures (reference, source). If an appropriate motion vector is found it is send together with the difference between the reference and source frame to the decoder. At the decoder side the information is then used to reconstruct the source frame by copying the macroblock pointed by the motion vector of the reference picture to the source picture. The difference of the two macroblocks are added to the copied macroblock yielding the source macroblock (see figure 5.1 for more information). For a B picture type there is another motion compensation type, here not one but two reference frames may exists. Now it is possible to predict a B picture from past and future reference, which improves picture quality.

Macroblock type decision In the MPEG standard the macroblock can have different types but are closely related to the type of picture it belongs to. For every macroblock in a I picture the macroblock type is intra. The macroblocks in a P picture are in general coded as forward predicted but it's also possible to predict it as an I frame or as a null motion prediction or skipped. For a B picture the possible prediction types are intra, skipped, forward, backward and interpolated. The different macroblock types are discussed next in more detail:

Intra macroblocks are blocks which are send to the decoder without motion vectors, the picture information is independent of other pictures.

Null motion compensated macroblock are blocks were the picture information is a difference information to the macroblock of the previous picture. Since the motion vectors are zero these vectors are not send to the decoder.

Skipped macroblocks are like null motion compensated macroblocks except that after quantization there is no information left to send to the decoder.

Forward compensated macroblocks should add their picture information with a macroblock pointed to by the motion vector of the previous picture.

Backward compensated macroblocks should add their picture information with a macroblock pointed to by the motion vector of the next picture.

Interpolated compensated macroblocks should add their picture information with two macroblocks interpolated which are pointer to by the forward/backward vector in the previous/next picture.

In table 2.2 the possible macroblock types for every picture type are given. The macroblocks types described above divides the picture information into motion vectors and (difference) picture information. This (difference) picture information furthermore compressed by spatial compression technique.

Spatial compression The spatial compression techniques used by the MPEG coding scheme uses the Discrete Cosine Transform for transforming the intra or inter picture

Signal	Intra	Skipped	Null motion	Forward	Backward	Interpol.
I picture	√					
P picture	√	√	√	√		
B picture	√	√		√	√	√

Table 2.2: Possible macroblock type coding for the I-P-B picture types.

information from the spatial to the frequency domain. Since the HVS is less sensible for the high frequencies these are quantized coarse for high frequencies. Another advantages of the DCT transform is that the possibility for high frequencies to exists in the video signal is lower than for low frequencies. The consequence is that the probability distribution for the high frequencies is much lower and are more likely to have a value of zero. The zigzag scan of the block is used to start with the high and end at the low frequencies. All the elements of the zigzag scan are transmitted to the decoder using the run amplitude technique, and are entropy coded .

Figure 2.5 gives a possible architecture of a compression algorithm of MPEG. The scheme can by divided into three parts which are discussed separately.

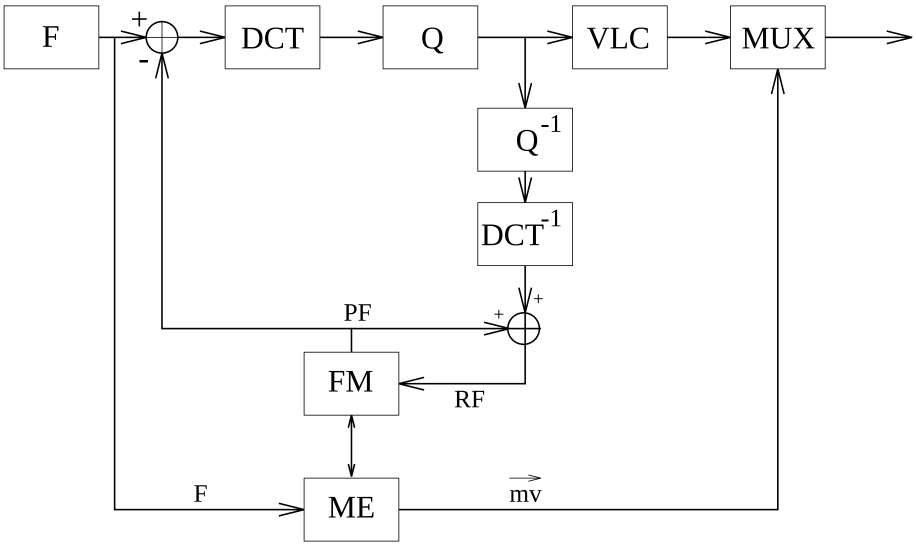

Figure 2.5: An overview of the MPEG encoding algorithm.

The forward path The first part of the scheme is the forward line of the input to the output of the encoder. Here the input frame denoted by F is first transformed by the DCT. The redundant output of the DCT is then is suppressed by the quantizer (Q) and is then coded by the variable length coder (VLC). The MUX performs the addition of extra motion compensation vectors which will be discussed later. This part of MPEG only compresses the spatial redundancy.

The inner loop The second path of the encoder is the inner loop with the inverse DCT and the Frame Memory (FM). This loop is used for plain predictive coding (= using the previous frame as the prediction for the current frame). The quantized coefficients, Q(DCT(F)), are transformed back to the pixel domain and added to the pixel data of the previous coded and re-transformed frame. Together they reconstruct the current frame that will serve as the prediction frame for the next frame. These reconstructed frames $DCT^{-1}Q^{-1}(Q(DCT(\triangle F)))$, are stored in frame memoryi. A similar reconstruction is performed by the decoder, where the original frames are not available because the transmitted Q(DCT(F)) can not be transformed back to yield the original frames F as the quantization Q is an irreversible operation. By using the reconstructed frames for the prediction this problem is solved as the decoder can produce these predictions Q(DCT(F)) as well.

The outer loop To perform motion compensated predictive coding with this encoder, the original frames F are used by a motion compensator. Using the reconstructed reference frames that are stored in the frame memory this motion compensator determines the motion vectors and produces the related prediction block. These blocks are now used to construct the prediction frame. The vectors are multiplexed with the VLC coded coefficients to produce the coded bitstream.

2.3 MPEG-2 extensions

The MPEG-1 standard is designed for audiovisual coding of digital storage media like CD-rom. MPEG-2 is an extension to the MPEG-1 standard and is designed to support higher bitrates, interlaced video and other tools to improve the quality of the encoder. These MPEG-2 extentions are not necessary to produce a correct MPEG-2 bitstream. The MPEG-2 extentions should only be implemented if they are usefull for the encoderi (i.e. if the video input of the encoder is only non-interlaced the interlaced extensions of the MPEG-2 are thus not neccesary). This section discusses the aspects of interlaced video and the compatibility with MPEG are discussed.

Interlaced video sequence The input sequence of pictures of a MPEG-2 codec are progressive or interlaced. If the sequence is progressive the MPEG-2 codec could handle these sequence just as the MPEG-1 codec. In interlaced sequences one picture consists of two fields: the even field consists of the even-scanline and the odd field holds the odd-scanline.

The MPEG encoder can interpret this interlaced sequence as a progressive sequence by merging the odd and even interlaced fields, or by coding the odd and even fields separately (see figure 2.6). The advantage of merging the fields is that a high compression rate is established if there is a small motion in the sequence. This is possible because when there is a small motion the difference in the scanlines due to the time difference of the even/odd lies is likely to be small. However if there is a lot of motion in the scene the are bad motion artifacts that stem from the fact that moving objects are in different places in the two fields. The advantage of separate coding of the odd and even fields is that the problem of the bad motion artifacts is solved, but the compression of the sequence is worse since the redundancy between the scan-lines of the fields are lower than for frames, due to the greater temporal difference. The best solution is to compress the sequence using the merged odd and even fields if there is not much motion in the sequence, and using the compressing the odd and even fields separately if there is much motion.

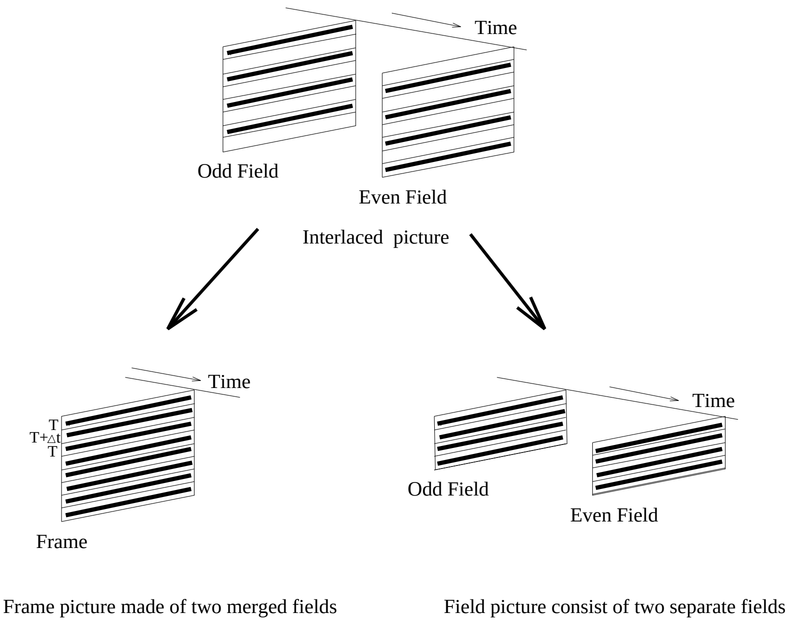

Figure 2.6: The encoder could interpret interlaced sequence by two ways.

Level and Profile The profile classifies the MPEG encoders/decoders by the algorithms which they support. The profiles in the MPEG standards are:

Simple. The simple profile is the same as the MAIN profile, only the B-pictures are not supported.

Main. The main profile is used by 95% of the applications and is discussed in this chapter.

Main+. The main+ is like the main profile with spatial and SNR scalability.

High. The High profile is like the main+ profile and the 4:2:2 macro block is used.

The levels in the MPEG standard classify the limits of the parameters supported by the encoder/decoder. The levels in the MPEG standard are:

low support CIF and consumer tape equivalence. The picture dimensions are 352 x 240 pixels at 30 frames per second (fps). This level is compatible with the MPEG-1 Constrained Parameters Bit-streams.

Main The main level meets the needs of the majority of users. The supported picture dimensions are 720 x 480 at 30 fps or 720 x 576 at 25 fps.

High 1440 Support high definition TV parameters like 1440 x 1152 at 30 fps.

High Is used for studio quality and high bitrate.

Audio layer The audio layer of the MPEG-2 standard describes the standard for low bitrate multichannel audio coding. The audio coding will support up to five full bandwidth channels for the left, right, center, and two surround. The MPEG-2 Audio multichannel coding Standard will provide backward-compatibility with the existing MPEG-1 Audio Standard (ISO/IEC IS 11172-3). MPEG-2 audio attempts to maintain as much compatibility with MPEG-1 audio syntax as possible, while adding discrete surround-sound channels to the original MPEG-1 limit of 2 channels (Left, Right or matrix center and difference). The main channels (Left, Right) in MPEG-2 audio will remain backwards compatible, whereas new coding methods and syntax will be used for the surround channels.

System layer MPEG-2 Systems Standard specifies coding formats for multiplexing audio, video, and other data into a form suitable for transmission or storage. There are two data stream formats defined: the Transport stream, which can carry multiple programs simultaneously, and which is optimized for use in applications where data loss may be likely, and the Program stream, which is optimized for multi-media applications, for performing systems processing in software, and for MPEG-1 compatibility.

Both streams are designed to support a large number of known and anticipated applications, and they retain a significant amount of flexibility required for such applications. The Transport Stream is well suited for transmission of digital television and video telephony over fiber, satellite, cable, ISDN, ATM, and other networks, and also for storage on digital video tape and other devices.

The Program Stream is similar to the MPEG-1 Systems standard (ISO/IEC 11172-1). It includes extensions to support new and future applications. Both the Transport Stream and Program Stream are built on a common Packetized Elementary Stream packet structure, facilitating common video and audio decoder implementations and stream type conversions. This is well-suited for use over a wide variety of networks with ATM and alternative transports.

2.4 The SSG MPEG encoder implementation.

The SSG MPEG encoder program is written by the *MPEG Software Simulation Group*. The software is written in the high level language C. The program now converts a video sequence into a MPEG-2 bitstream compliant with ISO/IEC 13818-2 DIS [1] (MPEG-2). The goal of the Software Simulation Group is to evolve this version of the MPEG-2 video encoder program to become ISO/IEC 13818-5 Software Simulation of MPEG-2 Systems, Video, and Audio. Some important futures supported by the SSG encoder are:

- The encoder is based on the MPEG-2 test model 5 (TM5) rev. 2.

- The encoder produce a constant bit-rater. The quantization is adjusted per macroblock to meet the required bit-rate.
- Progressive and interlaced video are supported by the encoder. The encoder determines if an interlaced should be frame or field coded.

The current limitation of the encoder are yet:

- The encoder can not produce scalable extensions.
- Check if the number of bits generated per macroblock does not exceed the maximum limit.
- Special low delay mode rate control.
- Variable bit-rate control to produce a constant quality bitstream.
- The control rate is not adapted to a scene change in the video

The reason that the SSG MPEG program is used for this research is that it is qualitative good MPEG encoder. The second reason is that at the start of this research it was the only available non commercial MPEG-2 encoder. In the next section the implementation of the SSG– MPEG encoder are specified. The SSG– MPEG is based on the SSG MPEG and it support a subset of the SSG MPEG implementation.

2.5 The SSG– MPEG encoder implementation.

The SGG– is the MPEG encoder used as a basis in this report. The SSG– MPEG program is a subset of the SSG MPEG encoder implementation. In the SSG– MPEG the interlaced features of the original SSG MPEG are both supported. The reason is get a better overview of the MPEG encoding algorithm. Further the parameters of the SSG MPEG encoder are fixed. Some of these fixations are:

- The encoder does not support a user defined quantize matrix.
- Except for the first all GOPs are closed.
- The GOP size has a width of 12 pictures, the I-P distance is 3 an GOP contains thus a IBBPBBPBBPBB... sequence.
- For the quantizer only the non-linear type is supported.
- The encoder produces main level and profile bitstream.
- The aspect ratio of the pixel elements is set to square.
- The color primaries, transfer characteristics and matrix coefficients are fixed.

3 Profile information of the SSG– MPEG program

The SSG– MPEG program is written by **MPEG Software Simulation Group** in the high level language C. The SSG– program provides an encoder which handles many features of the MPEG 2 standard, like constant bitrate streams and progressive/interlaced video. The current limitations of the encoder software are scalable extensions, bitrate control for scene change and variable bitrate control. In this section some characteristics of the SSG– MPEG encoder are given by monitoring the execution time distribution of the algorithms. First the test video sequences used as input for the MPEG encoder are described.

3.1 Test sequences

The input test sequences used for the encoder are dived into two types, the first type is a small CIF and the second is a CCIR size. An overview of the different pictures are given in table 3.1. The *HPRL-car* is a sequence were a car is driving in front of a building. The camera itself is standing still and is rotating to follow the driving car. This sequence contains a large horizontal motion. The *flower garden* is a sequence were the camera is moving and is filming a garden with a house. The flowers in the foreground contains high frequency information which makes it difficult to compress. The camera is panning form left to right, between the house and the camera a tree is standing and is moving relative to the houses in the background. The *tennis* sequence shows a man playing table tennis. The camera is zooming out and the moving table tennis ball makes it difficult to motion estimate the sequence. Furthermore, the pictures contains less high frequencies. The *Dvsteeny* is a close-up of a girl which is turning her head. The *Mobile* sequence is a

Name	CIF	frame size	motion
HPRL car	CIF	352*288	high
Tennis	CIF	352*288	high
Flower garden	CIF	352*240	medium
Dvsteeny	CIRR	720*576	medium
Mobile	CIRR	720*576	slow
Susie	CIRR	720*576	slow

Table 3.1: Characteristics of the test sequences used.

train driving on a track and pushing a ball in front off him, the camera is following the movement of the train. The *Susie* is a close-up off a girl telephoning and turning her head. This sequence is a merge interlaced sequence.

3.2 Execution time distribution

Although the quality of the MPEG-2 bitstream produced by the SSG– encoder is good, the execution time is too high to compute in real-time. To reduce the execution time first some characteristics of the SSG– program are measured. In table 3.2 some execution times of the encoder are given. The encoder compresses 25 frames of the Mobile sequence into a MPEG bitstream. The program is compiled with the GNU c-compiler and executed on a Silicon Graphics Irix (SGI) computer at 100 MHz, with a SPECint92 of 62.8.

There are four test executed, two tests with optimized code (using CC -O2) and two with different search windows (12 and 3 pixels). The number of pixels for the search window is an indication how good the motion-estimation will perform, for more information about the motion-estimation used in the SSG– program see chapter 5. Since the goal of this report is to process the MPEG encoding algorithm in real-time, the test to compress 25 frames should execute in 1 second. The speed-up off the optimized encoder program with a window size of 12 should thus be about 1000. If the program is executed on a application specific processor chip which exploits the instruction level parallelism which is about 2-5 for normal programs, the encoder should thus have a speed-up of about 300. The use of the MOVE framework also gives a speed-up if special function units (FUs) are used, see chapter 6 for more information. It should be noticed that these numbers are inaccurate and only presented to give an indication about the speed-up to established.

From the information given in table 3.2 it is observed that increasing the window size strongly increase the execution time, which indicates that the motion-estimator plays an important role in the execution time of the encoder.

To measure the influence of the different compression algorithms of the MPEG encoder, the SGG– program is executed with the pixstats option. This option measures many statistically characteristics of a program compiled and executed on a SGI computer. For compiling the program the compiler of the SGI computer is used without the optimize (-O2) option. The pixstats option gives information like the time distribu-

optimized	search window	execution time
no	12 pixels	2361 sec.
no	3 pixels	550 sec.
yes	12 pixels	935 sec.
yes	3 pixels	237 sec.

Table 3.2: Execution time for compressing 25 frames with the SSG– encoder program executed on the SGI computer.

tion of the procedures and the instruction distribution of the program. The information about the distribution of the procedures is given in figure 3.1. From this figure becomes clear that the motion estimation(ME) is a major bottleneck of the program. A second procedure which consumes much execution time is the DCT.

In figure 3.2 the instruction distribution used by the encoder program which compress 25 frames with two different window sizes for the motion-estimation is drawn. The instruction set is divided into 5 sets. The ALU set are all alu-instruction, the LOAD and STORES are the loads respective stores from/to memory, the BRANCH are the conditional branches. The NOP are instruction which no operation; these instructions are used to fill branch delays (about 11% of all the NOPs in the program). On SGI the loads having dependencies with the next instruction are followed by a NOP, in contrast to other CPU's were a stall is generated by the CPU itselfr. The NOPs used to handle the load dependency is about 85 % of all the NOPs used in the program. In the next sections improvements are proposed the minimize the execution time of the SSG– encoder program.

3.3 Improvements for the SSG– MPEG encoder

In the previous chapter some characteristics are given for the SSG– encoder. In this section improvements are proposed and are elaborated in the rest of this report. Two major bottle-necks of the SSG– program are discussed in the next sections. The solution for these bottle-necks are given in the chapters 4 and 5.

Motion-estimation From figure 3.1 it is clear that the motion-estimation (ME) algorithm consumes most of the execution time of the encoder program. In order to improve the execution time the ME algorithm will be optimized. The most important part of the ME algorithm is the *dist1* procedure; for the 12 pixel test this procedure uses 84.3% of the total program execution time and hence it is almost completely responsible for the execution time of the program.

The algorithm compares a set of candidate macroblock with one source macroblock, and finds one macroblock of the set of candidate macroblocks which has the smallest mean absolute error (MAE) (see equation 5.2 for more information about block-

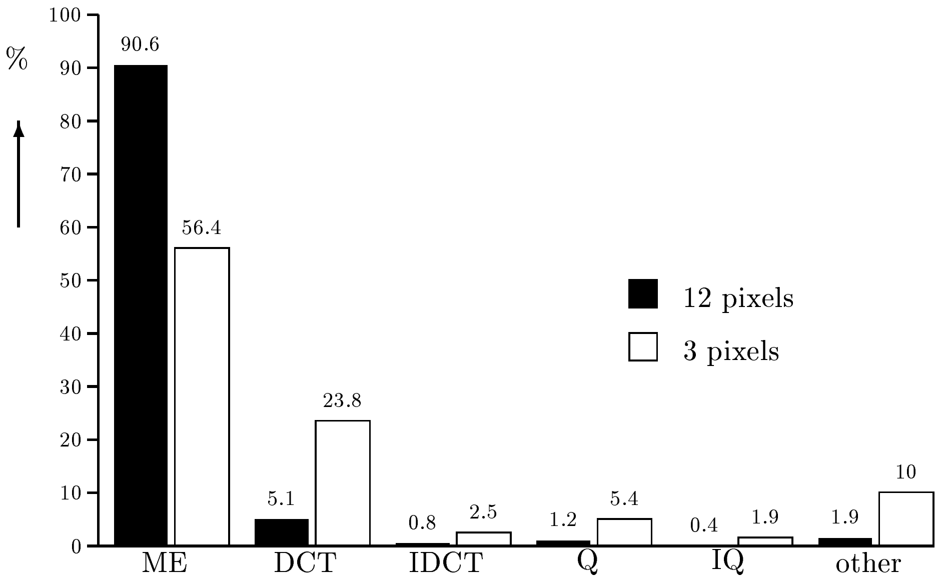

Figure 3.1: The execution time distribution of the SSG– encoder.

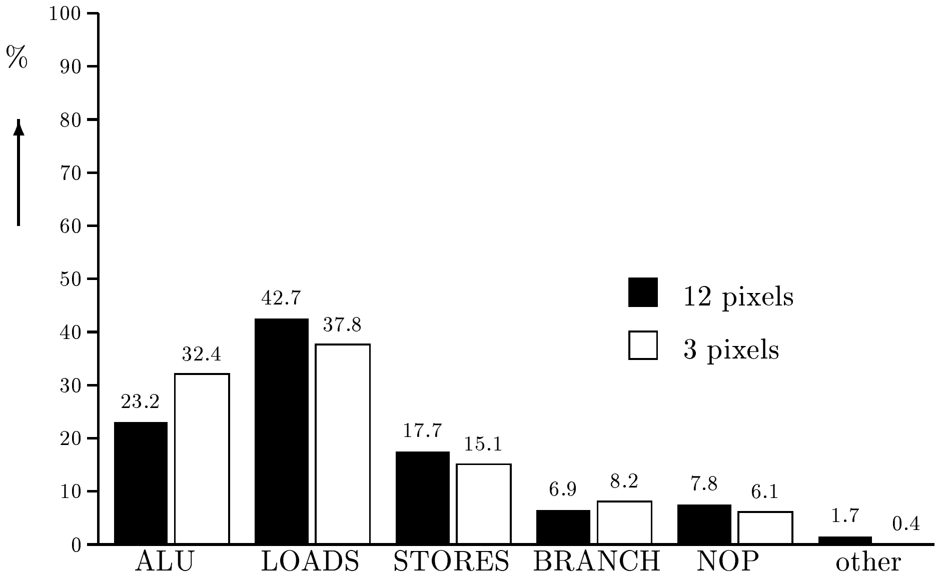

Figure 3.2: The instruction distribution of the SSG– encoder.

matching). The most considerable loop of the *dist1* procedure is printed in figure 3.3. This loop computes the absolute difference between the luminance parts of two macroblocks of 16×16 pixels. The variable s in the source code holds the sum of all the differences of the pixels of the macroblocks. After a row of 16 pixels are computed the algorithm checks if the sum of the difference is greater than the variable distlim (see line 21). This variable holds the smallest MAE of the previous compared macroblocks. Since

```
LINE    CODE

1       for (j=0; j<16; j++)
2       {
3         if ((v = p1[0]  - p2[0])<0)  v = -v; s+= v;
4         if ((v = p1[1]  - p2[1])<0)  v = -v; s+= v;
5         if ((v = p1[2]  - p2[2])<0)  v = -v; s+= v;
6         if ((v = p1[3]  - p2[3])<0)  v = -v; s+= v;
7         if ((v = p1[4]  - p2[4])<0)  v = -v; s+= v;
8         if ((v = p1[5]  - p2[5])<0)  v = -v; s+= v;
9         if ((v = p1[6]  - p2[6])<0)  v = -v; s+= v;
10        if ((v = p1[7]  - p2[7])<0)  v = -v; s+= v;
11        if ((v = p1[8]  - p2[8])<0)  v = -v; s+= v;
12        if ((v = p1[9]  - p2[9])<0)  v = -v; s+= v;
13        if ((v = p1[10] - p2[10])<0) v = -v; s+= v;
14        if ((v = p1[11] - p2[11])<0) v = -v; s+= v;
15        if ((v = p1[12] - p2[12])<0) v = -v; s+= v;
16        if ((v = p1[13] - p2[13])<0) v = -v; s+= v;
17        if ((v = p1[14] - p2[14])<0) v = -v; s+= v;
18        if ((v = p1[15] - p2[15])<0) v = -v; s+= v;
20
21        if (s >= distlim)
22           break;
23
24        p1+= screenwidth;
25        p2+= screenwidth;
26      }
```

Figure 3.3: A part of the C source code of dist1.

the smallest MAE should be found the loop can stop if the partial computed MAE is already greater than the variable distlim. This technique together with the loop-unrolled lines 3 - 18 indicates that this loop is already optimized.

There are some other optimizations possible like pixel sub-sampling but this gives a maximum speed-up, with respect to the quality, of about 8. The ME algorithm is also speed-up if the number of MAEs computed is decreased. This is already shown by decreasing the window size from 12 to 3 in the previous tests. The disadvantages is that decreasing the search window size also decreases the quality of the motion-estimation is decreased, and hence the quality of the encoder.

From the previous discussion it is concluded that optimizing the SSG– ME algorithm gives not the necessary speed-up to solve the motion-estimation bottle-neck, so the complete algorithm should be replaced with a much faster algorithm with a comparable quality. In the literature some motion estimations are known which produce good quality and have a fast execution time. The algorithm which will be implemented in the final encoder is the three dimensional recursive search (3D-RS) [7]. The 3D-RS algorithm and the full-search algorithm which is used by the SSG– encoder are discussed in detail in chapter 5.

Discrete cosine transform The execution time distribution of the DCT algorithms is about 5 to 24 percent, depending of the window size of the motion estimation, of the

```
LINE    CODE

1       void fdct(block)
2       short *block;
3       {
4           int i, j, k;
5           double s;
6           double tmp[64];
7
8           for (i=0; i<8; i++)
9             for (j=0; j<8; j++)
10              {
11               s = 0.0;
12               for (k=0; k<8; k++)
13                 s += c[j][k] * block[8*i+k];
14               tmp[8*i+j] = s;
15              }
16
17          for (j=0; j<8; j++)
18            for (i=0; i<8; i++)
19              {
20               s = 0.0;
21               for (k=0; k<8; k++)
22                  s += c[i][k] * tmp[8*k+j];
23               block[8*i+j] = (int)floor(s+0.499999);
24              }
25        }
```

Figure 3.4: The C source code of the fdct procedure.

total execution time of the program. If the ME is optimized the relative presence of the DCT will automatic increase. The figure 3.4 gives the source code of the DCT algorithm. The algorithm computes the two-dimensional DCT using the row-column method (for more information about the DCT see chapter 4). The row-column method computes the DCT by first computing all the rows (line 8-15 of the source code) and second compute all the columns (line 17 - 23). The number of operations to compute this implementation of the two-dimensional DCT includes 1024 multiplications and 1024 additions, which is easily extracted from the source code.

The computations of the DCT can be reduced when optimized algorithms are used. In the literature many optimizations of the DCTs are known. In [6] more information about these optimized DCTs are given. Further, the variables used in the SSG– routine are floating-point numbers, and the time to compute floating point operations takes longer when compared to integer operations. In section 4.2 an optimized DCT algorithm is proposed which needs less operations (111 multiplications and 466 additions) and uses 16 bit integer arithmetic to compute the two dimensional DCT.

Inverse discrete cosine transform The inverse discrete cosine transform (IDCT) used in the SSG– program is optimized using the DCT of Chen-Wang [17]. The two-dimensional IDCT is computed with the row-column method which separates it into 16 one-dimensional transforms. These one-dimensional transforms are then computed using the optimized DCT of Chen-Wang, where the IDCT is easily derived from the DCT by inverting the flowgraphs. Although the SSG– IDCT yields good execution performance the algorithms requires a 32 bit arithmetic. In the final MPEG encoder program another IDCT is implemented which uses 16 bit arithmetic. Although the proposed IDCT algorithm uses 16 bits arithmetic, which decrease the precision the proposed IDCT, it still passes the IEEE Standard 1180-1990 accuracy test for IDCT specifications.

Other optimizations cwWithin the remainder of the SSG– program several optimizations are used to speed- up the execution time, these however are not discussed further in this report.

Definitions and properties of the DCT 4

In signal theory it is shown [11] that most signals have redundancy because not every amplitude of the signal has the same probability to occur in the signal, an example is a Gausian signal. Another redundancy of a signal is that neighbour samples of a pixel are highly correlated. Transformations can be used to decorrelate the signal. In the next section the DCT (which is a transform) is explained. In section 4.2 an optimized Fast DCT is presentedr. First an one dimensional DCT is proposed which is used by the proposal for a two-dimensional DCT. The final section discusses some aspect of implementing the proposed two-dimensional DCT.

4.1 The discrete cosine transform (DCT)

A transformation transforms a sequence of discrete signals into another data sequence $\{x_0 \cdots x_{N-1}\} \rightarrow \{z_0 \cdots z_{N-1}\}$. The purpose of the transform is to separate the input sequence into relevant and irrelevant information. The irrelevant information is then suppressed by a quantizer. Figure 4.1 shows how the input signal $X = \{x_0 \cdots x_{N-1}\}$ is transformed into the sequence $Z = \{z_0 \cdots z_{N-1}\}$. The elements z_k of this sequence are then separately quantized by the quantizer q_k. This yields the data $Z_q = \{\hat{z_0} \cdots \hat{z_{N-1}}\}$ and is for example transmitted over a communication channel. At the receiver side the data is then dequantized to Z' where $Z' \approx Z$. The data Z' id then transformed back to X'. The purpose of these transformation is to compact as much relevant information in few as possible signal elements. There are many kinds of transforms. For more information see [11]. One transform is the Discrete Cosine Transform (DCT), which, was first introduced by Ahmed et. al. in 1974 [2] and is related to the Discrete Fourier

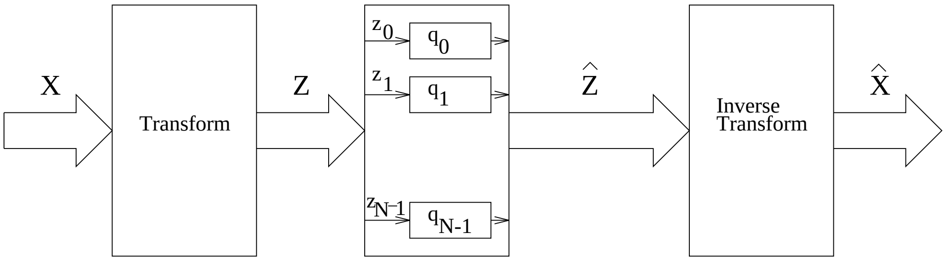

Figure 4.1: A scheme of transformation coding.

Transform (DFT). It is shown that the properties of the DCT are closer related to the theoretical optimal Karhunen Loeve Transform then other known transformations. The advantages of the DCT are:

- The DCT has almost optimal decorrelation properties for video signals [11].
- The DCT has less discontinuities at the border of a data block, see [6] for more information.

There are six types of DCT (DCT I ... DCT VI) were the DCT II is the most common type used. The MPEG coding uses the DCT II for transformation of its blocks and is the only DCT which will be discussed in this report. The DCT transforms a data sequence from the time domain into a sequence of the frequency domain $\vec{X} \rightarrow \vec{Z}$. The formula to compute the DCT is:

$$z_k = \sqrt{\frac{2}{N}} \; c_k \sum_{n=0}^{N-1} x_n \; \cos\frac{k(2n+1)\pi}{2N} \tag{4.1}$$

where

$$\vec{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_{N-1} \end{bmatrix}, \; \vec{Z} = \begin{bmatrix} z_1 \\ \vdots \\ z_{N-1} \end{bmatrix} \text{ and } \; c_k = \begin{cases} \frac{1}{\sqrt{2}} & k = 0 \\ 1 & k \neq 0 \end{cases}$$

The DCT formula (4.1) can also be written in matrix notation, which results in:

$$\vec{Z} = \vec{X} \cdot C \tag{4.2}$$

where C is a square matrix which can be derived from formula 4.1. The formula for the two dimensional DCT is:

$$z_{k,l} = \frac{2}{N} \; c_k \; c_l \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x_{m,n} \; \cos\frac{(2m+1)k\pi}{2N} \; \cos\frac{(2n+1)l\pi}{2N} \tag{4.3}$$

and its matrix notation is:

$$Z = C \cdot X \cdot C^t \tag{4.4}$$

DCT algorithm	Optimizing method	No of mul.	No of adds	No of shifts
normal	none	64	56	0
Chen	matrix	16	26	0
Duhamel	algebraic formula	11	43	15
Lee	algebraic formula	12	29	0
Hou	matrix	12	29	5

Table 4.1: Computational complexity of different 8 point 1-D DCT algorithms.

In most optimized algorithms the scaling factors $\sqrt{\frac{2}{N}}, \frac{2}{N}, c_k$ and c_l are neglected because the formulas without these factors can be optimized. These algorithms without the scaling factors are in this report called *core DCT* algorithms as shown in formula 4.5.

$$z_k = \sum_{n=0}^{N-1} x_n \cos \frac{k(2n+1)\pi}{2N} \tag{4.5}$$

For a correct DCT computation the scaling factors should be added by all core algorithms. In the literature many different kinds of optimizations method are used for the DCT. The first distinction is if the algorithms were optimized for implementing in VLSI or to reduce the computational complexity. In [6] many optimization algorithms are discussed and classified and in table 4.1 an overview of these fast DCT is presented. Most of the methods for optimizing DCT can be classified into three classes:

Indirect method In [2] it was already shown that the DCT can be computed using the FFT. With this method the algorithm can be separated into three parts. First the input data is adjusted. Second the data is transformed by the DFFT. Third the output of the FFT is then again adjusted. These three steps together perform a DCT transform. The advantage of this approach is that if the input and output blocks are designed one can choose many well proven DFFT. A disadvantage is the computational overhead caused by adjusting the input and output blocks.

Matrix optimizing method The DCT can also be calculated using matrixes and since this matrix is constant and unitair, the DCT matrix can always be factorized into smaller matrices. These algorithms are tailored for the DCT matrices and shown to use less operation than the first method. An example of this method is shown in section 4.2.1.

Algebraic formula optimizing method The algorithm given by equation 4.1 is optimized using algebraic solutions like the polynomial transforms and prime factors.

In the next section an optimized DCT is discussed which is used in the final MPEG program.

4.2 An optimized DCT

To compute the DCT transform, using formula 4.3, gives a very high computational complexity and therefore a high computing capacity is necessary. The computational complexity of the DCT algorithm is expressed by the number of additions, multiplications and shifts necessary to compute the DCT. After the DCT was defined many optimization of the DCT algorithm were proposed which lead to a significant reduction in the computational complexity.

In search for a good algorithm to implement in hardware there are a number of factors which determine the best algorithm. One aspect is the number of additions and multiplications necessary to compute the DCT. In table 4.1 it can be seen that the algorithms of Hou and Lee have the least number of operations and since their flow-graphs are very regular it seems that these algorithms are better to implement in hardware than the others. In [6] it is shown that the algorithm of Hou yields the best solution, which is described next.

4.2.1 Hou's Fast DCT

Hou uses the matrix optimizing method to optimize the one dimensional DCT algorithm. The method proofs that a N-point DCT can be computed by two N/2-point DCTs and some extra multiplications. This method is used again to compute the two N/2-point DCT recursively. The proof of this method is shortly discussed in this section but detailed information is given in appendix B. The core DCT matrix is written as:

$$Z = C_c(N) \cdot X \tag{4.6}$$

where $C_c(N)$ is an $N \times N$ core DCT matrix and X and Z are the column vectors of the input and output data sequence in natural order. The matrix $\hat{C}$ equals the C_c only the row and columns are shifted horizontal and vertical. Because of this the input and output sequence of matrix $\hat{C}$ are also reordered. To get a better understanding of the matrix $\hat{C}$ a few simple examples are given in appendix B. From these three examples in the appendix you can see that the following recursive property for the DCT matrix $\hat{C}(N)$ can be derived:

$$\hat{C} = \begin{bmatrix} \hat{C}(\frac{N}{2}) & \hat{C}(\frac{N}{2}) \\ \hat{D}(\frac{N}{2}) & -\hat{D}(\frac{N}{2}) \end{bmatrix} \tag{4.7}$$

Equation 4.7 is proved in report [6], here also the relation between $\hat{C}$ and $\hat{D}$ is derived. we can write:

$$\begin{bmatrix} \hat{z}_e \\ \hat{z}_o \end{bmatrix} = \begin{bmatrix} \hat{C}(\frac{N}{2}) & \hat{C}(\frac{N}{2}) \\ K\hat{C}(\frac{N}{2})Q & -K\hat{C}(\frac{N}{2})Q \end{bmatrix} \cdot \begin{bmatrix} \tilde{x}_p \\ \tilde{x}_r \end{bmatrix} \tag{4.8}$$

where $K = RLR$ and $Q = diag[\cos\phi_m]$ The N point vector Z multiplied by the N point matrix $\hat{C}$ multiplication can be reduced to two N/2 point multiplication be rewriting equation 4.8 to:

$$\begin{aligned} [\hat{z}_e] &= \left[\hat{C}(\tfrac{N}{2})\right] \cdot [\tilde{x}_p + \tilde{x}_o] \\ [\hat{z}_o] &= \left[K\hat{C}(\tfrac{N}{2})Q\right] \cdot [\tilde{x}_r - \tilde{x}_o] \end{aligned} \tag{4.9}$$

DCT Algorithm	optimization method	No of mul.	No of adds
row/column with Lee's	row/column	192	464
Kamangar	matrix	128	430
Duhamel and Guillemot	algebraic formula	96	484
Cho with Lee	algebraic formula	96	466
Cho with Hou	matrix	96	466
Cho with Duhamel	algebraic formula	88	577

Table 4.2: Computational complexity of different 8×8 points 2-D DCT algorithms.

The operation of reducing the matrix multiplication costs N/2 additions and N/2 subtractions. The formulas of equation 4.9 can both be reduced to four N/4 point multiplication by applying the algorithm of Hou recursively. As an example in figure 4.2 the flow graph

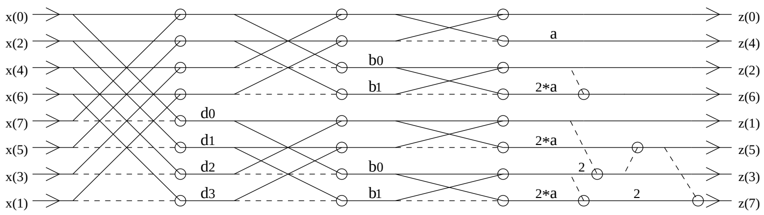

Figure 4.2: Flow graph of the 8 point DCT using Hou's algorithm.

of the 8×8 DCT is given. More information about flow-graphs see appendix A.

4.2.2 The two-dimensional DCT.

From the table 4.2 it can be seen that the row-column method with the fastest one dimensional DCT is still the slowest two dimensional DCT. To use this method is for the computational complexity not good but the fact that it is used in many real implementations is because the row column methods is easy to implement (see chapter 4.3).

A much better algorithm with respect to the number of operations is the algorithm of Cho. This algorithm consists of a number of adders (232) and eight one-dimensional core DCT. This one dimensional core DCT can be any fast algorithm like Lee's [5] or Hou's [8]. In the table three different implementation of the one dimensional core DCT are used. Although the Cho with Duhamel approach use the fewest multiplication, the number of additions is very large due to the Duhamel one dimensional core DCT implementation (see table 4.1). In the next chapter more information about the algorithm of Cho is given, but without the implementation of the one dimensional core DCT.

4.2.3 Cho's two dimensional DCT algorithm

Cho's two dimensional algorithm will is proofed using the core DCT formula. The same as in the one dimensional case the factors ϵ_k and ϵ_l are neglected so the core algorithm will be:

$$z_{k,l} = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} x_{n,m} \cos\left(\frac{k(2n+1)\pi}{2N}\right) \cos\left(\frac{l(2m+1)\pi}{2N}\right) \tag{4.10}$$

Cho's method is mapping equation 4.10 into N one dimensional core DCTs of length N. The algorithm is described by the following matrix formula.

$$Y = P_3 \cdot T_2 \cdot P_2 \begin{bmatrix} DCT_0 & 0 & \cdots & \\ 0 & DCT_1 & & \\ \vdots & & \ddots & \\ & & & DCT_{N-1} \end{bmatrix} \cdot T_1 \cdot P_1 \cdot X \tag{4.11}$$

where P_i are permutation matrices and T_i transposition matrices. The matrix with the DCT_n elements performs the N core 1D-DCT transform. The derivation of equation 4.11 is given in report [6]. Although the derivative of the Cho algorithm is complex the flow-graph in figure 4.3 of the algorithm for a 8×8 point DCT transform is regular. In this flow-graph also the tasks performed by the matrices used in equation 4.11 are given.

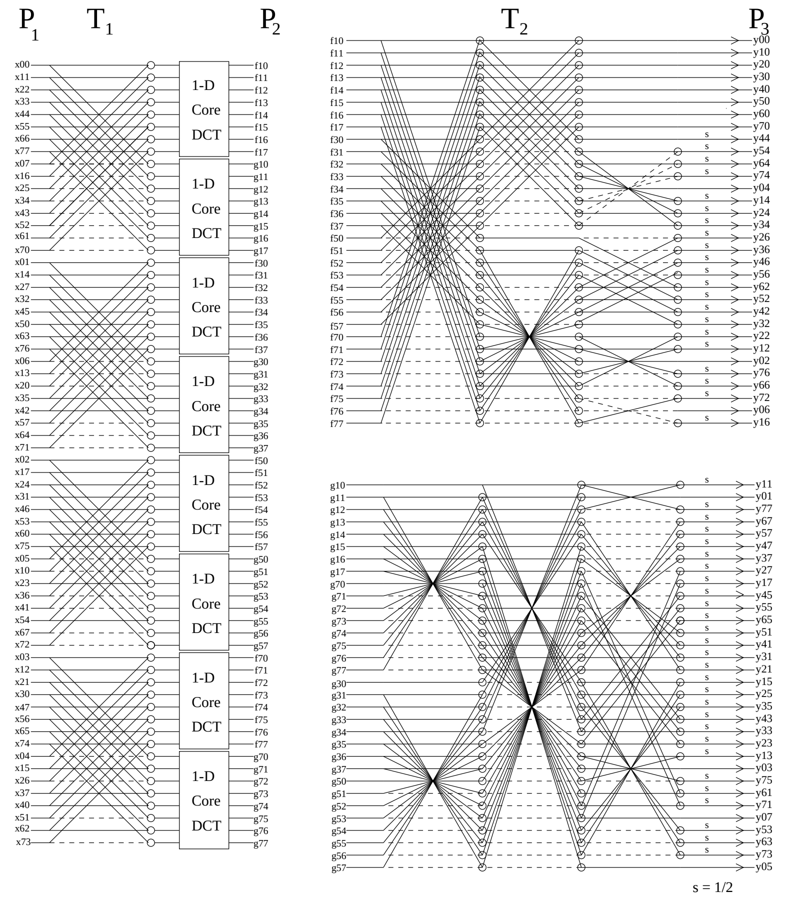

Figure 4.3: Implementation of Cho's 8×8 DCT using one dimensional DCTs.

4.3 The proposed algorithm

In the section 4.2.3 it was already shown that the algorithm of Cho was the best in computational complexity. To implement this algorithm with the MOVE framework is easy because, apart from the eight one dimensional core DCTs, you only need 233 add operations to perform, and the functional units you must have are equal to the ones required by the other algorithms. The Cho algorithm will be used in this report to implement the two dimensional DCT algorithm. Because Cho's algorithms uses a one dimensional core DCT a choice for this algorithm must be made. In section 4.2.3 it was already shown that the algorithms of Lee and Hou are algorithms with low number of operations. Although the algorithm of Lee uses less operations than Hou's algorithms there are also some disadvantages within Lee's algorithm. The flowgraphs of Cho (fig 4.3) with the Lee or Hou flowgraph can be computed using floating point operation (adds and multipliers). The accuracy of the computation would be satisfying, but to use floating point functional units in the MOVE framework, would require a large chip area and a long time to compute the result.

A better solution is to use integer calculations to compute the algorithm because these calculations are fast and the chip area used by the FUs is smaller then with floating point operations. A disadvantage is that the roundoff errors introduced by the integer calculations are much higher than for floating point. Another problem is that the range of the number is limited and if the algorithm exceed this range an overflow will occur.

In the next section the problems of using integer data format in the DCT algorithm are discussed. First the bit length to represent the number that can occur in the Cho algorithm is determined. Second the bit length of the Hou algorithm is discussed.

4.3.1 Two's complement integer data format

An important aspect of implementing the DCT algorithm is to determine the number of bits to represent the numbers that occur in the algorithm. Integer data formats have mostly a fixed length like eight or sixteen bits per word. On these words the add and multiply operation are performed. The incoming data of the DCT algorithm should be interpreted by the MPEG algorithm as a sequence of 9 bits two's complement numbers.

If a decimal integer G is represented by a two's complement notation of N bits $R_2^n(G)$ the decimal range of G, is defined as:

$$-2^{N-1} \leq G \leq 2^{N-1} - 1 \tag{4.12}$$

Decimal numbers greater than defined by 4.12 can not be represented by $R_2^n(G)$. The relation of $R_2^n(G)$ and G is as follows:

$$G \leftrightarrow R_2^n(G) \leftrightarrow -b_{N-1}2^{N-1} + b_{N-2}2^{N-2} + \cdots + b_1 2^1 + b_0 2^0 \tag{4.13}$$

When adding two decimal numbers of range G then the answer will be of range $2 \times G$. If two two's complement representation $R_2^n(G)$ of two decimal numbers are added than the range of the two's complement notation should be $R_2^{n+1}(G)$, so when adding two numbers of N bits then the result is maximal $N + 1$ bits.

The number of bits required to represent a number when it is multiplied by another number can also be determined. A general formula of the bits required when to numbers ($z = x \times y$) are multiplied is:

$$R_2^a(z) = R_2^b(x) \times R_2^c(y)$$

$$\text{then } a = b + c \text{ for } b, c > 1 \text{ and } a = b \text{ for } b > 1 \text{ and } c \leq 1$$

With these two rules it is possible to estimate the numbers of bits required for Cho's algorithm.

4.3.2 Bit length required for the Cho algorithm

In this section the minimal number of bits necessary to represent the integer numbers that occur in the Cho algorithm is determined. In this section there are two methods used to determine these number of bits:

- Using flowgraphs of the algorithm. The number of bits are determined by the rules defined in 4.3.1. With this method it is possible to determine the bit-length of the numbers that occurs internal in the DCT algorithm.
- Using the DCT formula. Only the bit-length of the output of the algorithm can be determined. This method is more accurate than the method using flowgraphs.

First the maximum integer number of each output and second the internal bit-length of the Cho algorithm, is determined.

It is possible to determine the number of bits required for the output of the two dimensional core DCT algorithm by comparing the 9 bits input with the two dimensional core DCT formula

$$z_{max,k,l} \equiv max \left(\sum_{m=0}^{7} \sum_{n=0}^{7} x_{m,n} \cos \frac{(2m+1)k\pi}{2N} \cos \frac{(2n+1)l\pi}{2N} \right)$$

Because in the algorithm only add and multiplications are used all the numbers computed at every stage are first degree functions of the 64 input values, in formula notation this is:

$$z_{max,k,l} \equiv c_0 \cdot x_{0,0} + c_1 \cdot x_{1,0} + c_2 \cdot x_2 + ...c_{63} \cdot x_{7,7}$$

The maximum of $z_{max,k,l}$ is obtained by adding the maximum of all the terms (i.e. $c_a \cdot x_a$). To determine the maximum of $c_a \cdot x_a$ the input value x_b should be either maximal positive or maximal negative value whether the constant c_a is positive or negative. For determining the maximal negative value of $z_{max,k,l}$the input values should thus be the opposite of the maximum case.

$$z_{max,k,l} \equiv max(x_{m,n}) \sum_{m=0}^{7} \sum_{n=0}^{7} abs \left(\cos \frac{(2m+1)k\pi}{2N} \cos \frac{(2n+1)l\pi}{2N} \right) \tag{4.14}$$

Equation 4.14 gives the maximal output of each element of the two dimensional core DCT. When equation 4.14 is evaluated for every k, l then most elements can be represented by 14 and some by 15 bits. The maximal number required to represent all output elements is thus 15 bits. Now the bit-length necessary to represent the internal integer in the Cho algorithm is determined. In figure 4.3 it can be seen that the two dimensional core algorithm of Cho can be separated into six stages, these are:

- The first stage is the input of the data matrix x here the range of the data is represented by 9 bits.
- The second stage is the data after it has been added by the row of adders. Since adding two numbers of the same range involves that the result is represented by an extra bit. The input of the core one dimensional DCT are than 10 bits.
- The third stage is the data matrix after the one dimensional DCT. Assume that the number of bits here is $T + 10$.
- The fourth stage is the stage after the second row of adders in the figure. Here the required number of bits is $T + 11$.
- The fifth stage is after the row of adders following stage four. The number of bits should be $T + 12$.
- The sixth stage is the the output of the two dimensional core DCT algorithm. In this stages there is a row of adders and shifters. The shifter divides the data by two and since every adder is followed by a shifter the extra number of bits after this stage equals $T + 12$. To compute the sixth stage however, there could be one extra bit necessary because the results of the adder should first be represented by a number of $T + 13$ bits before it is decreased by the shifters to $T + 12$ bits.

From examining the stages and determined the bit-length of the internal integers with the flowgraph method it is clear that the internal precision of the algorithm should be $T+13$ bits and that the output of the algorithm can be represented by $T+12$ bits. With formula 4.14 it is proven that the output elements with the shifter at the sixth stage can be represented by 14 bits instead of 15 bits, so the maximal bit-length required in the sixth stage is 15 bits and it is proofed that all the internal number used in the algorithm except from the one dimensional core DCT can be represented by 15 bits.

Now it is easy to derive the number of bits required to represent the output of the one dimensional core DCT algorithm. This will equal to $15 - 2 = 13$ bits which could also be derived from equation 4.5 and the fact that the number of bits of the input of the core DCT equals 10 bits. In the next section the difference between the internal integer bit-length of Hou's one dimensional core DCT algorithm is discussed.

4.3.3 Bit length required for Hou's algorithms

In table 4.1 it is shown that the algorithm of Hou has a higher computational complexity then the algorithm of Lee [5]. The reason that the algorithm of Hou is used to implement in VLSI is because the fact that it is more stable to compute. In the previous section it was shown that the input data of the one dimensional core DCT equals a ten bits two complement word. From equation 4.5 it is seen that the maximal output of the core DCT with ten bits (-512, 511) input, is the output element z(0) and will be 13 bits wide. To determine the bit width of the other (internal) numbers that occur in the algorithm, the flowgraphs of Hou and Lee are separated into five stages.

- The first stage is the input of the algorithm.
- The second stage is after the first row of adders and the constant multipliers $d0-d3$ in the flowgraph.
- The third stage is after the second row with adders and the constant multipliers $b0$ and $b1$.
- The fourth stage is after the third row of adders and the constant multipliers a or $2*a$.
- The fifth stage is the output of the algorithm.

Because in the algorithm only add and multiplications are used all the numbers computed at every stage are first degree functions of the eight input values, in formula notation this is:

$$s = c_0 \cdot x_0 + c_1 \cdot x_1 + c_2 \cdot x_2 + c_3 \cdot x_3 + c_4 \cdot x_4 + c_5 \cdot x_5 + c_6 \cdot x_6 + c_7 \cdot x_7$$

Here the number s in the formula represent every number that occur in the Hou or Lee algorithm. When for all possible s in the algorithm the maximum is determined the required bit length is also known.

When all possible combinations of maximal and minimal input values are processed by the core DCT algorithm and every s is monitored for its maximum and minimum values then the maximal and minimal of every possible s is known. The method described above is used on the algorithms of Lee and Hou and the results are given in graph 4.4. The maximal number of bits required to compute Hou's algorithm is equal to thirteen bits and for Lee's algorithm fourteen bits are used. The fact that Hou's algorithm uses less bits than Lee's is because the constant multiplications of Hou are all smaller then one, Lee's constant are smaller than approximately. 2.5. In the graph there is also the average of bit required for a set of random input vectors.

Because the algorithm of Hou is more stable than Lee's the algorithm of Hou is used for implementing in the algorithm of Cho. The c code to compute this proposed algorithm is printed in appendix C (Hou algorithm) and in appendix C the code for Cho's algorithm is given.

4.4 Conclusion of the Fast DCT

In this section the quality and the execution time of the Fast DCT are presented. The Cost to implement of the are also taken in consideration.

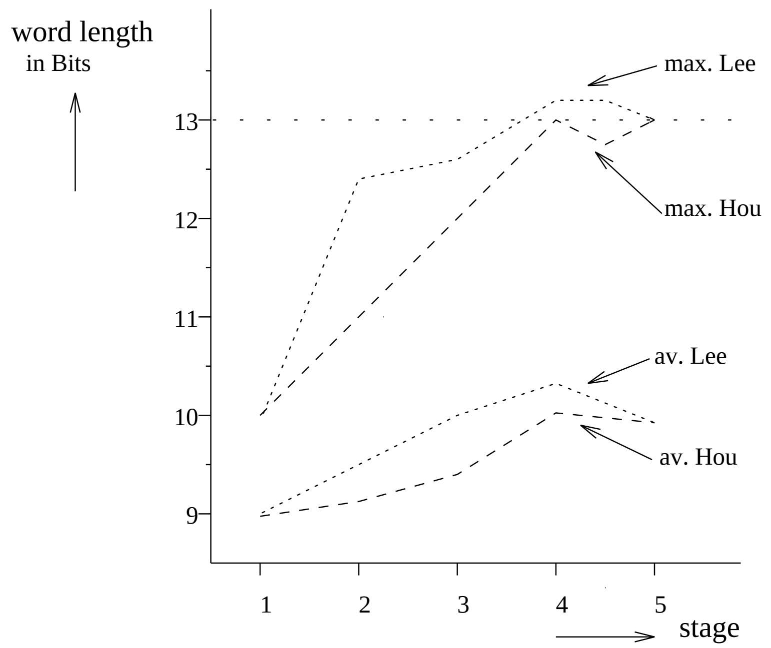

Figure 4.4: The maximal number of bits required for Hou's algorithm is lower then Lee's algorithm.

Quality and the cost of the FDCT Improving the execution time of the original DCT is not the only aspects to optimize. The other parameter taken in consideration are the cost to implement and the quality of the DCT. Since the quality of the FDCT is closely related to the number of bits used to represent the words used to compute the DCT. And the number of bits to represent the word closely related to the cost, these two aspects are discussed together.

To compute the DCT several multiplications with real constants are performed. These multiplications introduce rounding errors which affect the quality of the encoder. To improve the accuracy, the Fast DCT exploits the maximal bit-length required to hold the two-complements data format, proven in the previous chapter. The bit-length of the input data is nine bits, and for every stage in the flowgraph of the FDCT the maximal bit-length is known. The proposed FDCT uses a method of storing the maximum number of bits in the most significant part of a sixteen bits word. The resulting bits are used for precision of the computations. In figure 4.5 an example is given, here two nine bits words with seven bits precision are added and the ten bits answer is stored with six bits precision. This technique is used for every stage and results in a good accuracy of the FDCT.

To investigate the quality of the FDCT algorithm is compared with a qualitative reference DCT. Since the original DCT is a qualitative reference DCT the error of this DCT is assumed to be zero. To measure the error of the FDCT a measure environment is set up. This measure environment is given in figure 4.6 The result of the test are given in table 4.3. This test is closely related to the IEEE 1180-1990 accuracy test for IDCT.

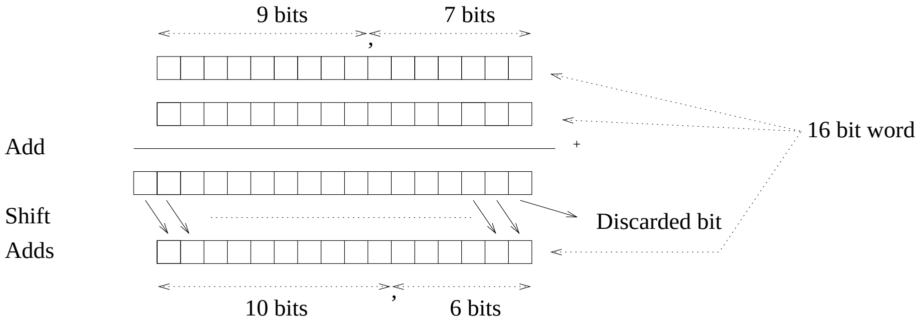

Figure 4.5: The adds operation consists of an add and a shift operation.

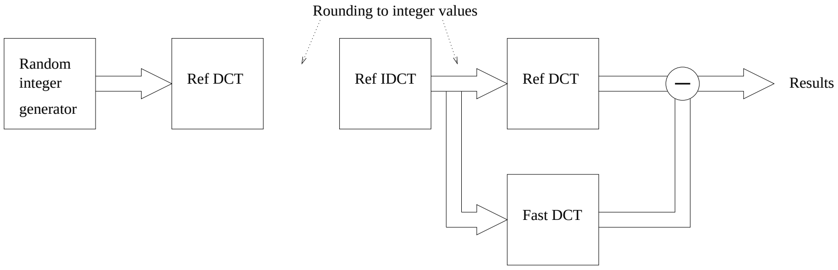

Figure 4.6: The scheme to test the accuracy of the FDCT.

Execution time of FDCT To determine the speed-up introduced by the FDCT the MPEG encoding program is tested with the original and the FDCT algorithm. The improvements are shown in table 4.3 and it is shown that the FDCT yields a better execution time. When the FDCT is implemented by the MOVE framework another speed-up is achieved if the operation adds, subs and muls are computed by special function units. The adds operation is up till now simulated by an add and a shift function, but this operation in the MOVE framework is computed in a single operation.

Type	Exc.time Giga Cycles	Worst peak error	Worst peak mean square error	Arithmetic
original DCT	2.5	-	-	Floating point
proposed FDCT	0.95	0.0405	0.0179	16 bit integer

Table 4.3: Comparison of the original DCT and the proposed FDCT.

5 Motion estimation

In the MPEG codec scheme the motion estimation is used to compress the temporal redundancy. The way motion estimation works is to a find blocks in a reference frame which match with blocks the source frame. When such a match is found not the picture information itself, but only the reference where the information can be found in the reference frame is transmitted to the decoder. The quantity of the reference information is much smaller than the picture information itself, hence a compression is established.

Chapter 5.1 describes some general information about motion estimation. In section 5.2 the original motion estimator, which is the full-search, is discussed. In section 5.3 the three dimensional recursive search (3D-RS) is explained in detail. This motion estimator is used in the Fast MPEG encoder program. In section 5.4 the aspects about implementing the 3D-RS algorithm within the MPEG encoding scheme are discussed. In section 5.5 the quality versus the executions time of the 3D-RS and the original full search algorithms are compared.

5.1 Motion estimation overview

In this section some general properties of the motion estimation algorithms are explained. The literature describes many kinds of motion estimators. Most of these operate in the space domain of the picture (i.e. pixel-domain) but there also exists motion estimators which operate in the frequency domain. These however are not discussed in this report. Some motion estimation algorithms are:

- Full search

- Three step search
- Telescopic search
- Sum search

More information about the algorithm and their computational complexity are given in [9]. In this report only the **three dimensional recursive search (3D-RS)** and the full search motion estimators are discussed. The full search is used in the original encoder program, the 3D-RS is proposed as the successor to the full search. Although the 3D-RS was original designed for a consumer display field rate conversion, the algorithms also produce good result when used as a motion estimator for the MPEG scheme. In this section more information about motion estimation in general and the block matching are discussed.

Motion estimation can be interpret in two ways. The first is that the motion estimator finds the motion vectors of the moving objects in screen. These estimators are also called *true motion estimators.* The other interpretation is to find a block in the source frame which match the best with a block in the reference frame. Matching means their exists a formula which classifies how good two patern matches. Although these two interpretation almost look the same there are some important differences. With the first interpretation it is worthwhile to use the previous motion vector as a reference for the search of a new motion vector. This is possible because a moving object in a sequence is likely to have a constant speed. This techniques doesn't work for the best block search algorithm (like full search) since in that case it is not sure wether the 'motion vector' truly means that this is the motion of the object. A second pitfall is how to interpret the best match, which is explained later. Most motion estimation algorithms are processed in two steps:

1. Determine a set of candidate blocks in the reference frame.
2. Determine the block from the set of candidate blocks, which match the best with the source block, see figure 5.1.

The first step determines if the motion estimator produce true motion vectors. The motion estimator which produce true motion vectors composes the set of macroblocks in such a way that the number of block in the set is low but the probability that the true motion block is contained in the candidate set is high. The other motions estimators, like the full-search, compose a large set of candidate blocks and use the block matching to find the best-match block. Here the decision to choose the best motion vector dependent only on the the best-match. This does not always produce the best answer, and is discussed in more detail next.

For the second step of computing the motion estimation, the set of candidate blocks are matched with the source block in the source frame. To determine the block which matches the best there exist two well-known algorithms which are used in many motion estimator algorithms. The best match algorithm is mostly computed by subtracting the source from candidate blocks. The sum of the absolute or quadratic value of the difference is an indication for the match. This technique is called *Block matching.* Two well know block matching algorithms are the *Mean Absolute Error (MAE)* and the *Mean Square error, (MSE)* and are computed using the functions defined as follow:

$$MSE = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (S_{ij} - C_{ij})^2 \tag{5.1}$$

$$MAE = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \mid S_{ij} - C_{ij} \mid \tag{5.2}$$

Where S_{ij} is a pixel of the source block and C_{ij} a pixel of the candidate block. The value N is equals 16 for the MPEG scheme. The block matching is computed for all possible

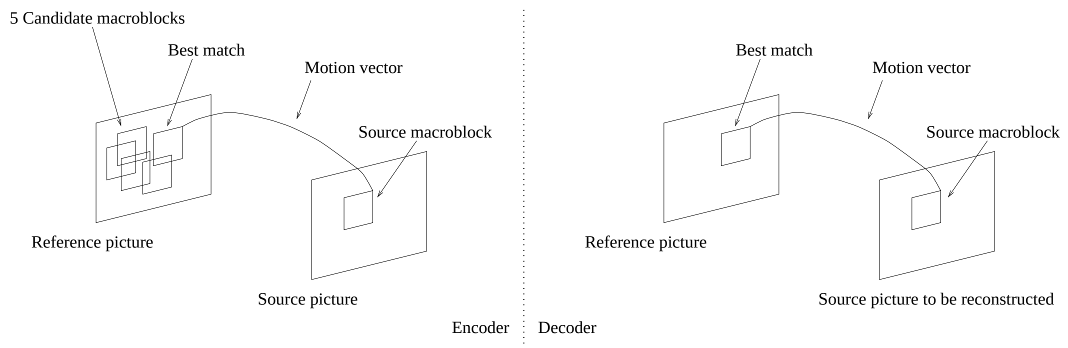

Figure 5.1: A picture is reconstructed by a motion vector and block difference information.

candidates in the set of blocks using the MSE or the MSA. In figure 5.1 a situation of block matching is illustrated. At the encoder side there is a set of 5 candidate blocks, all these blocks are matched with the source block in the source picture using the MAE/MSE. The candidate block with the smallest MAE/MSE is the best match. Now the difference of the best match and the source block is send together with the motion vector to the decoder. At the decoder side the decoder uses the motion vector to locate the best match block and adds the contents of this block with the difference send by the encoder yielding the source block. It should be noticed that the decoder should have a copy of the reference picture to reconstruct the source picture.

The disadvantage of using the MSE or MAE to find the best block matching is that the information to be transmitted by the encoder is not necessary smaller if the MSE or MAE is smaller, on the other hand the main goal to use motion estimation is the reduce the transmitted information. This problem is caused by the DCT transform, which is emphasized in figure 5.2. In this figure two macroblocks are compared with a source macroblock. For clarity only eight of the pixels are shown. The figure is divided into three parts. The first part shows the two candidate blocks which are matched with the source block. In the second part of the picture the difference of the two candidates block is drawn. The upper block has a MAE of 8 and the bottom a MAE of 14. The motion estimator uses the block with the lowest MAE so the upper block is send to the decoder. However a better solution for finding the best match would be to use the difference

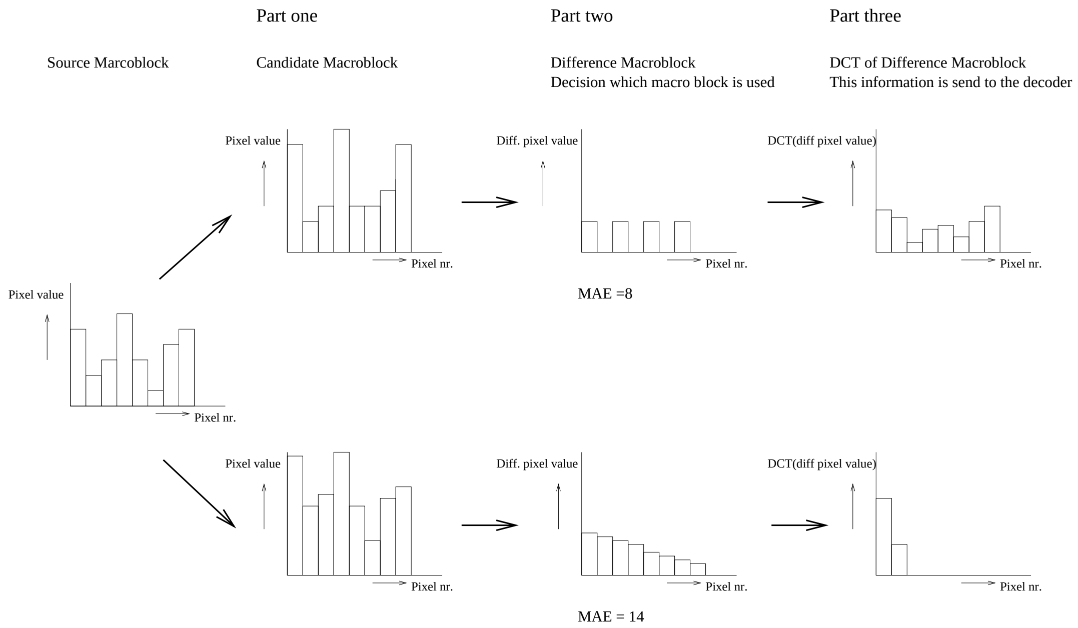

Figure 5.2: The smallest MAE thus not always yield the best solution.

information after it is transformed by the DCT. See part three of figure 5.2. It is clear that the information of the lower block uses less information than the upper-block. Hence the lower block should be the best match. The decisions are made at the second stage because to, compute the DCT for every candidate macro block is computational too excessive.

Conclusion: From the previous example it is concluded that the MAE and the MSE are the only known block matcher but care must be taken since this algorithm not always gives the best solution.

Pixel sub-sampling The operations that are required to determine the block difference of two blocks account for most of the computational complexity as there is an almost linear relation between the number of pixels in a block and the computational complexity. Thus, a reduction in the number of pixels per block that are required for a block comparison would yield a similar reduction in the overall computational complexity of any search algorithm. In [14] sub-sampling of frames by a factor of 2 was proposed for this reason. This means that only half of the pixels in a block are used in a comparison. A sub-sampling pattern like in Figure 5.3 can be used. In [14] it was shown that this 2 to 1 sub-sampling led to an increase in the prediction error entropy of only 0.04 bit/pixel, which is very acceptable as the related computational complexity is reduced by 50%. A sub-sampling factor of 4 is worth considering as well because it provides a reduction factor of 4. It was found to lead to an increase of the error entropy of 0.1 bit/pixel, which was judged *excessive* in [14].

In [13] it was mentioned that due to the recursion within the 3D-Recursive Search

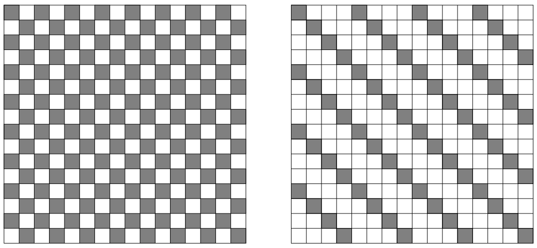

Figure 5.3: Patterns for sub-sampling

algorithm it is sensitivity to sub-sampling is expected not to be very high. An increase in the MSE of 0% and 1% for a sub-sampling factor of 2 and 4, respectively, were reported. A serious degradation of the performance could occur, however, for sub-sampling by a factor 4 when the pattern shown in Figure 5.3 is used. This is for scenes that contain periodic structures along the diagonal axis that have the same interval as the lines in the subsample pattern. These structures might not be traced then.

5.2 Full-search motion estimation

The full ME search is the algorithm that is used in the original MPEG encoding program. This estimator doesn't produce true motion vectors, and a it is time consuming algorithm for the encoding program, which was shown in chapter 3. In section 5.5 more information about the performance of the full-search compared with the 3D-RS is given. The full-

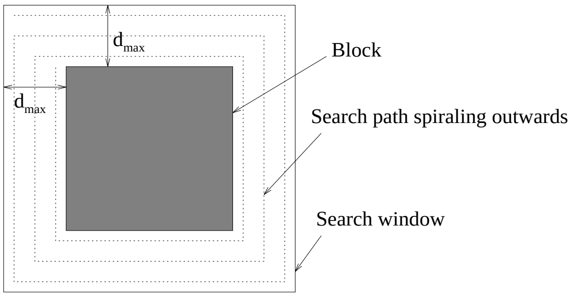

Figure 5.4: The search window and the search path spiraling outward of the full-search motion estimation algorithm .

search is the best algorithm to find the best matching block. It search for the best match in the set of candidate macroblocks which lies in the search window. The search window is an area around the source macroblock as indicated in figurer 5.4. The number of block-matches are related to the surface of the search window, and equals to $(d_{max}+1)^2$, where d_{max} is the maximum for the x or y component of the vector found by the full-search. The formula for the number of block matches indicates that the number of block matches is of quadratic order with d_{max}.

There are three optimization possible to minimize the number of block matches.

1. The aspect ratio of the width and hight of the search window can be optimized.

This is profitable because on average the x component of the motion vector is larger than the y vector (i.e. there are more horizon than vertical displacement of the sequence).

2. The block matching algorithm can be optimized. An example is given for the MAE block-matching, but it can also be used by MSE block-matching. The block match algorithm tries to find the minimum of the difference of two blocks by summing the difference of every pixel of the block. In case of a block of 16×16 pixels this would require 256 summations of the absolute differences. If for a given $i < 256$ the partial sum of the absolute difference is already greater than a previous block match i.e. 256 summations of absolute difference, the current block match is never a best match, hence the summation of the $256 - i$ are unnecessary and the execution of the current block can stop despite not all the summations are calculated.

3. With this optimization it is worthwhile to order the search path of the best-match within the search-window. It is efficient to start with the candidate which on average is likely to be the best match. Since on average the best match block is found near the source block, a speed-up is achieved if the search path is organized in an outward spiral. In [12] it is shown that these last two optimization reduce the time to compute the full-search by 50% and 30% respectively. These previous discussed optimizations are also utilized by the full-search algorithm of the original MPEG encoder program.

5.3 The 3D-RS algorithm

The 3D-RS [7] is a true motion estimator designed for a field rate conversion of television. The main goal of the design was to prevent flickering of the moving pictures. This is established by introducing a new field between two successive fields, by motion compensated interpolation. Of the two successive fields. To implement this field-rate conversion used within consumer electronics the cost of the implementation in hardware must be low. In **??** it was also concluded that for the human visual system strong local distortion seems worse than a global distortion of the picture, although the first distortion has a significant smaller MSE than the global distorted picture. In [9] an implementation was proposed to use the 3D-RS in the MPEG scheme. Also some modifications of the 3D-RS algorithm were proposed. In this section the original 3D-RS and some modifications described in [9] are explained.

The name of the algorithm is divert from the fact that the estimator uses a recursive way to determine the motion vector. For the recursion previous determined vectors of neighbour blocks are used. These vectors are called estimate vectors, since they give an estimation of current motion vector. The 3D-RS algorithms is discussed in three steps were the first two steps determine the set of candidate blocks and the third step determines the best candidate block using a best-match algorithm. For determining the candidate blocks the 3D-RS uses four estimate vectors taken from neighbour (in time and space) blocks. Which neighbour blocks are used is discussed next.

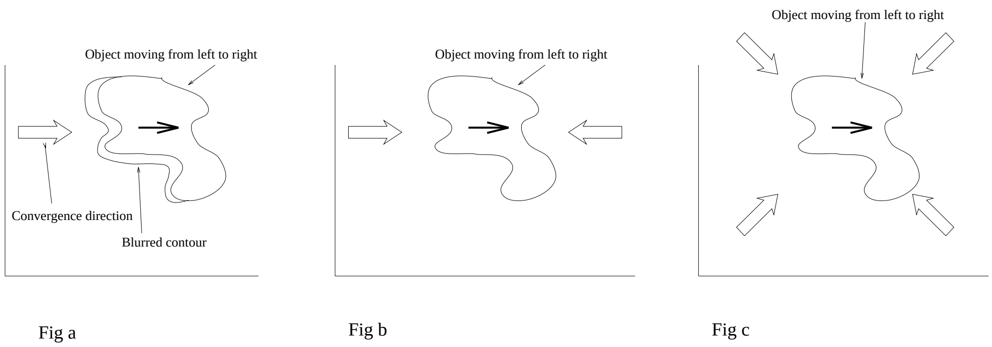

Figure 5.5: The convergence direction for different kind of dimensions.

One-dimensional recursive search It was already suggested that the use of previous determined motion vecctors is good starting point to determine the current vector. In [10] a recursive algorithm was determined using a neighbour vector from the spatial domain. The performance of the algorithm however yields a poor quality. The reason for the poor quality is that the discontinuities in the vector field (i.e. changes in the vector size or direction) which arise at the border of a moving object are not handled efficiently. See figure 5.5a for more information.

Two-dimension recursive search In the two-dimensional search the current vector is determined by using two previous vectors with opposite direction. This is shown in figure 5.5b. In the figure the moving object is also moving from left to right. Here there are two convergence directions and the convergence is complete . One problem using this method is that the motion vector of the left and right neighbour should be know to use this method. Hence this method is impossible. The determining of the motion vectors of the blocks starts at the top left en ends at the bottom right macroblock processing the blocks line by line. For a given block the three upper blocks of the previous line and the left block of the same line are yet processed. To solve this problem another the temporal component is exploit.

Three-dimension recursive search With the three-dimensional search for the previous motion vectors the vectors of the current and the previous pictures are used. In figure 5.5c the top convergence is established using the motion vector of the current frame (i.e. spatial domain) and the bottom convergence is established using motion vector of the previous frame. To overcome the Gaussian problem in the two-dimensional search this algorithm only use the the upper left and right blocks called spatial predictions Sa and Sb. For the opposite convergence the blocks shifted diagonally over 2 blocks downwards, see figure 5.6. The fact that the estimators Tb and Ta are shifted two block instead of one block is because the convergence of the Ta and Tb estimators are smaller than the estimators Sa and Sb. Experiments in increasing the number of diagonal shift shows an optimum for the value of 2.

Now the four estimators are determined we must construct the candidate motion vectors from the four estimators. Four candidates vectors are extracted from the vector

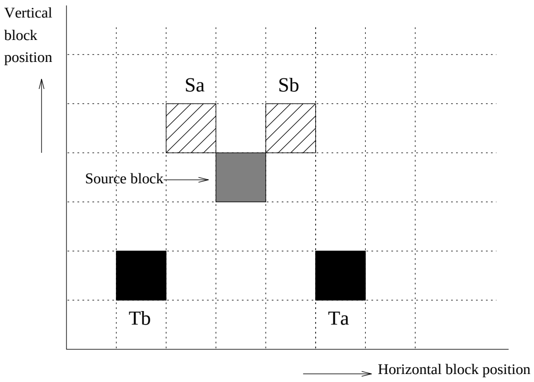

Figure 5.6: The convergence direction for different kind of dimensions.

of the estimators itself. From the two spatial estimators another vector is constructed using the *smart asynchronous cyclic search*. This algorithm is proposed by [9] and was closely related to the asynchronous cyclic search (ACS) suggested in [7]. This original ACS algorithm suggested to construct a vector composed of the estimator vector and a random offset vector circling asymmetrical around the the estimator vector which is called the L-update vector, see figure 5.7. The smart ACS use the information from previously composed ACS vectors.

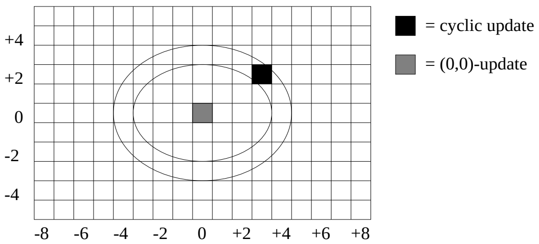

Figure 5.7: Asynchronous cyclic update-scheme

Asynchronous Cyclic Search Within the original ACS update scheme there is no relation between the chosen update and it is size and position; the L-update simply circles around the (0,0)-update. But it seems logical to continue the search in the direction of the last L-update as this position exhibited obviously less distortion than the (0,0)-update. A dynamic update scheme was developed that tries to exploit this information to determine the optimal L-update value and the related search direction for future searches. This scheme can be described as follows:

- Obtain the errors for the spatial prediction vector (Sa) and the vector related to the L-update (Sa + L) and determine the smallest of the two.
- If the L-update resulted in the smallest error then the L-value is updated, using an error-evolution criterion. The L-update for the second spatial prediction vector (Sb + L) is now in the same direction as the previous search, although it is size

is slightly different. As the obtained prediction vectors Sa and Sb are likely to be equal or nearly equal to one another, this update will result in a search at a position where the distortion is probably even smaller. Depending upon the outcome of this search the following actions are taken:

- The 1st L-update results in the overall best match: the evolution direction of the L-value is inverted and reset to it is primer value that led to the overall best match. The direction of the L-update is not changed, however, so that the search for the next block, that is expected to show great resemblance with the current block, will start with this L-value as well.
- The 2nd L-update results in the overall best match: the L-value is updated in the appropriate direction as obviously the distortion decreases monotonically in this evolution direction. The direction of the L-update is not changed.
- The 2nd (0,0)-update resulted in the smallest error: both of the L-updates seem to be in the wrong direction and/or of the wrong size. Therefore the direction is rotated over 90 degrees, while it is size is reset to it is initial value. By this rotated update-direction a future convergence towards the minimum distortion position is hopefully triggered as this position was obviously not located in the last two directions that were searched.

- The 2nd L-update results in the smallest error: the L-value is updated in the appropriate direction as obviously the distortion decreases monotonically in this direction. The direction of the L-update is not changed.
- The 1st or 2nd (0,0)-update results in the smallest error: both of the L-updates seem to be in the wrong direction and/or of the wrong size. Therefore the update direction is rotated over 90 degrees, while it is size is reset to it is initial value. By this rotated update-direction a future convergence towards the minimum distortion position is hopefully triggered as this position was obviously not located in the last two directions that were searched.

In Figure 5.8 two typical update situations are depicted.

In situation A the L-update direction is initially in the wrong direction in block x. After the (0,0)-update vector has been selected for Sa the direction of the L-update is rotated over 90 degrees. This still does not result in the selection of the L-update and therefore the direction is changed another 90 degrees before block x+1 is processed. Now the L-update yields the best match and after an update of the L-value Sb obtains the block that is located one pixel to the right from the absolute best match as the best match for this spatial search. In block x+2 a similar situation occurs but although the L-value seems to miss the actual position by one pixel the update scheme won't lead to any further overshoot. In situation B the L-update happens to be in the correct direction. After the L-update in combination with Sa has led to the best match the L-value is updated by 1 and used in combination with Sb, which results in a even better match. The L-value is now updated by 1 before it is used by the search for block x+1. This actually results in the ideal L-value as it would yield the absolute best match when combined with Sb. But before it is used by Sb it is updated by 1 which results in a

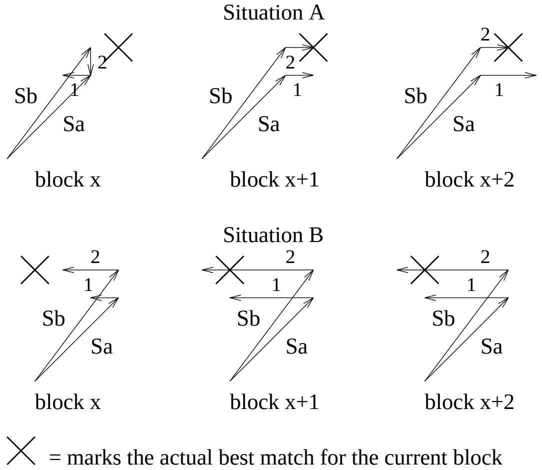

Figure 5.8: Smart L-update scheme for ACS

too large update-value. It is shown again that this won't lead to an actually overshoot situation.

Block sub-sampling Besides reducing the number of search-positions per block and the number of elements per block, the number of blocks per frame for which a search is performed could be subject to reduction as well. This reduction is based on the characteristic of the motion field that neighboring blocks have identical or nearly identical motion vectors. Using this assumption, sub-sampling of the blocks could be considered. For a fraction of the blocks a motion vector is determined by the search algorithm and then the motion field is interpolated in a appropriate manner to yield the motion vectors for the non-estimated blocks. As this interpolation requires only a marginal number of operations in comparison to the difference calculation, it can be said that a block sub-sampling factor of 2 reduces the computational complexity almost by a factor of 2 as well. The use of block sub-sampling could, however, lead to serious degradation of the performance of the algorithms when objects with the same dimension as the block size move with a velocity completely different from those of their neighboring blocks.

In [13] the motion vector of a non-estimated block is calculated by taking the three-point median of two neighboring vectors and a third vector from the corresponding block in the previous frame. A cascade of a horizontal and vertical median filter was used, described by the following equation:

$$\vec{v}_{i,horizontal}(t) = median[\vec{v}_{i-1}(t), \vec{v}_{i+1}(t), \vec{v}_i(t-1)] \tag{5.3}$$

$$\vec{v}_i(t) = median[\vec{v}_{i-\#hb}(t), \vec{v}_{i+\#hb}(t), \vec{v}_{i,horizontal}(t)] \tag{5.4}$$

$$with\ \#hb\ =\ number\ of\ horizontal\ blocks.$$

A drawback of this interpolation on the implementation level, however, is it is use of vectors from blocks that are located below the blocks for whom the interpolated vectors are determined. This means that the interpolation of a block can't be performed until the related lower block is processed. Therefore, in respect to hardware regularity and complexity a median filter in the horizontal direction is preferred, although this inhibits a similar latency in respect to the block at the right-hand side of the non-estimated block. Another possibility is therefore the use of the blocks located vertical above and at the left-hand side of a non-estimated block. The motion vectors are then all known when processing the blocks in a sequential order from the upper left corner to the lower right corner of a frame. The effect upon the quality of these models has to be verified however.

Block sub-sampling was used in [13] not primarily to reduce the computational complexity but to generate a smooth vector field as well. In it is original field rate conversion applications 3D-Recursive Search attempted to produce smooth images, whose absolute correctness is not as important as the perceptive impression of an image upon a viewer. With respect to the minimisation of the prediction error such a smooth set of interpolated motion vectors might not be satisfying at all and could reduce the performance of the estimator seriously. Therefore it seems advisable to restrict the gain of block sub-sampling somewhat by introducing an extra 3 (or 4) block-matches, related to the motion vectors of the surrounding blocks. The vector that has the smallest MAE is chosen as the vector of the initially non-estimated block (see also [15]). Again verification must show whether this increase in block matches counterbalances the performance gain in respect to the above described interpolation of motion vectors.

5.4 3D-RS implemented in the MPEG scheme

The 3D-RS was not designed for the MPEG scheme, in this section the requirement for the 3D-RS to use in the MPEG encoding algorithm are discussed.

The original 3D-RS always used the motion vectors of the previous picture as a reference for determine a motion vector of the current picture, so the distance of the vectors produced by the original 3D-RS algorithm always has a temporal distance of one. In the MPEG scheme the vectors which are determined have a maximal temporal distance of three in case of a ...IBBPBBP..... GOP, see figure 5.10. Another difference is that in the MPEG scheme there exists forward and backward motion vectors. To implement the 3D-RS algorithm a number of modification for the estimator vectors are suggested. The Fast MPEG encoder uses a fixed picture type sequence of ...IBBPBBP.. This regular structure is used by the motion estimator to a prescribed treatment of these sequences. In figure 5.9 it is shown how the whole sequence is separated into parts used by the motion-estimator. For each such a part, called mini GOP in this report the motion-estimator finds all the motion vectors, see figure 5.10 The first sequence is an I/P-picture followed by two B-pictures. The mini GOP sequence ends again with an I/P-picture, this last I/P picture is the first I/P picture for the next mini GOP. The motion vectors computed for a mini GOP are numbered according to figure 5.10 and for the inter-frame distance correction the equations by 5.5 are suggested.

$$vec_0 = \quad 1/3 \quad vec_2$$

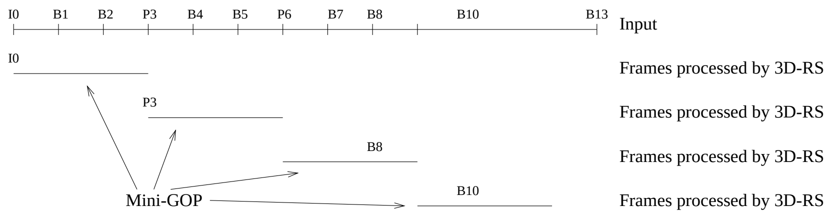

Figure 5.9: The input sequence is separated into parts of 4 frames which are processed by the motion estimator.

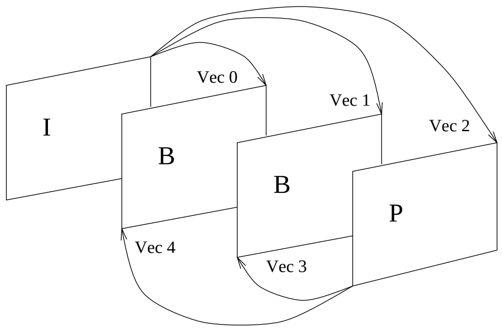

Figure 5.10: The inter-frame distance for the motion vectors of the MPEG scheme is greater than one.

$$
\begin{aligned}
vec_1 &= \quad 2 \quad vec_0 \\
vec_2 &= \quad 3/2 \quad vec_1 \\
vec_3 &= \quad 1/3 \quad vec_2 \\
vec_4 &= \quad 1/3 \quad vec_3
\end{aligned}
\qquad (5.5)
$$

The suggested inter-frame corrections for vec_2 in equation 5.5 could also computed using the following relation $vec_2 = 3vec_0$. In the following test it is shown that using inter-frame distance correction of $vec_2 = 3/2vec_1$ is better than $vec_2 = 3vec_0$. Another assumption is that $vect_0$ of the mini GOP is estimated with vector two of the previous mini GOP. In this report the following test showed an improvement of using this technique in the contrary to the tests in [9]. To test the two assumed inter-frame distance correction three test for two different sequence are performed. The results are shown in figures 5.11 and 5.12 which show the MAE for the three test $vec_2 = 3vec_0$, $vec_2 = 3/2vec_1$ and $vec_0 = 0$. From the results is concluded that using the inter-frame distance correction in equation 5.5 are the best. They are used in the proposed 3D-RS.

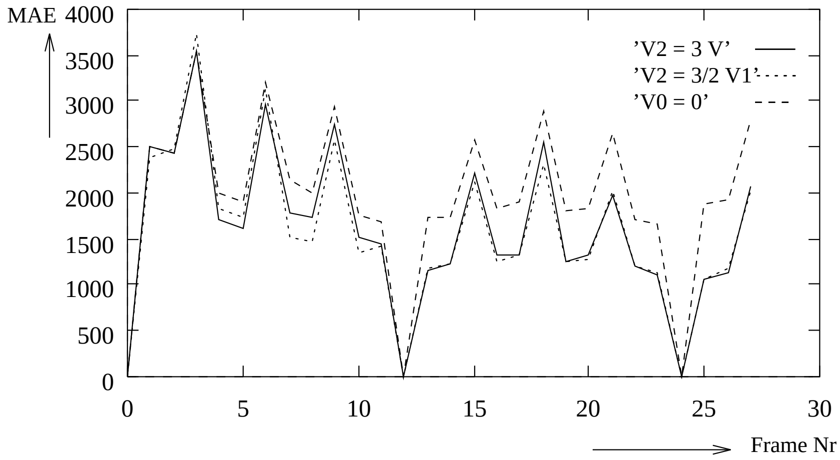

Figure 5.11: Three motion-estimation of the HPRL-car sequence to test the temporal heredity.

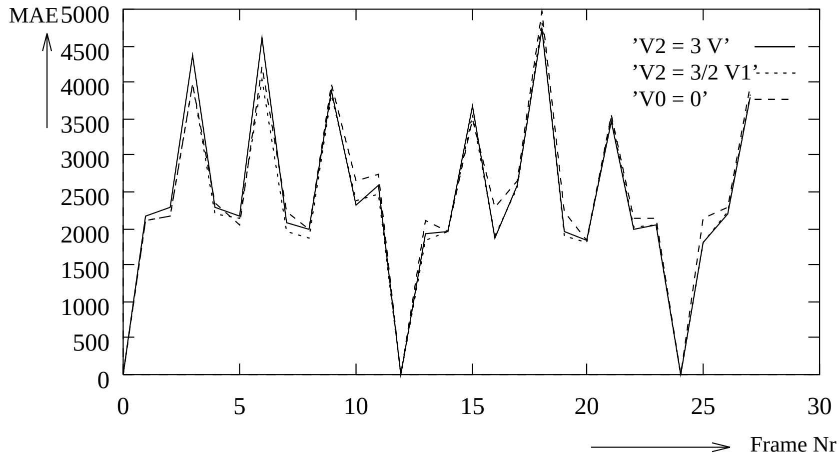

Figure 5.12: Three motion-estimation of the Garden sequence to test the temporal heredity.

5.5 Conclusions.

In this section the difference in quality and performance of the original full-search and the proposed 3D-RS algorithm are presented. For comparing the different motion estimation techniques tree test are used. the first test measures the quality of the pictures which are only motion estimated, the other the quality of the picture after it is compressed and decompressed by the MPEG codec. The last test represents a vector map of the two motion compensations techniques.

A problem of measuring the quality of a sequence is that none of the traditionally quality measurements are really satisfactory. For example it is easy to construct two images with the same quality but one is noticeable better for the human visual system (HVS). Another problem is how the HVS interprets moving pictures, it is likely that the HVS is more susceptible to different errors. In this report the errors of the motion-estimated pictures are measured with the mean absolute errors (MAE). The MAE is computed for every macroblock (i.e. 16×16 pixels) for a MAE of 700 the difference of the pixels in the source and estimated picture is about 2.73 (700/256), so decreasing MAE means better quality. For measuring the quality of the MPEG coded picture the PSNR is used. PSNR is defined as

$$PSNR = 20.Log_{10}\frac{255}{\sqrt{MSE}}$$

were MSE is defined by equation 5.2. Here a smaller PSNR means worser quality.

The first series of tests are performed on the motion-estimators algorithms only. For the test the sequence of the HPRL-car is used since this sequence has a high motion. For this test all permutations of the full-search and the 3D-RS are executed by motion-estimation of 16 pictures. The average of the MAE of the 16 motion-compensated picture is given in figure 5.13. The first series of test were performed to compare the quality and

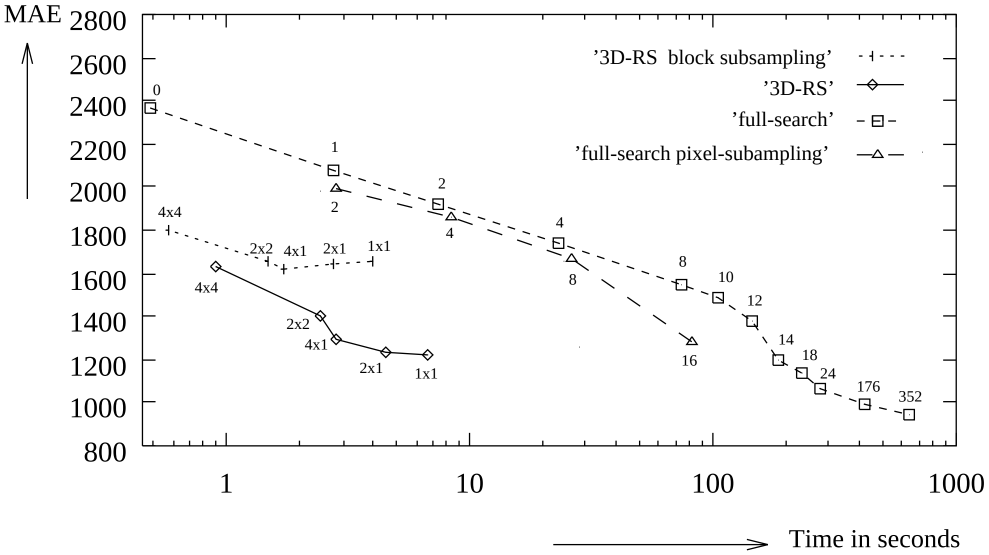

Figure 5.13: Overview of test to measure the quality verses execution time for the full-search and the 3D-RS estimators.

the execution time of different motion-estimators.

figure 5.13 the results of all these tests are represented. For the full-search two kind of test are performed. The first is the full-search with normal block-matching the other full-search test uses a pixel sub-sampling of 4 to 1. The number note by the full-search in the figure indicates the size of the search window. The results of the 3D-RS are also shown in figure 5.13. For the 3D-RS also 2 version are shown in the figure, the first is the 3D-RS with, and the second without the block sub-sampling option. The numbers of the 3D-RS test indicates the horizontal and vertical pixel sub-sampling ratio (i.e. 2x1 means a sub-sampling of 2 for the horizontal pixels). From the figure it is noticed that increasing the window size of the full-search to a value around 12-18 a noticeable reduction of the MAE is established. This is caused by the fact that the length of the motion vector for the HPRL-car sequence is about 15 pixels per frame.

For the motion-estimation test the vectors computed are the same as defined in the previous section. An I frame the only copies information of its own picture and hence the MAE is zero. A P-picture only copies information from its previous P or I picture. A B-picture can uses the picture information of the past and future I or P picture, just like the MPEG scheme.

From figure 5.13 it is concluded that the 3D-RS is performing better than the full-search with respect to the execution time and quality. In the figure also a performance difference between the 3D-RS with and without the block sub-sampling is noticeable. However in section 5.1 it is proven that the smallest MAE not always gives a better estimator. This is now emphasized by another test.

In figure 5.1 the full-search without sub-sampling and a window size of 14 pixels has a MAE of 1199. The 3D-RS algorithm with pixel-sub-sampling of 4×1 and block sub-sampling has a MAE of 1624. When the decision of the 'best' motion-estimator is made using the MAE the full-search estimator with a search window of 18 would be better since it is MAE is about 25% smaller. However, if both estimators are implemented in a MPEG encoder and the quality of the reconstructed sequence is tested the 3D-RS with pixel and block sub-sampling performs better. This is shown in figure 5.14. From

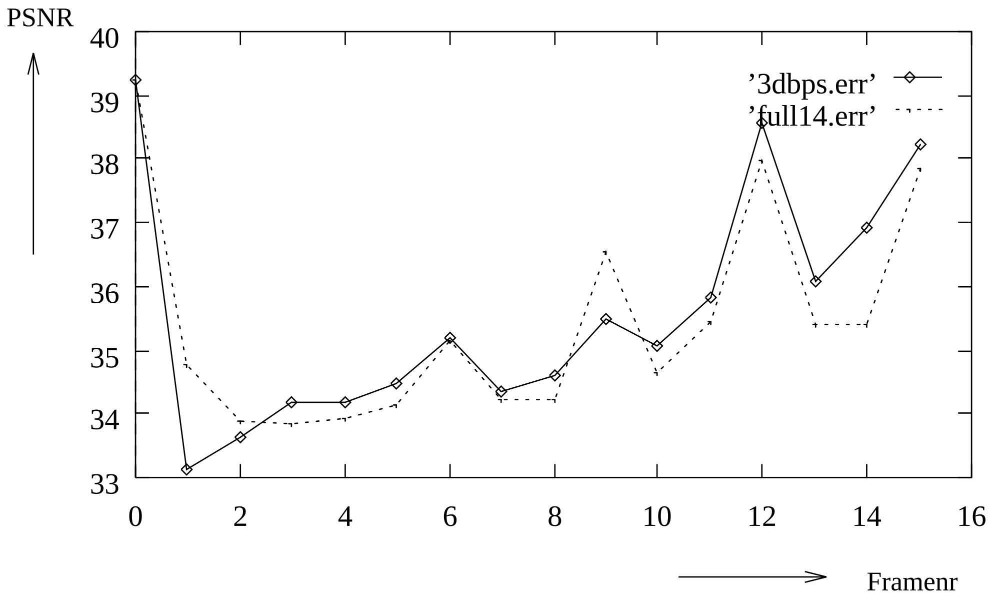

Figure 5.14: In the MPEG scheme the 3D-RS performs better than the full-search.

figure 5.14 it is observed that the 3D-RS perform slightly better. The First frames the 3D-RS performs worse, but this is caused by the fact that the 3D-RS needs to converge to the correct motion vector.

The second test is performed on sequences which are compressed by the MPEG codec scheme. The compressed and decompressed sequence is compared with its original, and the difference is a measurement for the quality. To test different motion-estimation techniques these are all implemented in the MPEG encoder. In table 5.1 the test for different sequences is shown. The tested motion-estimator are the 3D-RS and the full-search. For the 3D-RS all permutation of the block and pixel sub-sampling are tested. Since the 3D-RS in this report only produce full-pel vectors, this in contrary to the original full-search which produce half-pel vectors. The advantage of using half-pel vectors is that it increases the quality, a disadvantage is the execution time to compute it. To measure the influence of half-pel the full-search is tested with and without the half-pel search. The execution times noted in the tables 5.1 gives only an indication of the time to compute the motion-estimation, since also the remainder part of the MPEG and the quality of the decoded pictures are computed. For this test Garden, HPRL-car and the Mobile sequences are used and represented in table 5.1.

To test the difference of the true motion vector between the full-search and the 3D-RS the vector maps of some sequences are presented, see figures 5.15 - 5.23. From the vector map figures, in particularly the HPRL-car sequence, it is seen that the 3D-RS produce motion vectors which corresponds more with the true motion of the sequences.

Conclusion The 3D-RS improves a significant speed-up of the motion-estimation algorithm. The quality of the 3D-RS is overall slightly worse, but is acceptable.

Garden sequence 100 frames of (352x240) pixels

block subsamp.	pixel subsamp.	search-window	PSNR. (dB)	Ex. time (s)
-	-	2x2 half pel	26.73	173
-	-	2x2 full pel	25.65	153
-	-	1x1 half pel	24.88	148
-	-	1x1 full pel	23.74	127
1	1	-	26.34	72
1	2	-	26.29	66
1	4	-	26.23	61
1	8	-	26.10	60
2	1	-	26.07	68
2	2	-	26.05	64
2	4	-	26.01	60
2	8	-	25.91	60

HPRL-car sequence 139 frames of (352x288) pixels

block subsamp.	pixel subsamp.	search-window	PSNR. (dB)	Ex. time (s)
-	-	12x12 half pel	33.55	1810
-	-	12x12 full pel	32.67	1767
1	1	-	33.26	121
1	2	-	33.24	111
1	4	-	33.19	102
1	8	-	32.87	101
2	1	-	33.05	114
2	2	-	33.07	107
2	4	-	33.02	101
2	8	-	32.82	100

Mobile sequence 40 frames of (720x576) pixels

block subsamp.	pixel subsamp.	search-window	PSNR. (dB)	Ex. time (s)
-	-	12x12 half pel	35.40	1554
-	-	12x12 full pel	34.30	1506
1	1	-	34.11	140
1	2	-	34.09	127
1	4	-	34.04	117
1	8	-	33.97	115
2	1	-	34.04	131
2	2	-	34.02	123
2	4	-	33.97	116
2	8	-	33.94	114

Table 5.1: Test results of different sequences coded by the MPEG scheme using different motion-estimators.

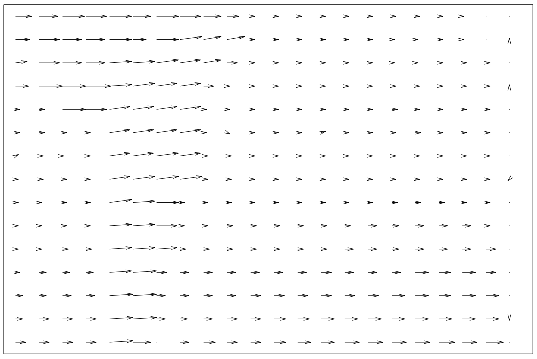

Figure 5.15: Vector map of a frame from the garden sequence produced by the 3D-RS motion estimator.

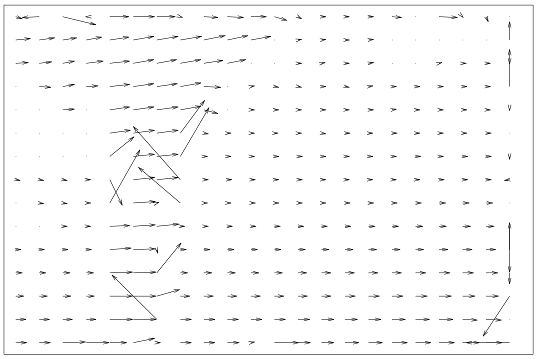

Figure 5.16: Vector map of a frame from the garden sequence produced by the full-search motion estimator with a search-window of 12.

Figure 5.17: Gray-scale picture of the garden sequence.

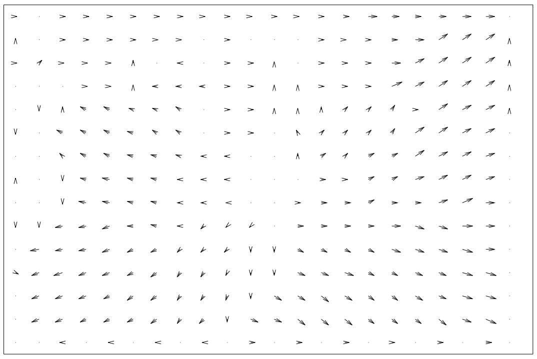

Figure 5.18: Vector map of a frame from the tennis sequence produced by the 3D-RS motion estimator.

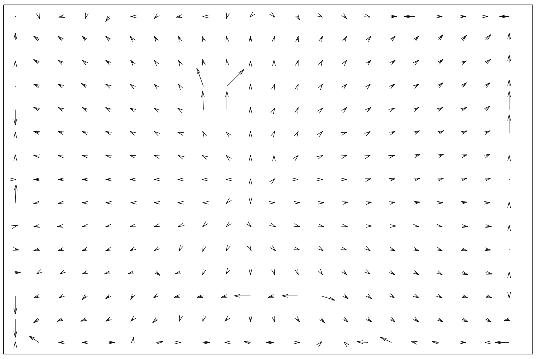

Figure 5.19: Vector map of a frame from the tennis sequence produced by the full-search motion estimator with a search-window of 12.

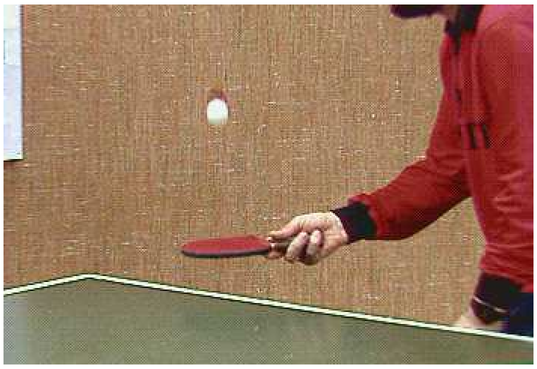

Figure 5.20: Gray-scale picture of the tennis sequence.

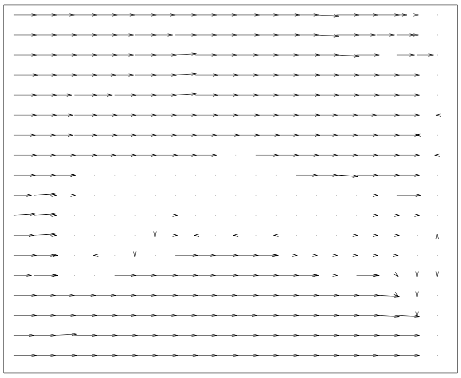

Figure 5.21: Vector map of a frame from the HPRL-car sequence produced by the 3D-RS motion estimator.

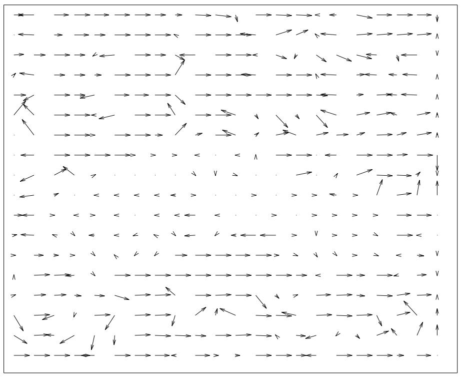

Figure 5.22: Vector map of a frame from the HPRL-car sequence produced by the full-search motion estimator with a search-window of 12.

Figure 5.23: Gray-scale picture of the HPRL-car sequence.

The MOVE framework 6

The MOVE framework is a processor design tool developed at the TU DELFT, faculty of Electrical Engineering, section Computer Architecture. The framework is a tool for designing special purpose processors within a sort design time. In this chapter a short overview of the framework is presented. In section the concepts of the MOVE frame work are given. In section 6.2 the compiler used in the framework is discussed. Section 6.3 gives an explanation about the description of the processor architecture. In section 6.4 the scheduler used in this report is presented.

6.1 The MOVE architecture

In this section a description of the MOVE framework is given. The MOVE framework uses *transport-triggered architectures* (TTAs) instead of operation-triggered architectures (OTAs). With the OTAs architectures the operations and operands are specified by a instruction stream and are executed by the processor. TTAs only have one instruction which is the MOVE operation and moves data from registers to registers. The operands of the move instruction are specified by the source register, and the operation is specified to the FU were the move instructions are moved to. A move processor consists of functional units (FU) which perform a specific operation. The FU are connected by one or more buses via the FU registers. The register set of the FU is divided by

1. operand (O) register
2. trigger (T) register
3. result (R) register

To perform an operation with two operands the first one is moved to the O-register of the FU. The second operand is moved to the T-register and the FU is triggered to start its operation. After some time the FU has computed the result of the operation and is available in the R register of the FU. The result in the R register can now be moved to another operand register or to a general purpose register (GPR) for later use. These GPR registers are combined and connected to the buses. To perform branches in the instruction flow the Program counter (PC) is visible for the programmer and a branch is performed by moving a new address in the PC. Conditional execution of instructions is done by adding a guard to the instructions. A compare unit computes a boolean result. If the logic result of the output of the compare unit (the boolean result) equals the guard then the instruction is executed. When this guard option is used with a branch then it's possible to perform conditional branches. More information about the move framework is found in [3]. The advantages of the MOVE frameworks are:

- The performance of the processor is scalable by adding extra functional units, busses or installing faster FU. The changes of the instruction program in the three cases are all predictable and straight forward.
- For implementation of common FUs one can use a library were these FU are already specified.
- The design time with the MOVE framework will be short because the implementation is simple. The short design time means less cost for designing the processor so the performance-cost ratio will increase.

6.2 The MOVE compiler

With the MOVE framework Application Specific Instruction set Processors are designed (ASIP). To use such an ASIP the designer should design the program which contains the instructions, and the processor which executes the instructions. In this section the compiler which produce the MOVE code from the high level language is discussed.

The compiler translate the high level language C/C++[1] into the sequential MOVE code. The compiler is derived from the GNU C Compiler and is called gcc-move. Along with the compiler also a linker (gls-move), an assembler (gas-move) and the BSD libc and libm libraries are designed. The register set used by the compiler is:

1. A 128×32 bits integer register file (`r0..r127`)
2. A 128×64 bits floating point register file (`f0..f127`)
3. A jump register (`jump`)
4. A call register (`call`)
5. A trap register (`trap`)
6. A one bit boolean register (`bool`)

[1] In the future the support for the language Fortran will also be supported

7. A return address register (`ra`)
8. Operand, trigger, and result registers for the supported operations

The operation-set used by the compiler is:

integer arithmetic	: `add`, `sub`, `mul`, `div`, `divu`, `mod`, `modu`
integer conditional	: `eq`, `gt`, `gtu`
floating point arithmetic	: `addf`, `subf`, `negf`,`mulf`, `divf`
floating point conditional	: `eqf`, `gtf`
load/store	: `ld`, `ldb`, `ldd`, `ldh`, `lds`, `st`, `stb`, `std`, `sth` ,
	: `sts`, `sxbh`, `sxbw`, `sxhw`
logical	: `and`, `ior`, `xor`, `shl`, `shr`, `shru`
type conversions	: `f2i`, `f2u`, `i2f`, `u2f`, `insb`, `insh`, `extb`, `exth`
User-defined operations	: `usr1`, ... `usr10`

The last operation, the usr1 ... usr10 are operation which are used to design special user defined operations. These user defined operations are executed by special FUs.

The instruction set given above can also be decreased. This is profitable if some instructions are not (or almost not) used. These instruction are then simulated by instructions the compiler does support. For example the floating point operations in the programs which use only integer arithmetic could be disabled. The advantage is that the processor should not support floating point operations and does not have to posses a floating point function unit.

To get a better insight of the compiler an example is given,see 6.1. Here a source C code is compiled to sequential MOVE code. For the MOVE code there is one operation per line. To performs one operation there are one or more MOVE instructions necessary.

6.3 Description of a TTA

The second part of the of the MOVE framework is the description of the MOVE processor. Since the structure of TTA are very simple they are also described very simple. Figure 6.2 shows an example of a TTA, containing the following components:

- Nine busses
- Two immediate units
- One instruction unit
- One Boolean registerfile
- One floating point registerfile
- One integer registerfile
- One floating point unit

```
    Source c code

int foo(int arg[])
{
        int i;

        for(i = 0; i < 10; i++)
                bar(arg[i]);

        return 0;
}




    MOVE code generated by the gcc-move compiler.


 _foo:   sp -> st_o; fp -> st_t;
         sp -> fp;
         fp -> sub_o; 4 -> sub_t; sub_r -> sp;
         sp -> st_o; ra -> st_t;
         /* deleted function prologue code */
         0 -> r8;
         r3 -> r7;
 L0:     r7 -> ld_t; ld_r -> r3;
         _bar -> call;
         r7 -> add_o; 4 -> add_t; add_r -> r7;
         r8 -> add_o; 1 -> add_t; add_r -> r8;
         r8 -> gt_o; 9 -> gt_t; gt_r -> bool;
         !L0 -> jump;
         0 -> rv;
         /* deleted function epilogue code */
         sp -> ld_t; ld_r -> ra;
         fp -> sp;
         fp -> ld_t; ld_r -> fp;
         ra -> jump;
```

Figure 6.1: Example of the source code and the MOVE code produced with the gcc-move compiler.

- One ALU unit
- One load store unit
- 27 sockets

To describe a processor like figure 6.2 the MOVE framework use the so called "machine description file". The machine description is a text file which contains a number of section which describes the different object of a MOVE processor. These sections are:

- MoveBusses
- Sockets

- FunctionUnits
- ImmediateUnits
- RegisterUnits
- InstructionUnit

In appendix D a example of the mach file which describes the processor of figure 6.2 is given.

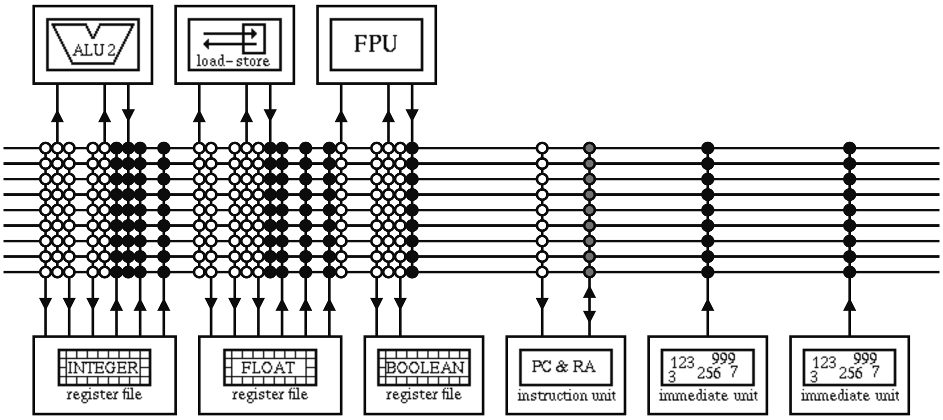

Figure 6.2: transport it.

6.4 The MOVE scheduler

The scheduler in the MOVE framework translates the sequential move code produced by the compiler to a parallel move code. To produce the parallel code the scheduler uses the machine description file to use the recourses of the processor. The scheduler also use the profiling information of the program.

7 Results

Up till now the SSG– MPEG-2 program is evolved into the Fast MPEG program by optimizing the MPEG algorithm in the high level language C. The important optimized algorithms are the motion-estimation and DCT. The applied optimizations were discussed in chapters 4 and 5. The optimizations let to a considerable speed-up of the execution time of the SSG– MPEG program (about 22 times). The Fast MPEG program is then used to develop an architecture to implementation the MPEG algorithm into VLSI.

In section refch.res.overview the Fast MPEG is compared with the SSG– MPEG. In section 7.3 the Fast MPEG C program is compiled to MOVE code and a processor architecture is developed. In section 7.4 the Fast MPEG is rewritten were the whole structure of the program is changed in such a way that the program works on a macroblock basis, instead of a frame basis. This program is called the Macro Block (MB) MPEG program. An advantage of this structure is that the program is more adopted to use special function units, which are used in this version of the program. In section 7.5 the program is split up into four parts, so a multi-threaded program is created. The multi-threaded program runs on four different processors which communicate with eachother. The advantage of using multi-threaded programming is that the execution time is decreased, and the use of special function units is more effective.

7.1 Overvoew of the presented results

In this chapter several versions of the Fast MPEG are mapped to a MOVE processor. There are also two version of the Fast MPEG program developed, these are called the MB MPEG program and the multi-threaded program. These programs exploit the possibilities which arise from using special hardware, like special function units and multi-threading techniques. To get a better overview of the different implementations of the Fast MPEG and other programs, an overview of the developed processors is draw in figure 7.1. For all test performed in this chapter an input sequence of 25 frames of 720×576 pixels is used. These tests in this chapter should thus be executed for a **real-time MPEG encoder in 1 second**. The description of the different MPEG program

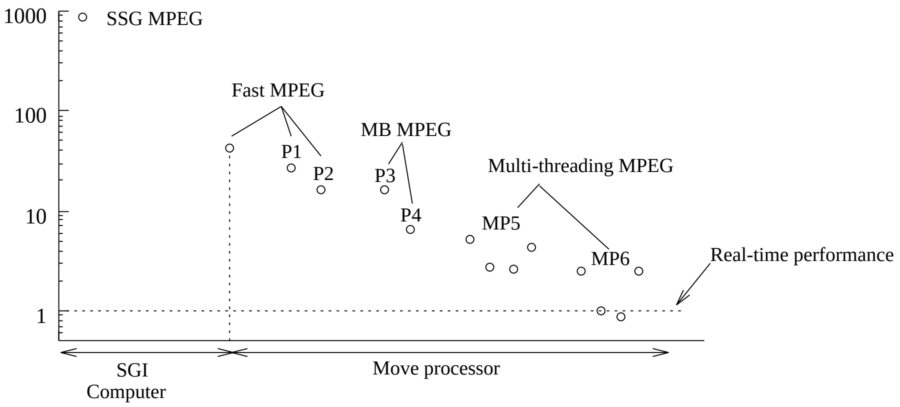

Figure 7.1: Overview of the MPEG programs and the different implementation.

versions are:

1. **SSG– MPEGr** The SSG MPEG is the original MPEG encoder program written in the high level language C. The execution time of this program for a CCIR sequence is about 935 seconds.

2. **Fast MPEG** The Fast MPEG program is the optimized version of the SSG–program. The speed-up achieved is 22 with a reduction of the quality of about 5%.

   P1 The Fast MPEG program is compiled to MOVE code and simulated on a MOVE processor, P1 with the same 'functionality' as a MIPS 4600 processor used in a SGI computer. With this test the ability of the MOVE processor to exploit the instruction level parallelism is demonstrated.

   P2 To improve the execution time of P1 the Fast MPEG program is mapped onto a P2 processor with much more function units. The cost of this processor is about 5.2 times more expensive than the P1 processor. The speed-up realized however is only 2.15 times.

3. **MB MPEG** To improve the execution time of the previous processors the Fast MPEG program is rewritten into a macroblock basis and adapted to use special function units.

   P3 The MB MPEG is mapped to the P3 processor. For the P3 processor special function units are added which support the on-chip memory and the adds, subs and muls operation designed for the DCT algorithm. The execution time of the program on processor P3 is about 12 sec, which still is not satisfying.

P4 To improve the execution time of P3 the Fast MPEG program is mapped on processor P4 which improved the execution time by a factor 2 the cost however of the processor is increased with a factor of 5

4. **Multi-threaded MPEG** Here the MB MPEG program is split up in four different programs which run on different processors. The execution time of the complete encoder equals to the execution time of the slowest processor.

MP5 The multi-threaded MPEG program is mapped onto four processors. The fasted processor is still a factor three to slow.

MP6 The execution time of the four processors is again decreased by mapping the MOVE code onto large processors again a speed-up of about 2 is established

The processors P2, P4 and MP6 are only tested in this chapter to show that increasing the number of function units of the processor does not decrease the execution time of the program inverse proportional. From the test with the P2, P4 and MP6 processors also the procedures with parallelism are identified since these procedures will give a better relative speed-up than the remainder procedures. In the following section the different MPEG programs and their implementations are discussed.

7.2 A comparison between the Fast and the SSG– encoder.

Before mapping the Fast MPEG program onto a processor with the MOVE framework first the difference between the SSG– MPEG and the FAST MPEG program are presented. By designing the Fast MPEG program four important factors to optimize the SSG– program were taken into account. These are:

Execution time. The execution time of the SSG– program is, for a search window of 12, about 1000 seconds. To execute this program in real-time (i.e. 1 second) a major speed-up must be established.

Encoder quality. Although the execution time is important, the quality should be taken in considerance. The Fast MPEG in this report shows a small decrease of the SSG– quality. This in contrast to other encoder implementations like the MPEG encoder chip of IBM. This implementation produces I-frames only, and will probably produce a lower quality for equal bitrate.

Cost. The cost is defined as the VLSI area used by a specific implementation. In the SSG– MPEG program the DCT uses floating point and the IDCT uses 32 bit arithmetic for computing their algorithms. The algorithms of the Fast MPEG are designed to compute in 16 bit arithmetic. With this implementation some speed-up and quality is afforded but in return the cost is decreased significantly.

No average cheating. The Fast MPEG program is written in such a way that it does not depend on the statistics of the input data. Many optimization are possible

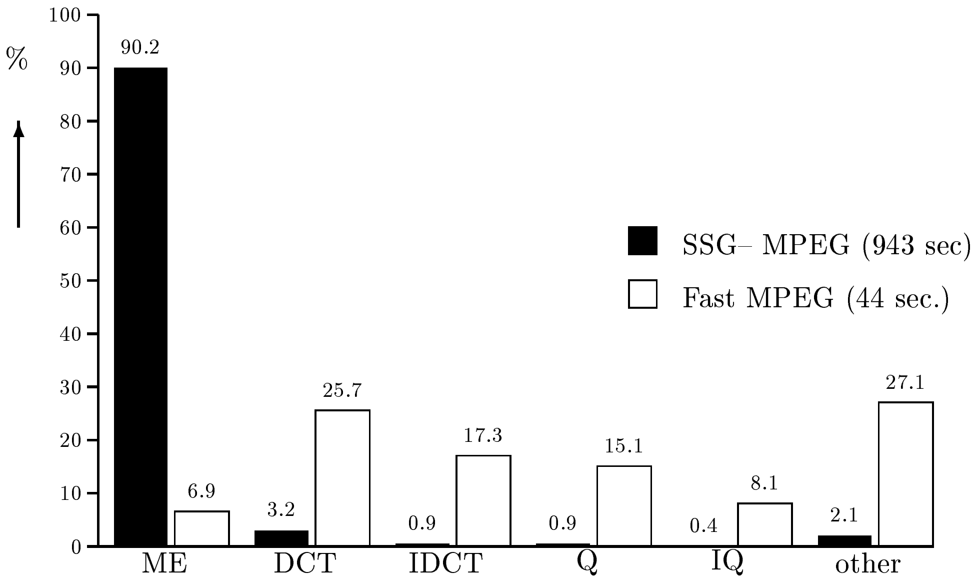

Figure 7.2: The execution time distribution of the Fast MPEG and the SSG– MPEG encoder.

if one exploit the advantage of the statistically properties of the input data. For example the IDCT is speed-up significantly if the row and column are tested for zeros before computed. However in a worst case scenario these statistics can not be used and hence the speed-up is not established.

The execution time distribution of the SSG– and the Fast MPEG programs are given in figure 7.2. The distribution of the different algorithms are normalized to their total execution time. The total execution time for the SSG– for 25 pictures of the *mobile* sequence (720 x 576 pixels) is 935 seconds on a SGI computer. The Fast MPEG program executes the same test in 44 seconds. In the Fast MPEG program the 16 bits version of the IDCT is not yet implemented. The Fast MPEG algorithm also produces a different bitstream then the SSG– since the routines for macroblock type decision are changed to produce faster code with some decline of the quality of the encoder.

From figure 7.2 it is clearly seen that the distribution of the algorithms are significantly changed. The motion-estimation (ME) is decreased from 90.2% to 6.9%, which is caused by replacing the full-search ME with the 3D-RS ME algorithm. Further, it is seen that the absolute execution time of the Fast DCT algorithm is decreased by a factor of 2.7.

7.3 Synthesis of a MOVE processor.

In this section a MOVE processor is developed which executes the Fast MPEG program. First the Fast MPEG program is compiled to sequential MOVE code. Second a file is created which describes the architecture of processor P1. This file is called the machine description file. An example of a description file is given in chapter 6.3 The machine description file of processor P1 and the sequential MOVE code of the Fast MPEG program is processed by the scheduler. The scheduler translates the sequential MOVE code to a parallel MOVE code which can be executed by processor P1.

In the section 3.2 it was suggested that the MOVE framework is capable to exploit the instruction level parallelism in programs. To test this suggestion a MOVE processor is designed which is closely related to the processor in the SGI computer we used for all the previous tests. The CPU processor of the SGI is imitated by a MOVE processor with one ALU, one LOAD/STORE and one FPU unit. These units are fully connected with eight data busses. Fully connected means that every register of a function unit is connected to all the busses. With this processor the Fast MPEG program is executed and the results are given in figure 7.3. The figure gives an overview of the distribution of the cycles used by every procedure. For the MOVE processors the time to execute a program is notated in the number of cycles the program is computed. The overall cycle time of the Fast MPEG program executes on a SGI computer is 3927 Mega cycles and the numberof cycles for the MOVE processor is 2615 M cycles. Exploiting the instruction level parallelism of the Fast MPEG program a speed-up of 1.48 has been achieved.

To improve the execution time of the Fast MPEG program on the P1 processor an attempt is made to construct a processor with 15 function units which support all instructions. The VLSI area used by such a processor is to big to fit on a normal chip area, however, it gives an idea how-much impact increasing the cost of the processor has on the execution time. The ratio of the cost of processor P1 and P2 is 5. The speed-up of the execution time however is 2.15. Hence from this experiment it is concluded that increasing the number of function units and thus the cost of the processor does not give the desired solution.

With the previous experiment the assumption is made that the cycle count could represent the execution time of the program. This assumption holds if the cycle time of both processors are equal. However, the cycle time in the MOVE framework depends on the number of sockets connected to the busses. To give an indication about the influence of the number of bus connections and the cycle time, the cycle time for processors P1 and P2 are estimated. From the estimations it is concluded that processor P1 has 80 % faster cycle time than processor P2.

Conclusions In the previous tests it is shown that increasing the number of FUs does not yield the correct solution. However the execution time of the Fast MPEG encoder program on processor P1 is still about 26 times to slow to compute CIRR video sequence in real-time. In the next section another solution is presented which does give a better execution time.

7.4 Optimizing the Fast MPEG program structure

To improve the execution time of the Fast MPEG program this program is rewritten into the another version of the Fast MPEG called the MB MPEG program. By changing the structure of the Fast MPEG program it is possible to exploit an on-chip memory. The on-chip memory is simulated by special function units which simulate the load and store of the on-chip memory. The advantage of using an on-chip memory is that the loads and stores are executed in one clock cycle. To utilize the on-chip memory the Fast

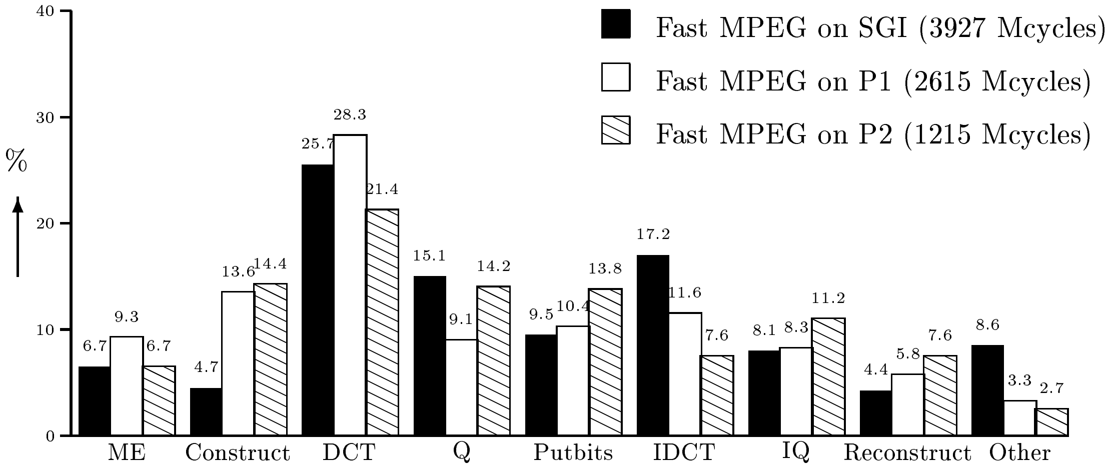

Figure 7.3: The execution time distribution of the Fast MPEG executed on a SGI and a MOVE processors.

MPEG program is rewritten into the MB MPEG program. The main difference of these programs are shown in figure 7.4. The Fast MPEG program executes one operation (like the DCT, Quantize, Construct, etc.) on all the macroblocks in the pictures. The MB MPEG executes all operations for one macroblock before execution the next macroblock.

With the MB MPEG program the on-chip memory is used to store the macroblock temporally for every operation. A macroblock is loaded from main memory to the on-chip memory by the Predict_macroblock, see line 4 of the pseudo code of MB MPEG program. The macroblock is then handled by the DCT operation. The DCT program now reads its data from the on-chip memory. The result from the DCT are again stored in the on-chip memory and the program will execute the quantization operation. This process is repeated for all operation in the main loop of the pseudo code of the MB MPEG program. In figure 7.7 the data-flow of the macroblock and the operation on the macroblocks are given.

To support an on-chip memory which stores the data of the macroblock the on chip memory should contain 6×64 words. The on-chip memory contains thus 6 blocks of 64 bytes of memory: this in similarity of the macroblock construction of the MPEG scheme (see figure 2.4 for more information). However, in the implementation of the MB MPEG program there are 7×64 words of memory. The extra block of 64 words is used for temporal storage. This is illustrated with an example where the DCT of block number 2 is computed. The DCT is computed in 5 stages, see figure 4.3, the first stage the data is read from block 2, the answers are stored in block 7. For the second stage the data is read from block 7 and stored in block 2 again. Since the read and write operation on the on-chip memory are always performed on the separated blocks the read and write

```
        pseudo code of Fast MPEG main loop.

LINE    Code

     For every macroblock in picture do Construct_motion_vectors
     For every macroblock in picture do Predict_macroblock
     For every macroblock in picture do Sub_macroblock
     For every macroblock in picture do DCT_block
     For every macroblock in picture do Quantize_block
     For every macroblock in picture do Put_macroblock
     For every macroblock in picture do Inv_Quantize_block
     For every macroblock in picture do Inv_DCT_block
     For every macroblock in picture do Add_macroblock




        pseudo code of MB MPEG  main loop.

LINE    Code

     Construct_motion_vectors
     For every macroblock in picture do
          {
           Predict_macroblock
           Sub_macroblock
           For i=0 to 5 do
             {
              DCT_block
              Quantize_block
             }
           Put_macroblock
           if I or P picture
             {
              For i=0 to 5 do
                {
                 Inv_DCT_block
                 Inv_Quantize_block
                 }
              Add_macroblock
             }
          }
```

Figure 7.4: Difference between the main loop of the Fast MPEG and the MB MPEG program.

operations are in-depended. Independent means that all read and write operations can be scheduled freely without producing data hazards.

To support this feature the on-chip memory is not addressed linearly, but on a block based index.

So for all the read and write operations in stage there are no data dependencies. To exploit this property there is an operation which identifies on which blocks the read and write are performed. In figure 7.5 and implementation of the addressing of the on-chip memory is given.

The supported function on the on-chip memory are:

SetPort The SetPort select the block were the read and write operations are performed on. The read and write are never performed on the same block. The values set by the SetPort are stored in the Block Write Select (BWS) and the Block Read Select (BRS). See figure 7.5 for more information.

Write Write performs a write on the on-chip memory. The input values are a Write Address (WA register) in the range of 0 to 63 and the value which should be stored at the memory location. The block number where the value is stored depends and the value of the BWS which is set by the SetPort operation. A formula notation for a linear addressing is: $Mem(64 \times BWS + WA) = value$.

Read Read performs a read operation on the on-chip memory. The input value is an address (RA register) in the range 0 to 63. The actual return value depends on the value of the address and the value of the BRS value. A formula notation for a linear addressing is: $value = Mem(64 \times BRS + RA)$.

The main advantages using the *SetPort, Write and Read* operation on the on chip memory is that the writes and reads become independent. The scheduler has now a large flexibility in scheduling the MOVE operations. In the MB MPEG program there are

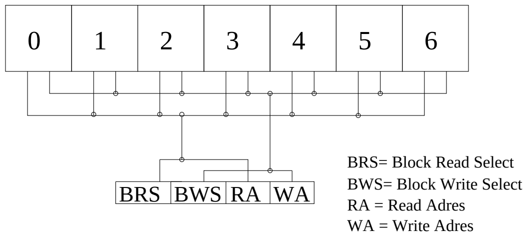

Figure 7.5: Overview of the macroblock structure of the Fast MPEG program.

also other optimization exploited. These are:

- The reference path of the encoder which compute the IDCT, IQ and the Reconstruct macro block are only executed for I and P pictures. Only the I and P type pictures are necessary as a reference frame for the motion-estimation and the reconstruct algorithms. For every mini GOB now only one picture is reconstructed instead of three.
- The original IDCT is replaced with an IDCT algorithm which use 16 bit arithmetic.
- There are special function units designed which execute the functions *adds*, *subs* and *muls* used by the IDCT and DCT algorithm.

The result of the optimizations of the high level language and the utilization of the special function units proposed led to a decrease of the execution time of a factor of

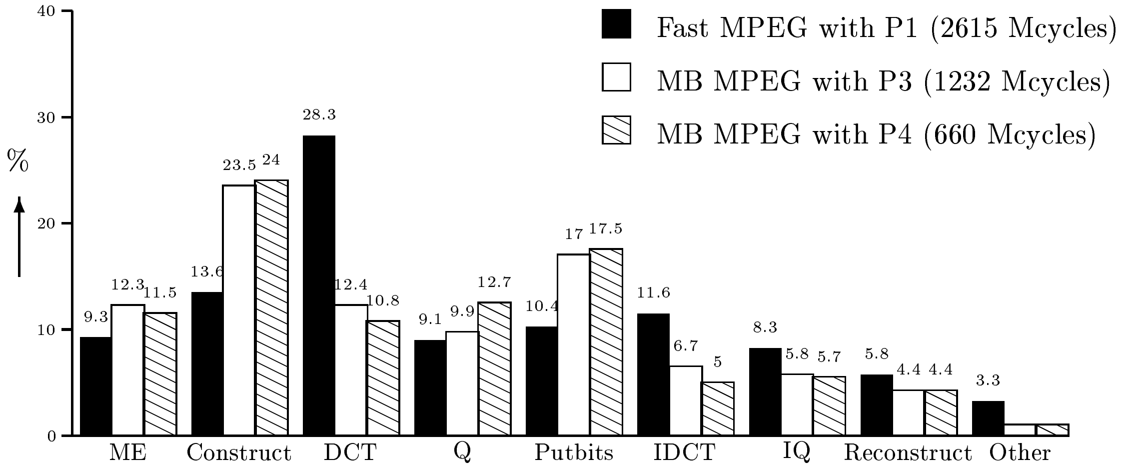

Figure 7.6: The execution time distribution of the MB MPEG program on a MOVE processors.

two. The relative cycle count for the MB MPEG program is given in figure 7.6 for the processors P3 and P4. The implementation of the Fast MPEG on the P1 processor is also given for comparison. From the figure it is concluded that the difference of the execution time of the DCT on processor P1 and P3 significantly reduced. The cause of this decrease is the use of the special function units. The Putbits operation of the MB MPEG program shows an increase of the execution time. This is caused by the fact that the on-chip memory of the P2 and P3 processors is not yet fully exploited. The decrease of the IDCT, IQ and the Reconstruct operation is caused by the fact that only I and P type picture are reconstructed. So the speed-up is not achieved by a better performance of the IQ and the Reconstruct algorithms, but the IQ and the Reconstruct are about three times less executed.

7.5 Multi-threading of the final encoder

From the previous chapter it is shown that the MB MPEG reduced the execution time by a factor of 2. The implementation with processor P4 indicated that the MB MPEG is still to slow to compute in real-time. To improve the execution time more special function units and code optimization could increase the execution time, however the speed-up will not be enough. This is concluded from the cost and execution time performance of processors P3 and P4. On the other hand the parallelism upper-bound of the MB MPEG program is about 430 operation per cycle. This indicates that if there are unlimited hardware resources 430 operations could be executed in one clock cycle. On the contrary increasing the number of function units of the processor does not yield the solution.

This is concluded by mapping the MB MPEG program to processor P4. To solve this problem another solution is tried. In the previous section it is showed that the MB MPEG program operates in a number of stages. The stages consists of one operation on the macroblocks (i.e. Construct, DCT, Q .. etc). A speed-up is established if the stages of the MB MPEG program are operated in a pipelined fashion. This means that in stage 1 the first macroblock is processed by the DCT. In the second stage this macroblock is processed by the Quantizer (Q), but the second macroblock is now already processed by the DCT. To use this feature the MB MPEG program is rewritten as a multi-threaded program and is called the multi-threaded MPEG program in this report. A multi-threaded program means that the program is split up in separate programs, which run on separate processors[1]. The communications of the separate programs is done by FIFO (First In First Out) memory. The operation to send or receive a data-word from/to the FIFO is implemented by Special Fuction Units (SFU). For the simulation of the communication of the multi-threaded MPEG the UNIX socket feature is used. This feature transmits packets of information over the network to the other computers. The multi-threaded MPEG program consists of fout parts. In figure 7.7 the separate part are indicated, these are:

Part 1 The motion-estimation and construct macroblock algorithm.

Part 2 The Discrete cosine transform (FDCT) algorithm.

Part 3 The quantize (Q) and the Putbits algorithm.

Part 4 The inverse quantizer (IQ) , inverse discrete cosine transform (IDCT) and the Reconstruct algorithms.

It should be noticed that the decision of splitting the program as indicated above is made on difficulty of designing the program. The first and the fourth part of the multi-threaded MPEG uses the same reference frames stored in memory, see figure 7.7. To support this shared memory option in a unix environment these parts must run on the same computer . It should further be noticed that to synchronize the first and fourth part of the multi-threaded program, semaphores are used.

The multi-threaded MPEG program is also mapped on MOVE processors. Each part of the program is mapped on a separate processor. The software for the communications of the different parts of the multi-threaded program is rather complex and slow. In the MOVE framework a easier solution is used and the communications are performed by two operations. These operations are the SendWord (sw) and the ReceiveWord (rw) and are executed by SFUs. The special function units are FIFO memory buffers and are easy to implement in hardware. The results of mapping the multi-threaded MPEG program onto four different MOVE processors are presented in one graph; see figure 7.8.

Figure 7.8 shows the Mega cycles used to process 25 frames of the mobile sequence. If the processors run with a clock of 100 Mhz the slowest part of the multi-threaded program takes about 2.3 seconds for the MP6 set of processors. Although the MP6 implementation is still a factor two to slow and uses very big processors the results are still promising if the following is taken into account:

[1] In this case one processor executes one program.

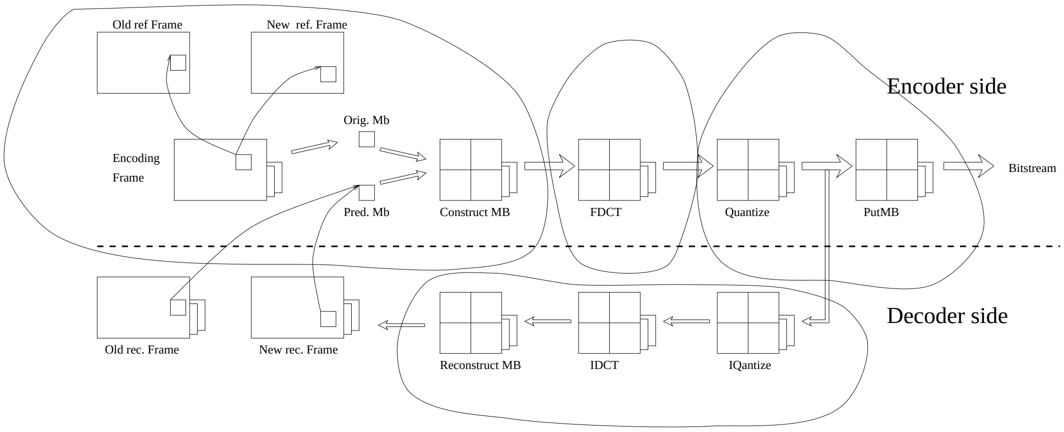

Figure 7.7: Overview of the macroblock structure of the Fast MPEG program.

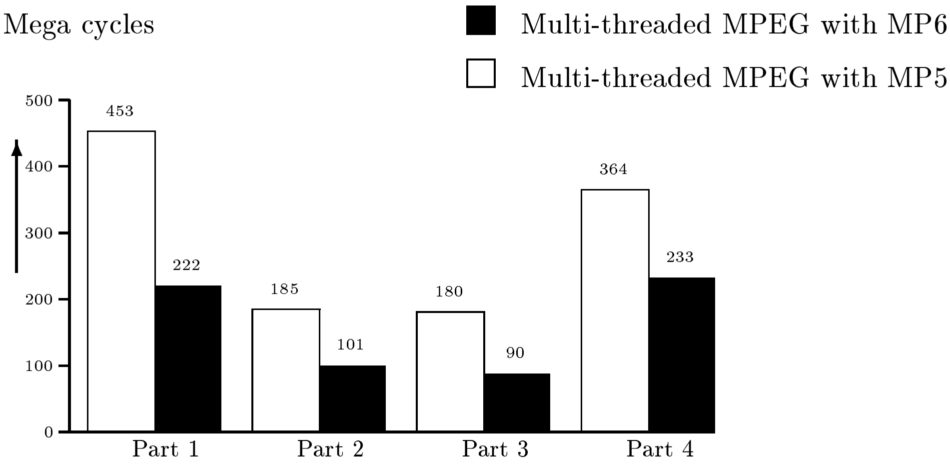

Figure 7.8: The execution time distribution of the MB MPEG program on a MOVE processors.

- The multi-threaded program can be further separated into smaller parts. Part 1 is easily divided into a motion-estimation and a construct part. Further the quantizer and the Putbits part can be separated. Although this would involve some problems since the quantizer depends on the Putbits operation.

- The use of the special function units in the multi-threaded program is more profitable since these are relative more used. In the four parts there are a number of options which could be optimized using special function units.

- The C-code of the program can be more optimized. A special case is the loop-unrolling option. Most algorithms in the multi-threaded MPEG program contains the following loop construction

```
for (i=0; i<64; i++)
  {
   .... c code ....
  }
```

If the loop is unrolled 64 times the code is better scheduled by the scheduler of the MOVE framework. Premature results showed a speed-up of 13% for the MP5 processors and a speed-up of 50% for the MP6 processors. An extreme example of the loop unrolling speed-up is demonstrated by the following example of the C-code of the main loop of the second part (FDCT) of the multi-threaded program:

```
for (jp=0; jp<height; jp+=16)
    {
     for (ip=0; ip<width; ip+=16)
        {
         sw(1,rw(0));                /*   receive and send the  MB type  */
         sw(1,rw(0));                /*   receive and send the  Motion vectors (4)  */
         sw(1,rw(0));
         sw(1,rw(0));
         sw(1,rw(0));
         sw(1,rw(0));                /*   receive and send the  MB calc  */
         for (n=0; n<6; n++)i        /* process 6 blocks  */
            {
             sp(1,0);
             for (q=0; q<64; q++)
                 wr(q, rw(0));       /* receive the elements of the MB */
             fdct(n);                /* perform the FDCT algorithm  */
             sp(0,1);
             for (q=0; q<64; q++)
                 sw(1,rd(q));        /*  send the elements of the MB */
            }
         sp(106,1);                  /*   flush buffer nr 1  */
        }                            /* end of slice of Macroblock */
    }                                /* end of screen */
```

The most important routine in this code is the fdct algorithm. The code of the fdct routine is given in appendix C. The relative execution time for the MP6 implementation is for the fdct routine 36% and for the code given above exclusive the fdct routine is 63%!

- The cost of implementing the set of MP6 processors is high, but if the processors are optimized by the explore [16] tool of the MOVE framework the cost of the processor is decreased significantly with a small decline of the execution time.

Conclusion The use of multi-threading programming is a good solution for optimizing the MB MPEG program. Further the use of special function units used in the multi-threading program is more profitable since the relative probability of the operation performed by the function unit is higher.

Conclusions and further research

8

The goal of the research is to investigate the ability of the MOVE framework to produce ASIPs for the MPEG video compression codec. The research is split up into two different subjects. The first is the developing of a MPEG program written in a high level language. Second the developed MPEG is mapped onto ASIPs using the MOVE framework. In section 8.1 the conclusion of this research is given. Although the results of the investigation indicates that the results are promising, still further research is necessary to produce an architecture design of an ASIP processor. In section 8.2 an overview of the proposed further research are given.

8.1 Conclusions

- The 3D-RS algorithms produce good true motion-estimation vectors. Further is the execution time to compute the vectors is significantly smaller for the 3D-RS then the full-search motion-estimator.
- The Fast MPEG encoder is a qualitative good encoder since the quality is 5% less than the original encoder of the Software Simulation Group. The execution time off the Fast MPEG encoder is 2200% faster then the original encoder.
- The use of the special function units in the FDCT algoritm led to produce a qualitative good results. Further is the execution time small and is the cost to implement the algoritm low.
- The use of of the MOVE framewoks makes it easy to design architectures. This research took about 9 months were for about 85% of the time was spend devel-

loping the MPEG software. The rest was spent designing the architectures for the proposed ASIPs.

8.2 Further research

- The SSG– MPEG encoder should be extended to support interlaced video sequences.
- The MPEG-2 standard is still in development. The changes made which are not provided or used by the current implementation should be added if appropriate or necessary.
- The 3D-RS should be extended to produce information about the fact if a motion vector is a true motion vector. In the vector maps from figure 5.21 it is seen that the vector at the right edge are not correct estimated. This information could be used by the macro block type decision algorithm.
- The 3D-RS produce full-pel vectors. The quality of the encoder is increased if the 3D-RS is extended to produce half-pell vectors.
- The macroblock type decision must be extended to support I type macro blocks in B and P pictures. This decision can be made an the information provided by the extended 3D-RS algorithm. Further the null motion macro block type should be better investigated.
- The inverse DCT transform algorithm should be better optimized. first the maximum input values must be determined. When this information is known a better implementation could be used.
- Multi-threading made it easier to support special function units Some special function units which can exploit the space it is given by the multi-treading opportunities are the
  - Cost function in the block matching algorithm.
  - The Putbits function.
  - The clip functions in the inverse quantize routines.
- Better special function units are possible if the scheduler could handle more than one result per operation. The scheduler should be extended to support this option.
- The scheduler utilize the possebility of operand sharing. However the use of this option can not be exloid fully by non commutative binairie operations.
- The simulator which simulate the MOVE code should be extended to simulate the stalls produced by the FIFO buffer construction used in a multi processor application.

- The length of the vectors of the motion-estimation are mapped to a length of 128 pixels. The MPEG encoder could if appropriate decrease the maximum vector lenght.
- The quantize algorithm in the should better investicated for better optimization of the algorithm.

Bibliography

[1] ISO-IEC JTC1/SC2/WG11, *Committee Draft of Standard ISO11172, Coding of moving pictures and associated audio*, MPEG90/176 rev.2 ed., Dec. 1990. Draft.

[2] N. Ahmed, T. Natarajan and K.R. Rao *Discrete Cosine Transform,* IEEE Transactions on Computers, January 1974.

[3] H. Corporaal and J.M. Mulder, *MOVE:A framework for high-performance processor design,* Supercomputing 91, Albuquerque, USA, November 1991.

[4] N.I. Cho and S.U. Lee, *Fast Algorithm and Implementation of 2-D Discrete Cosine Transform,* IEEE Transactions on Circuits and Systems, vol. 38, no. 3, March 1991.

[5] B.G. Lee, *A New Algorithm to Compute the Discrete Cosine Transform,* IEEE Transaction on Acoustics, Speech and Signal Processing, Vol. ASSP-32, No. 6, December 1984.

[6] M.J.Q. van der Lem, *Classification of fast DCT algorithms,* report, Delft University of Technology, 1994.

[7] G. de Haan, *Motion estimation and compression,*

[8] H.S. Hou, *A Fast Recursive Algorithm For Computing the Discrete Cosine Transform,* IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. ASSP-35 No. 10, October 1987.

[9] F. van Camp, *Real-time motion estimation within the MOVE framework,* report, Delft University of Technology, 1993.

[10] Srinivasam and Rao *Predictive Codinf Based On Efficient Motion Estimation,* IEEE transactions on Communication, No. 8, 1985, pp. 888-896.

[11] Biemond and Lagendijk, *Digitale signaal codering,* colege dictaat, Delft University of Technology, 1989.

[12] K.L. Gong and L.A. Rowe Parallel MPEG-1 Videao Encoding,

[13] G. de Haan, and H. Huygen, *Motion estimator for TV-picture enhancement*, Philips research Laboratories, 1990.

[14] T. Koga, K. Iinuma, A. Hirano, Y. Iijima and T. Ishiguro, *Motion compensated interframe coding for video conferencing*, Proceedings of Nat. Telecommunications Conference (New Orleans, LA), Nov. 29-Dec. 3, 1981, pp. G5.3.1-5.3.5.

[15] B. Liu, and A. Zaccarin, *New Fast Algorithms for the Estimation of Block Motion Vectors*, IEEE Trans. Circuits and Systems for Video Technology, vol. 3, no.2, pp.148-157, April 1993.

[16] H. Corporaal and J. Hoogerbrugge, *Code Generation for Transport Triggered Architectures*, Code Generation for Embedded Processors, Kluwer Academic Publishers chapter 14 pp. 240-259, 1995.

[17] Chen and Wang

Flowgraphs

A

In this report several flowgraphs are displayed because they give an easy overview of complex formulas. In this appendix the rules how to interpreted these graphs are explained.

- The Signals flows from left to right of this graph and is also denoted by an arrow in the line.
- The dashed lines in the graph denotes that the data has to be inverted.
- A number noteded above a line indicates that the data should be multiplied by this number.
- The circles are addition of the the lines flowing in and the sum is the line flowing out. See figure A.1 for some examples.

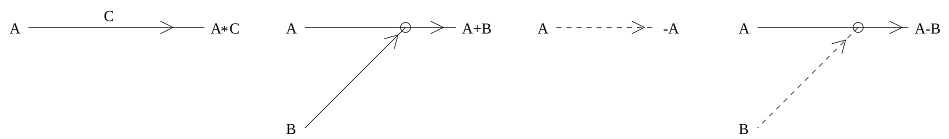

Figure A.1: Examples of Flow graphs.

Proofs of the DCT optimizing of Hou. B

In this appendix same exampes of the reordered DCT are given The input and output sequence of the DCT are reordered to get a regular matrix. If you divide the matrices into four quadrants you can see that upper two quadrants are equal and that the lower two quadrants are opposite in sign.

$$\begin{bmatrix} z_0 \\ z_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ \alpha & -\alpha \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

$$\begin{bmatrix} z_0 \\ z_2 \\ z_1 \\ z_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ \alpha & -\alpha & \alpha & -\alpha \\ \beta & -\delta & -\beta & \delta \\ \delta & \beta & -\delta & -\beta \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_2 \\ x_1 \\ x_3 \end{bmatrix}$$

$$\begin{bmatrix} z_0 \\ z_4 \\ z_2 \\ z_6 \\ z_1 \\ z_5 \\ z_3 \\ z_7 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \alpha & -\alpha & \alpha & -\alpha & \alpha & -\alpha & \alpha & -\alpha \\ \beta & -\delta & -\beta & \delta & \beta & -\delta & -\beta & \delta \\ \delta & \beta & -\delta & -\beta & \delta & \beta & -\delta & -\beta \\ \lambda & \mu & -\upsilon & -\gamma & -\lambda & -\mu & \upsilon & \gamma \\ \mu & \upsilon & -\gamma & \lambda & -\mu & -\upsilon & \gamma & -\lambda \\ \gamma & -\lambda & \mu & \upsilon & -\gamma & \lambda & -\mu & -\upsilon \\ \upsilon & \gamma & \lambda & \mu & -\upsilon & -\gamma & -\lambda & -\mu \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_2 \\ x_4 \\ x_6 \\ x_7 \\ x_5 \\ x_3 \\ x_1 \end{bmatrix}$$

were the constants are defined in table B.1 From these three examples in you can see

α	$\cos(\pi/4)$
β	$\cos(\pi/8)$
δ	$\sin(\pi/4)$
λ	$\cos(\pi/16)$
γ	$\cos(3\pi/16)$
μ	$\sin(3\pi/16)$
υ	$\sin(\pi/16)$

Table B.1: Constants of the DCT core matrix.

that the following property for the DCT matrix $\hat{C}(N)$ can be derived:

$$\hat{C} = \begin{bmatrix} \hat{C}(\frac{N}{2}) & \hat{C}(\frac{N}{2}) \\ \hat{D}(\frac{N}{2}) & -\hat{D}(\frac{N}{2}) \end{bmatrix} \tag{B.1}$$

The proof of equation B.1 is given in [6]

Two dimensional DCT algorithm code

C

In this appendix the c code of the optimized one dimensional DCT algorithm is given. The operations *#define adds(a,b) subs(a,b) muls(a,b) sp(r,w) rd(a) wr(a)* are for a normal c compiler replaced by the defenition given after the operation. With the MOVE compiler however these operation are performed by special function units as one instruction.

```
#include <stdio.h>
#include        "config.h"
#include        "global.h"

/*******************************************************
*                                                      *
*  This procedure computes the two dimensional DCT     *
*  with the optimized algorithm of Cho                 *
*  For the one dimensional core DCT the optimized      *
*  algorithm of hou is used                            *
*  The input is the extern parameter MB[0][]           *
*  The output s the extern parameter MB[1][]           *
*                                                      *
*******************************************************/

extern   short   MB[2][64];
         int     ReadPort,WritePort;


#define  d0     32138
```

```
#define  d1     18205
#define  d2    -6393
#define  d3    -27246
#define  b0     30274
#define  b1    -12540
#define  a0     23170
#define  PI     3.14159265358979323846


/*******************************************************
*                                                      *
*  The houm() macro is not defined as a function.      *
*  The reasen is to produce MOVE code which is         *
*  executes faster if it is scheduled                  *
*                                                      *
*******************************************************/

#define houm()                                  \
                                                \
h0=adds(p0,p7);    h4=muls(subs(p0,p7),d0);    \
h1=adds(p2,p5);    h5=muls(subs(p2,p5),d1);    \
h2=adds(p4,p3);    h6=muls(subs(p4,p3),d2);    \
h3=adds(p6,p1);    h7=muls(subs(p6,p1),d3);    \
                                                \
p0=adds(h0,h2);    p2=muls(subs(h0,h2),b0);    \
p1=adds(h1,h3);    p3=muls(subs(h1,h3),b1);    \
p4=adds(h4,h6);    p6=muls(subs(h4,h6),b0);    \
p5=adds(h5,h7);    p7=muls(subs(h5,h7),b1);    \
                                                \
h0=adds(p0,p1);    h1=muls(subs(p0,p1),a0);    \
h2=adds(p2,p3);    h3=muls(subs(p2,p3),a0)<<1; \
h4=adds(p4,p5);    h5=muls(subs(p4,p5),a0)<<1; \
h6=adds(p6,p7);    h7=muls(subs(p6,p7),a0)<<1; \


/*******************************************************
*                                                      *
*  The following macros are used to simulate the       *
*  special function units of the proposed implementation  *
*  of the Fast DCT within the MOVE framework.          *
*                                                      *
*******************************************************/

#define adds(a,b)   (((a)+(b))>>1)
#define subs(a,b)   (((a)-(b))>>1)
#define muls(a,b)   (((a)*(b)+16384)>>15)

#define sp(r,w)     ReadPort=r; WritePort=w
#define rd(a)       (Mblock[ReadPort][a])
#define wr(a)       (Mblock[WritePort][a])
```

```
void fdct2()


{short    x00,x01,x02,x03,x04,x05,x06,x07,
          x10,x20,x30,x40,x50,x60,x70,x44,
          l0,l1,l2,l3,l4,l5,l6,l7,l,r,k,
          r0,r1,r2,r3,r4,r5,r6,r7;

short   p0,p1,p2,p3,p4,p5,p6,p7,
        h0,h1,h2,h3,h4,h5,h6,h7;


sp(0,1);

p0= adds(l0=rd(0), r0=rd(7) ) ;
p1= adds(l1=rd(9), r1=rd(14)) ;
p2= adds(l2=rd(18),r2=rd(21)) ;
p3= adds(l3=rd(27),r3=rd(28)) ;
p4= adds(l4=rd(36),r4=rd(35)) ;
p5= adds(l5=rd(45),r5=rd(42)) ;
p6= adds(l6=rd(54),r6=rd(49)) ;
p7= adds(l7=rd(63),r7=rd(56)) ;

houm()

wr(0,h0);
wr(4,h1);
wr(2,h2);
wr(6,h3-h2);
wr(1,h4);
p7=(h7-h6)<<1;
p3=h6*2-h4;
wr(3,p3);
p5=(h5-p3);
wr(5,p5);
wr(7,(p7-p5));

p0= subs(l0,r0) ;
p1= subs(l1,r1) ;
p2= subs(l2,r2) ;
p3= subs(l3,r3) ;
p4= subs(l4,r4) ;
p5= subs(l5,r5) ;
p6= subs(l6,r6) ;
p7= subs(l7,r7) ;

houm()

wr(32+0,h0);
wr(32+4,h1);
wr(32+2,h2);
```

```
wr(32+6,h3-h2);
wr(32+1,h4);
p7=(h7-h6)<<1;
p3=h6*2-h4;
wr(32+3,p3);
p5=(h5-p3);
wr(32+5,p5);
wr(32+7,(p7-p5));




p0= adds(l0=rd(1),r0=rd(6) ) ;
p1= adds(l1=rd(12),r1=rd(11)) ;
p2= adds(l2=rd(23),r2=rd(16)) ;
p3= adds(l3=rd(26),r3=rd(29)) ;
p4= adds(l4=rd(37),r4=rd(34)) ;
p5= adds(l5=rd(40),r5=rd(47)) ;
p6= adds(l6=rd(51),r6=rd(52)) ;
p7= adds(l7=rd(62),r7=rd(57)) ;

houm()

wr(8+0,h0);
wr(8+4,h1);
wr(8+2,h2);
wr(8+6,h3-h2);
wr(8+1,h4);
p7=(h7-h6)<<1;
p3=h6*2-h4;
wr(8+3,p3);
p5=(h5-p3);
wr(8+5,p5);
wr(8+7,(p7-p5));

p0= subs(l0,r0) ;
p1= subs(l1,r1) ;
p2= subs(l2,r2) ;
p3= subs(r3,l3) ;
p4= subs(r4,l4) ;
p5= subs(l5,r5) ;
p6= subs(l6,r6) ;
p7= subs(l7,r7) ;

houm()

wr(40+0,h0);
wr(40+4,h1);
wr(40+2,h2);
wr(40+6,h3-h2);
```

```
wr(40+1,h4);
p7=(h7-h6)<<1;
p3=h6*2-h4;
wr(40+3,p3);
p5=(h5-p3);
wr(40+5,p5);
wr(40+7,(p7-p5));
```

```
p0= adds(l0=rd(2), r0=rd(5) ) ;
p1= adds(l1=rd(15),r1=rd(8) ) ;
p2= adds(l2=rd(20),r2=rd(19)) ;
p3= adds(l3=rd(25),r3=rd(30)) ;
p4= adds(l4=rd(38),r4=rd(33)) ;
p5= adds(l5=rd(43),r5=rd(44)) ;
p6= adds(l6=rd(48),r6=rd(55)) ;
p7= adds(l7=rd(61),r7=rd(58)) ;

houm()

wr(16+0,h0);
wr(16+4,h1);
wr(16+2,h2);
wr(16+6,h3-h2);
wr(16+1,h4);
p7=(h7-h6)<<1;
p3=h6*2-h4;
wr(16+3,p3);
p5=(h5-p3);
wr(16+5,p5);
wr(16+7,(p7-p5));

p0= subs(l0,r0) ;
p1= subs(l1,r1) ;
p2= subs(r2,l2) ;
p3= subs(l3,r3) ;
p4= subs(l4,r4) ;
p5= subs(r5,l5) ;
p6= subs(l6,r6) ;
p7= subs(l7,r7) ;

houm()

wr(48+0,h0);
wr(48+4,h1);
wr(48+2,h2);
wr(48+6,h3-h2);
```

```
wr(48+1,h4);
p7=(h7-h6)<<1;
p3=h6*2-h4;
wr(48+3,p3);
p5=(h5-p3);
wr(48+5,p5);
wr(48+7,(p7-p5));



/*************************************************************************/


p0= adds(l0=rd(3),r0=rd(4)) ;
p1= adds(l1=rd(10),r1=rd(13)) ;
p2= adds(l2=rd(17),r2=rd(22)) ;
p3= adds(l3=rd(24),r3=rd(31)) ;
p4= adds(l4=rd(39),r4=rd(32)) ;
p5= adds(l5=rd(46),r5=rd(41)) ;
p6= adds(l6=rd(53),r6=rd(50)) ;
p7= adds(l7=rd(60),r7=rd(59)) ;

houm()

wr(24+0,h0);
wr(24+4,h1);
wr(24+2,h2);
wr(24+6,h3-h2);
wr(24+1,h4);
p7=(h7-h6)<<1;
p3=h6*2-h4;
wr(24+3,p3);
p5=(h5-p3);
wr(24+5,p5);
wr(24+7,(p7-p5));

p0= subs(l0,r0) ;
p1= subs(r1,l1) ;
p2= subs(l2,r2) ;
p3= subs(r3,l3) ;
p4= subs(r4,l4) ;
p5= subs(l5,r5) ;
p6= subs(r6,l6) ;
p7= subs(l7,r7) ;

houm()

wr(56+0,h0);
wr(56+4,h1);
wr(56+2,h2);
wr(56+6,h3-h2);
```

```
wr(56+1,h4);
p7=(h7-h6)<<1;
p3=h6*2-h4;
wr(56+3,p3);
p5=(h5-p3);
wr(56+5,p5);
wr(56+7,(p7-p5));



/*************************************************************************/
/*************************************************************************/
/*************************************************************************/



sp(1,0);

wr(0, adds(l0=rd(0),r0=rd(24)) );
wr(1, adds(l1=rd(1),r1=rd(25)) );
wr(2, adds(l2=rd(2),r2=rd(26)) );
wr(3, adds(l3=rd(3),r3=rd(27)) );
wr(4, adds(l4=rd(4),r4=rd(28)) );
wr(5, adds(l5=rd(5),r5=rd(29)) );
wr(6, adds(l6=rd(6),r6=rd(30)) );
wr(7, adds(l7=rd(7),r7=rd(31)) );

wr(24, subs(l0,r0) );
wr(25, subs(l1,r1) );
wr(26, subs(l2,r2) );
wr(27, subs(l3,r3) );
wr(28, subs(l4,r4) );
wr(29, subs(l5,r5) );
wr(30, subs(l6,r6) );
wr(31, subs(l7,r7) );

wr(8,  adds(l0=rd(8), r0=rd(16)) );
wr(9,  adds(l1=rd(9), r1=rd(17)) );
wr(10, adds(l2=rd(10),r2=rd(18)) );
wr(11, adds(l3=rd(11),r3=rd(19)) );
wr(12, adds(l4=rd(12),r4=rd(20)) );
wr(13, adds(l5=rd(13),r5=rd(21)) );
wr(14, adds(l6=rd(14),r6=rd(22)) );
wr(15, adds(l7=rd(15),r7=rd(23)) );

wr(16, subs(l0,r0) );
wr(17, subs(l1,r1) );
wr(18, subs(l2,r2) );
wr(19, subs(l3,r3) );
wr(20, subs(l4,r4) );
wr(21, subs(l5,r5) );
```

```
wr(22, subs(l6,r6) );
wr(23, subs(l7,r7) );


wr(32, adds(rd(32),0) );
wr(33, adds(l1=rd(33),r1=rd(63)) );
wr(34, adds(l2=rd(34),r2=rd(62)) );
wr(35, adds(l3=rd(35),r3=rd(61)) );
wr(36, adds(l4=rd(36),r4=rd(60)) );
wr(37, adds(l5=rd(37),r5=rd(59)) );
wr(38, adds(l6=rd(38),r6=rd(58)) );
wr(39, adds(l7=rd(39),r7=rd(57)) );

wr(40, subs(rd(56),0) );
wr(47, subs(l1,r1) );
wr(46, subs(l2,r2) );
wr(45, subs(l3,r3) );
wr(44, subs(l4,r4) );
wr(43, subs(l5,r5) );
wr(42, subs(l6,r6) );
wr(41, subs(l7,r7) );

wr(48, adds(rd(40),0) );
wr(49, adds(l1=rd(41),r1=rd(55)) );
wr(50, adds(l2=rd(42),r2=rd(54)) );
wr(51, adds(l3=rd(43),r3=rd(53)) );
wr(52, adds(l4=rd(44),r4=rd(52)) );
wr(53, adds(l5=rd(45),r5=rd(51)) );
wr(54, adds(l6=rd(46),r6=rd(50)) );
wr(55, adds(l7=rd(47),r7=rd(49)) );

wr(56, subs(rd(48),0) );
wr(63, subs(l1,r1) );
wr(62, subs(l2,r2) );
wr(61, subs(l3,r3) );
wr(60, subs(l4,r4) );
wr(59, subs(l5,r5) );
wr(58, subs(l6,r6) );
wr(57, subs(l7,r7) );



/************************************************************************/
/************************************************************************/
/************************************************************************/


sp(0,1);

x00=adds(l0=rd(0),r0=rd(8));
if (x00>0) x00=(x00+8) >> 4;
```

```
else       x00=(x00+7) >> 4;

x44=subs(l0,r0);
if (x44>0) x44=(x44+8) >> 4;
else       x44=(x44+7) >> 4;


x40=(muls(adds(l0=rd(4),r0=rd(12)),a0)+4) >> 3;
x04= muls(subs(l0,r0)+5,a0)  >> 3;




x10=(muls(adds(l0=rd(1),r0=rd(9)),a0)+4) >> 3;
x20=(muls(adds(l1=rd(2),r1=rd(10)),a0)+4) >> 3;
x30=(muls(adds(l2=rd(3),r2=rd(11)),a0)+4) >> 3;
x50=(muls(adds(l3=rd(5),r3=rd(13)),a0)+4) >> 3;
x60=(muls(adds(l4=rd(6),r4=rd(14)),a0)+4) >> 3;
x70= muls(adds(l5=rd(7),r5=rd(15))+6,a0)  >> 3;

wr(9,  subs(l0,r0));
wr(10, subs(l1,r1));
wr(11, subs(l2,r2));
wr(13, subs(l3,r3));
wr(14, subs(l4,r4));
wr(15, subs(l5,r5));




x06= muls(subs(l0=rd(30),r0=rd(18))+5,a0)  >> 3;
x02=(muls(adds(l1=rd(26),r1=rd(22)),a0)+4) >> 3;
x01=(muls(adds(l2=rd(33),r2=rd(51)),a0)+4) >> 3;
x05= muls(subs(l3=rd(37),r3=rd(63))+5,a0)  >> 3;
x07= muls(subs(l4=rd(41),r4=rd(59))+5,a0)  >> 3;
x03= muls(subs(l5=rd(45),r5=rd(55))+5,a0)  >> 3;

wr(18, adds(l0,r0));
wr(22, subs(l1,r1));
wr(51, subs(l2,r2));
wr(37, adds(l3,r3));
wr(41, adds(l4,r4));
wr(45, adds(l5,r5));





wr(16,      rd(16)>>1);
wr(17, adds(l0=rd(17),r0=rd(31)));
```

```
wr(19, adds(l1=rd(19),r1=rd(29)));
wr(28, adds(l2=rd(28),r2=rd(20)));
wr(27, adds(l3=rd(27),r3=rd(21)));
wr(25, adds(l4=rd(25),r4=rd(23)));

wr(24,      rd(24)>>1);
wr(31, subs(l0,r0));
wr(29, subs(l1,r1));
wr(20, subs(l2,r2));
wr(21, subs(l3,r3));
wr(23, subs(l4,r4));

wr(32, adds(l0=rd(32),r0=rd(52)));
wr(34, adds(l1=rd(34),r1=rd(50)));
wr(35, adds(l2=rd(35),r2=rd(49)));
wr(36, adds(l3=rd(36),r3=rd(48)));
wr(38, adds(l4=rd(38),r4=rd(62)));
wr(39, adds(l5=rd(39),r5=rd(61)));

wr(52, subs(l0,r0));
wr(50, subs(l1,r1));
wr(49, subs(l2,r2));
wr(48, subs(l3,r3));
wr(62, subs(l4,r4));
wr(61, subs(l5,r5));



wr(40, adds(l0=rd(40),r0=rd(60)));
wr(42, adds(l1=rd(42),r1=rd(58)));
wr(43, adds(l2=rd(43),r2=rd(57)));
wr(44, adds(l3=rd(44),r3=rd(56)));
wr(46, adds(l4=rd(46),r4=rd(54)));
wr(47, adds(l5=rd(47),r5=rd(53)));


wr(60, subs(l0,r0));
wr(58, subs(l1,r1));
wr(57, subs(l2,r2));
wr(56, subs(l3,r3));
wr(54, subs(l4,r4));
wr(53, subs(l5,r5));




/***************************************************************************
*                                                                         *
*                                                                         *
*                     berekend tot na 1-DCT transformatie                 *
```

```
*                                                                          *
*                                                                          *
****************************************************************************/

sp(1,0);


wr(0, (x00));
wr(1, (x01));
wr(2, (x02));
wr(3, (x03));
wr(4, (x04));
wr(5, (x05));
wr(6, (x06));
wr(7, (x07));
wr(8, (x10));
wr(16, (x20));
wr(24, (x30));
wr(32, (x40));
wr(40, (x50));
wr(48, (x60));
wr(56, (x70));
wr(36, (x44));

l=rd(9); r=rd(15);

wr(44, ((l-r+7) >> 4));
wr(28, ((r+l+8) >> 4));
wr(52, ((rd(10)-rd(14)+8) >> 4));
wr(20, ((rd(14)+rd(10)+8) >> 4));
wr(60, ((rd(11)-rd(13)+7) >> 4));
wr(12, ((rd(13)+rd(11)+8) >> 4));

wr(34, ((rd(18)+rd(22)+8) >> 4));
wr(38, ((rd(22)-rd(18)+7) >> 4));
wr(22, ((rd(20)-rd(16)+8) >> 4));
wr(50, ((rd(16)+rd(20)+8) >> 4));
wr(30, ((rd(21)-rd(17)+7) >> 4));
wr(42, ((rd(17)+rd(21)+8) >> 4));
wr(46, ((rd(23)-rd(19)+7) >> 4));
wr(26, ((rd(19)+rd(23)+8) >> 4));

wr(18, ((rd(24)+rd(28)+8) >> 4));
wr(54, ((rd(24)-rd(28)+7) >> 4));
wr(10, ((rd(25)+rd(27)+8) >> 4));
wr(62, ((rd(25)-rd(27)+8) >> 4));
wr(14, ((-rd(31)-rd(29)+7) >> 4));
wr(58, ((rd(31)-rd(29)+8) >> 4));

wr(9,  ((rd(32)+rd(34)+9) >> 4));
```

```
wr(63, ((rd(32)-rd(34)+7) >> 4));
wr(41, ((rd(44)+rd(38)+8) >> 4));
wr(31, ((rd(44)-rd(38)+7) >> 4));
wr(33, ((rd(45)+rd(37)+9) >> 4));
wr(39, ((rd(45)-rd(37)+8) >> 4));
wr(25, ((rd(46)+rd(36)+8) >> 4));
wr(47, ((rd(46)-rd(36)+8) >> 4));
wr(17, ((rd(47)+rd(35)+9) >> 4));
wr(55, ((rd(47)-rd(35)+7) >> 4));

wr(13, ((rd(48)+rd(62)+8) >> 4));
wr(59, ((rd(48)-rd(62)+8) >> 4));
wr(21, ((rd(49)+rd(61)+8) >> 4));
wr(51, ((rd(49)-rd(61)+7) >> 4));
wr(29, ((rd(50)+rd(60)+8) >> 4));
wr(43, ((rd(50)-rd(60)+8) >> 4));
wr(35, ((rd(51)+rd(41)+8) >> 4));
wr(37, ((rd(51)-rd(41)+7) >> 4));
wr(27, ((rd(52)+rd(42)+8) >> 4));
wr(45, ((rd(52)-rd(42)+7) >> 4));
wr(19, ((rd(53)+rd(43)+8) >> 4));
wr(53, ((rd(53)-rd(43)+7) >> 4));
wr(11, ((rd(54)+rd(56)+8) >> 4));
wr(61, ((rd(54)-rd(56)+7) >> 4));

wr(49, ((rd(57)+rd(39)+8) >> 4));
wr(23, ((rd(57)-rd(39)+7) >> 4));
wr(57, ((rd(58)+rd(40)+8) >> 4));
wr(15, ((rd(58)-rd(40)+7) >> 4));


}
```

D Example of a machine description file

In this chpater an example of a machine description (mach) file is given. This mach file describes the TTA of figure 6.2.

```
//
//      Project     : A software pipelining scheduler for TTAs
//      Author      : Jan Hoogerbrugge (janh@duteca.et.tudelft.nl)
//      Description : An example machine description file
//      Copyright   : 1992 Jan Hoogerbrugge
//

#include "mach.def"

MoveBusses
{
       m1       64, 8, signed;
       m2       64, 8, signed;
       m3       64, 8, signed;
       m4       64, 8, signed;
       m5       64, 8, signed;
       m6       64, 8, signed;
       m7       64, 8, signed;
       m8       64, 8, signed;
       m9       64, 8, signed;
}

Sockets
{
```

```
        fu5_o           input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        fu5_t           input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        fu5_r           output, {m1, m2, m3, m4, m5, m6, m7, m8, m9};

        fu6_o           input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        fu6_t           input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        fu6_r           output, {m1, m2, m3, m4, m5, m6, m7, m8, m9};

        fu8_o           input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        fu8_t           input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        fu8_r           output, {m1, m2, m3, m4, m5, m6, m7, m8, m9};

        r_i1            input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        r_i2            input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        r_i3            input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        r_o1            output, {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        r_o2            output, {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        r_o3            output, {m1, m2, m3, m4, m5, m6, m7, m8, m9};

        f_i1            input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        f_i2            input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        f_i3            input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        f_o1            output, {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        f_o2            output, {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        f_o3            output, {m1, m2, m3, m4, m5, m6, m7, m8, m9};

        b_i1            input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        b_i2            input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};

        pc              input,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        ra              bidir,  {m1, m2, m3, m4, m5, m6, m7, m8, m9};

        ir_1            output, {m1, m2, m3, m4, m5, m6, m7, m8, m9};
        ir_2            output, {m1, m2, m3, m4, m5, m6, m7, m8, m9};
}

FunctionUnits
{
        ALU             hybrid, 1          , fu5_o, none, fu5_t, fu5_r, ALU_2;
        LS1             hybrid,1           , fu6_o, none, fu6_t, fu6_r, LSU;
        FP              hybrid, FPU_LATENCY, fu8_o, none, fu8_t, fu8_r, FPU;
}

ImmediateUnits
{
        i1              32, signed, ir_1;
        i2              32, signed, ir_2;
}

RegisterUnits
{
```

```
        Integer         N_IREGS, {r_i1, r_i2, r_i3}, {r_o1, r_o2, r_o3};
        FloatingPoint   N_FREGS, {f_i1, f_i2, f_i3}, {f_o1, f_o2, f_o3};
        Boolean         N_BREGS, {b_i1, b_i2};
}

InstructionUnit
{
        JumpLatency     JUMP_LATENCY;
        BoolLatency     BOOL_LATENCY;
        BoolExprSize    BOOL_EXPR_SIZE;
        ProgramCounter  pc;
        ReturnAddress   ra, ra;
}
```