Feasibility Analysis of MPEG decoding on reconfigurable hardware

Master's Thesis

by

Albert Molderink

Committee:
Prof. dr. ir. G.J.M. Smit
Ir. G.K. Rauwerda (Recore Systems)
Ir. P.T. Wolkotte
Dr. ir. L.T. Smit (Recore Systems)

University of Twente, Enschede, The Netherlands
29th March 2007

Abstract

This thesis describes the advantages and disadvantages of using an architecture consisting of different (types of) processors. These architectures are called heterogeneous architectures. We focus on advantages and disadvantages in processing power and energy-efficiency. We examined these by mapping MPEG-4 on a heterogeneous architecture consisting of an General Purpose Processor (GPP) and a reconfigurable processor architecture. The used GPP is an ARM and the reconfigurable architecture a Montium. We analysed the MPEG-4 algorithms and the characteristics of multi-processor architectures to decide on what processor type the algorithms can be executed best.

The analysis shows that two of the four parts (processes) of MPEG-4 can best be executed on the Montium: the inverse quantization (iQT) and the color space conversion (CS). The other two parts, variable length coding (VLC) and Motion Compensation (MC), can not be implemented on the Montium efficiently and are therefore implemented on the ARM. It is possible to implement the MC on a FPGA.

Our MPEG-4 implementation is based on the GPP implementation of the Smart Chips for Smart Surrounding (4S) project. The used hardware is the Basic Concept Verification Platform (BCVP) of the 4S project. The BCVP consists, next to other not used hardware, of two ARMs (an ARM946 and an ARM920) and three Montiums. We only used the ARM946. The used Operating Systems (OS) on the ARM is BasOS, an OS developed at the University of Twente. The 4S MPEG-4 implementation is adapted to run on BasOS. The mapping of the algorithms on the Montium is described in this thesis.

To implement an application consisting of multiple processes, communication between processes is required. These processes can run on the same processor or on different processors. Furthermore, for an energy-efficient heterogeneous architecture it should be possible to remap processes to other processors on run-time. Therefore, a communication system is developed that supports communication between processes on the BCVP and run-time remapping. For easy usage, a Controller is developed that initiates the communication between processes and processors and configures the Montiums.

Comparison of the MPEG-4 implementation with the original 4S implementation

shows that the communication and operating system overhead is smaller for BASOS. Executing MPEG-4 using the MONTIUMS increases the performance in both processing power and energy-efficiency. To investigate the performance of the MONTIUM, the energy-efficiency and processing power of the MONTIUM executing the inverse Discrete Cosine Transform (iDCT) algorithm of MPEG-4 are presented. These results are compared with the iDCT implementation on an ARM, TI DSP and an ASIC.

Contents

List of Figures

Acknowledgements

This thesis concludes my Master study Computer Science, track embedded systems, on the University Twente. I performed this thesis on the Computer Architecture Design & Test for Embedded Systems (CADTES) chair. This thesis is carried out for the FP6 Smart ChipS for Smart Surroundings (4S) project(IST-001908) supported by the European Commission.

Gerard Smit offered this assignment with a lot of enthusiasm. During the thesis he provided constant enthusiasm and ideas.

I also want to thank Marcel van de Burgwal, Gerard Rauwerda and Pascal Wolkotte for their insights in how to map the iDCT on a Montium. This was very difficult for me to do because I did not have very much experience with the concept of mapping algorithms and registers on parallel processors manually.

Furthermore, I learned a lot about streaming processing and LaTeX by the discussions with Pascal Wolkotte and Philip Hölzenspies.

I want to thank Lodewijk Smit for his introduction into the energy consumption, performance analysis and writing a scientific paper.

My graduation committee provided me with guidance, reviews and understanding for my difficult private situation, for which I am grateful.

I would not have been able to finish this thesis without the support of my friends and family. Especially, I want to thank my parents for the support during my study. The last tough year my mother supported me, despite her own problems, even more, for which I am very grateful.

Enschede, April 2007
Albert Molderink

Introduction

Fifty years ago a mobile phone was something from the future. Today, more mobile phones are sold every year than there live people in the Netherlands. Next to mobile phones much more mobile technology is available like PDAs, MP3 players and multimedia players. All these devices get more functionality that require more energy, while the battery has to last for at least a week. This requires more energy-efficient processors.

We see a continuous increase in the provided processing power of mobile technology. It is possible to play games and listen to music on your mobile phone. Nowadays, this processing power is mainly provided by General Purpose Processors (GPP) and Application-Specific Integrated Circuits(ASICs), but more and more reconfigurable processors become available for mobile technology. These two types of processors are sometimes combined in one architecture. Such an architecture consisting of different types of architectures is called a heterogeneous architecture. An architecture consisting of a GPP and a reconfigurable processor may be the solution for both the performance requirement and the energy-efficiency problem.

Many applications on mobile devices have common characteristics. Almost all of them are streaming applications like Moving Picture Expert Group (MPEG), Digital Radio Mondiale (DRM) and Universal Mobile Telecommunication System (UMTS). By analysing one of the algorithms in detail and study the different characteristics (like processing time, memory usage and throughput) between the applications one can make an estimation of analogous applications.

We see a trend that the MPEG-4 standard requires lower bit rates than previous MPEG standards and wireless connections get more and more bandwidth, so it is possible to send a MPEG-4 stream over a state-of-the-art licensed spectrum wireless connection (i.e. a mobile phone network). However, the decreasing bit rates of MPEG-4 are caused by higher compression ratios because the user's experience may not drop. These higher compression ratios require more processing power for decoding. With the increasing processing power of mobile devices it is possible to watch video on these devices.

In this master thesis it is analysed what the characteristics of the different parts of the video decoding of the MPEG-4 standard are. Next is examined if it is possible

to perform these tasks on reconfigurable hardware and which advantages and disadvantages this gives in terms of processing time and energy-efficiency. The feasibility study investigates the required memory, processing power, communication bandwidth and communication overhead. The required memory and processing power are compared with the available resources of the reconfigurable hardware. In this way we validate whether the tasks can be done in real time. The required bandwidth is compared with the bandwidth of the communication link between the GPP and the reconfigurable hardware and the interfaces of both architectures. Finally, the communication overhead for the GPP is compared with the reduction of required processing time of the GPP caused by outsourcing the tasks to the reconfigurable hardware. After the analysis the results are verified by building a prototype. Therefore, it is necessary to implement MPEG-4 on a heterogeneous architecture that consists of a GPP and reconfigurable processing tiles. This implementation is realized on real hardware to demonstrate the results of the project. The hardware is the Basic Concept Verification Platform (BCVP) of the 4S project [1] consisting of 2 ARMs connected with 3 Montiums (emulated in a FPGA) through a bus and a Network-on-Chip (NoC). A Montium is a word-level reconfigurable processor architecture. The MPEG-4 version that is implemented is the Simple profile of MPEG-4, QCIF format.

Next to the MPEG-4 implementation there is also a Controller implemented. This Controller initiates the communication between the processes on the ARM, the reconfigurable hardware and the PC (over an USB connection). The Controller also configures the reconfigurable hardware and starts the application. Before the Controller is implemented, it is analysed what functionality the controller requires, what the requirements are and how this can be implemented.

The first chapter of this report gives an description of the MPEG-4 decoding and the required algorithms. The second chapter describes the different available architectures and for which type of algorithms they are suitable. The third chapter analyses which parts of MPEG-4 are the most computational intensive and which parts can be implemented efficiently on the different processors of the BCVP. Chapter four gives a description of the implementation of the Controller. In the fifth chapter the test results are presented. We end up with a conclusion in chapter six.

CHAPTER 1

MPEG overview

MPEG is a frequently used compression technique for audio and video. In 1988 the Moving Picture Expert Group (MPEG) was established as a cooperation of academics and business people. MPEG is a working group of the International Organization for Standardization (ISO) [2]. The official title for the group is: Coding of moving pictures and audio. Their assignment is developing international standards for compression, decompression, processing and coding representation of moving pictures, audio and their combination, in order to satisfy a wide variety of applications [2].

Video consists of a stream of pictures. Each picture slightly differs from the previous one and by showing many pictures in a short time (about 15 to 30 per second) it appears like a moving scene. Such a picture in a video stream is called a *frame*.

Compression of video is necessary to reduce the bandwidth of the video stream. For mobile applications a resolution of 352 x 288 pixels is often used. Video usually consists of 25 frames per second. When 8 bits (one byte) per pixel are used for each color (red, green and blue), it would require 3 x 352 x 288 x 25 $\approx$ 7 MB per second. This requires a very high bandwidth to stream video without any form of compression, especially for a mobile connection. Beside the bandwidth limitation, the data traffic is expensive for mobile connections, so it would also be expensive without compression. MPEG-4 compresses the date with a ratio between 1:50 and 1:100.

This chapter gives a description of MPEG-4. In the first section a number of compression techniques and principles are pointed out and standard techniques used for digital video are explained. The next section gives a short introduction into the different MPEG versions. Section three provides an overview and a description of the different steps in the encoding and why these steps are included.

1.1 Compression principles

MPEG is a lossy way of data compression. The decompressed data differs from the

original data; information is lost, so the quality decreases. The idea of this way of compression is that only the information of the (for human eyes) less-visible part of video is lost (and for audio the less-hearable part).

MPEG only standardizes the format of the bitstream and the decoder. So the encoder is not standardized at all, only the format of the output it has to generate and how the original data has to be retrieved. This means that an arbitrary decoder can decode all MPEG streams, independent of the encoder [3].

Since a video stream consists of frames, video compression is in fact compression of a stream of pictures. The compression is based on two principles: spatial redundancy and temporal redundancy. In a picture, pixels next to each other do not differ much. When you divide an image in little blocks of pixels (in MPEG 8x8), the values of the 64 pixels are almost the same. This is called *spatial redundancy* [4].

In a video stream, two successive frames are almost the same. This gives the impression of the movement in a scene. The direction of the movement is depicted by the *Motion Vector (MV)*. When you subtract two successive frames (the first original and the second moved along the inversed MV), most of the pixel values are zero, only a few pixels have a value, which is the difference. This is called *temporal redundancy* [4].

The compression is done in two ways: a small part of the frames are just compressed using spatial redundancy, these frames can be decoded without further information. These frames are called *Independent frames (I-frames)*. The other frames are compressed based on the differences with other frames. To decompress these frames, the frame on which the compression is based is needed. These frames are called *Predicted frame (P-frames)* and *Bidirectional frames (B-frames)*. P-frames are only based on a previous I- or P-frame, B-frames are based on a previous and successor I- or P-frame (frames are only based on I- or P-frames, not on B-frames). The compression ratio of the P and B frames is much higher than the compression ratio of I-frames.

A video stream is split in sequences of frames. Such a sequence is called a *Group Of Pictures (GOP)* and consists of one I-frame, multiple P-frames and an optional number of B-frames (see Figure 1.1). Because all the frames are dependent of each other, a video stream can only be started at the beginning of a GOP. The number of frames in a GOP is a trade off: more P- and B-frames in a GOP leads to a higher compression. But because all frames are dependent an error in one of the frames gives errors in all successive frames [4].

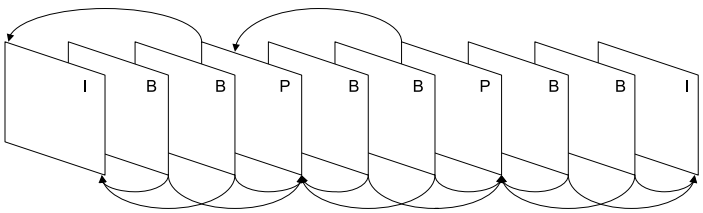


Figure 1.1: MPEG Group of Pictures

1.1.1 Conversion between RGB, YUV and YCbCr

Colors are in most digital systems represented by their RGB value. That means, the color is represented by its amount of Red, Green and Blue. In this way significant range of the colors of the visible spectrum can be depicted.

The first televisions were only able to show grayscale video. For showing grayscale video, only the amount of brightness is necessary. So, the television standard only implemented the (8-bit) brightness, called the luminance. When color televisions became available, another standard had to be defined.

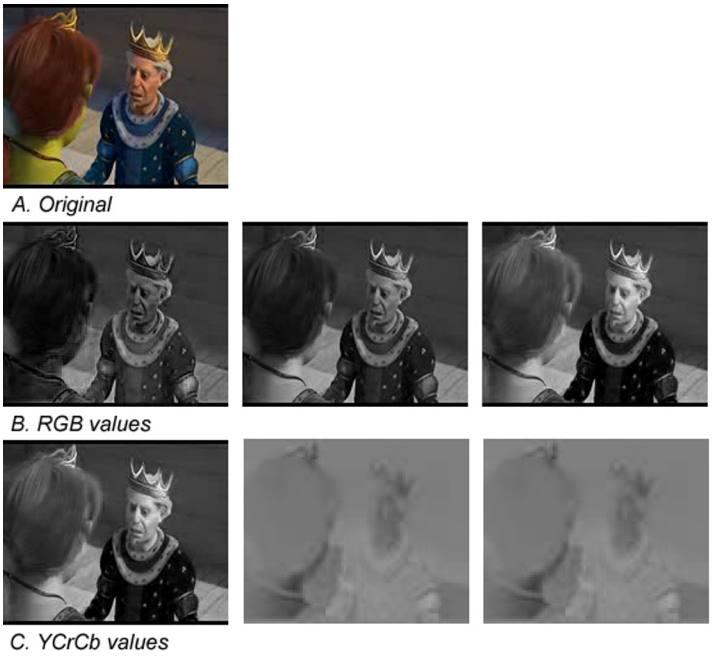


Figure 1.2: Original picture, RGB values and the luminance and chrominance values

The standard had to be able to represent colors and was supposed to be backward compatible: grayscale televisions must be able to read the standard. So, a RGB value was not the solution.

A YUV representation was developed. The Y factor is the luminance and the U and V are the hue (frequency) and saturation (amount of black) in the color. The luminance factor (Y) is equal to the luminance factor in the old standard and is calculated as the weighted sum of the RGB values (depending on the light intensity of the color for the human eye). Old grayscale televisions only use this factor to show grayscale video.

The U and V factors are also calculated as the weighted sum of the RGB values. The factors are encoded into the video signal using a subcarrier. In this way the grayscale televisions are not aware of these signals.

For MPEG a slightly different way of pixel representation is used, called YCrCb. This is often confused with the YUV standard. The Y is again the luminance factor,

the Cr and Cb are the chrominance factors (amount of blue and amount of red in the color). The range of the R, G and B samples is 0 to 255, the range of the Y sample 16 to 235 and the range of the Cr and Cb samples 16 to 240. The formulas to calculate the YCbCr values from RGB and vice-versa are the following:

$$\begin{aligned} Y &= 0.257 \cdot R + 0.504 \cdot G + 0.098 \cdot B + 16 \\ Cr &= 0.439 \cdot R - 0.368 \cdot G - 0.071 \cdot B + 128 \\ Cb &= -0.148 \cdot R - 0.291 \cdot G + 0.439 \cdot B + 128 \end{aligned}$$

$$\begin{aligned} R &= 1.164 \cdot (Y - 16) + 1.596 \cdot (Cr - 128) \\ G &= 1.164 \cdot (Y - 16) - 0.813 \cdot (Cr - 128) - 0.392 \cdot (Cb - 128) \\ B &= 1.164 \cdot (Y - 16) + 2.017 \cdot (Cb - 128) \end{aligned}$$

The first compression is performed by the conversion from RGB to YCbCr. The human eye is much more sensitive for luminance than for chrominance. This is shown in Figure 1.2. Figure 1.2A shows the original image, 1.2B the image divided in its R, G and B components and 1.2C the image divided in its Y, Cr and Cb components. Therefore MPEG uses not all chrominance values, only a quarter of the chrominance samples are kept (but all luminance samples!). So, for every four luminance values, there is one Cr and one Cb value. This is called the 4:2:0 video format. The format gives the relation between the number of luminance and chrominance values. The first number is the number of luminance samples, the second the number of chrominance samples in the odd lines of pixels and the third number gives the number of chrominance samples in the even lines (see Figure 1.3). So the 4:4:4 format is the format without any down sampling.

Figure 1.3 shows the exact location of the chrominance samples with respect to the luminance samples. Down sampling the chrominance values saves already half the bandwidth and storage capacity. The stored chrominance sample after down sampling is the average of the original chrominance pixels. In Figure 1.3 the chrominance sample is the average of the original four in a one box. When all chrominance samples are required (i.e. when the colorspace is converted to RGB) the stored sample is used for all pixels in one box. [5]

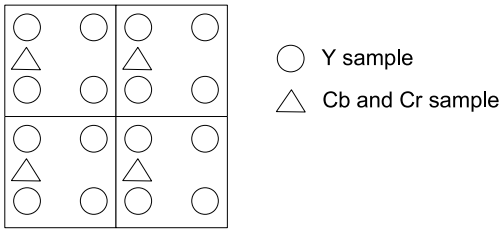


Figure 1.3: Position of the chrominance samples with respect to the luminance samples in 4:2:0 video format

1.1.2 Interlaced and Progressive

There are two ways of video transmission: *progressive* and *interlaced*. With a progressive transmission, each picture is entirely sent. It is scanned line by line (one horizontal line of pixels) and then compressed. Interlaced sends only half a picture. In the first frame the odd lines are send, in the second frame the even. Such half a picture is called a field. In this way the refresh rate is doubled without any extra bandwidth.

The afterglow of the phosphor of CRT tubes and the limitations of the human eye results in two fields being appeared as a complete image. This allows the viewing of full horizontal detail with half the bandwidth which would be required for a full progressive scan while maintaining the necessary CRT refresh rate to prevent flicker. Because of the afterglow of the phosphor is important for this principle, interlaced video does not work in other (new) technologies, e.g. TFT monitors. MPEG mostly uses a progressive transmission, but it supports also interlaced. [5]

1.2 MPEG versions

The information in this section is a summary from [2] [3].

The first goal for the workgroup was developing a standard for coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s. In practice this meant a standard for efficient storage and retrieval of audio and video on Compact Disc (CD). The standard consists of three *parts*: System, Video and Audio (the MPEG standards are divided in parts, every part describes a part of the standard, like video, audio and storage format). The multiplexing and synchronization of the elementary Audio and Video streams is provided by the System Part. The Video part provides an efficient encoding of non-interlaced pictures. The encoding of stereo audio with transparency is provided by the Audio part. The encoding can be done in three different bitrates. These three bitrates have a different level on encoding, such a level within a part is called a *Layer*.

These parts of the MPEG-1 standard where approved in 1992. A couple of years later two more parts where added to the standard: Conformance Testing and Software simulation. Conformance Testing provides methods and reference bit streams that can be used to asses conformance. Software simulation provides implementations of the System part and of the video and audio encoders and decoders.

The full MPEG-1 standard was used for Video CD decoding. Several hundreds million hardware Video CD (VCD) decoders have been sold worldwide. MPEG-1 Audio Layer III, better known as MP3, is a worldwide spread standard for storing audio.

In 1990 the Moving Picture Expert Group started to develop the second standard (MPEG-2) called "Generic coding of Moving Pictures and Associated Audio". The first three parts (System, Video and Audio) where approved in 1994. The system part provides two different versions, the Transport Stream version and the Program Stream version. The Transport Stream version provides support for efficient transmission over error-prone delivery systems. The Program Stream version is similar to the MPEG-1 version and is more useful for digital storage media. The improvement of the Video part is the efficient coding of interlaced pictures and different spatial

resolutions. The improvement of the Audio part is the support of encoding multi-channel audio.

MPEG-2 defines more parts than MPEG-1. There are several parts providing session and network protocols. Part 7 is the Advanced Audio Coding. This part provides multichannel audio coding and is not backward compatible with MPEG-1. In total there are 11 parts.

The System, Video and Audio parts (part 1, 2 and 3) of MPEG-2 are used for DVD decoding.

Developing the MPEG-4 standard started in 1993 and the first version was approved in 1998. The MPEG-4 standard is called "Coding of audio-visual objects".

In 1999 a major extension (called version 2) was approved. After these two versions more functionality was added that could be qualified as version. All the versions are backwards compatible. But, recognizing the different versions is not too important, it is more important to distinguish the different *profiles*. Because there is so much functionality available for MPEG, not all functionality is necessary for some implementation. But, if you do not implement everything then you do not follow the standard, so it is not a MPEG decoder anymore. Therefore, the functionality is divided into different boxes, called profiles. In the simplest profile only little functionality has to be implemented, in the most complex profile all functionality has to be implemented. The official definition of a profile is "a defined subset of the syntax of the specification". Examples of profiles within MPEG-4 and their application domain are: the Simple Profile for Internet, the Advanced Simple Profile for video on demand and the Studio profile for studio applications.

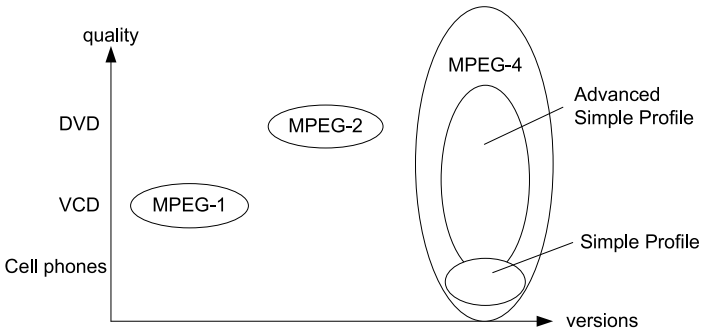


Figure 1.4: MPEG versions

MPEG-4 is a much broader standard than MPEG-1 and MPEG-2. MPEG-1 was only developed for video CD, MPEG-2 for DVD. Because the standard is divided in different profiles, there are profiles suitable for mobile connections and profiles suitable for studio quality video (see figure 1.4).

Within the profiles there are levels defined. The formal definition of a level is "a defined set of constraints on the values which may be taken by the parameters of the specification within a particular profile". An example of such a constraint is the bounds of the image size. A combination of a profile and a level is depicted like *Profile@Level*.

The name of the first 5 parts corresponds with MPEG-2, but there are a number of significant differences between MPEG-1/MPEG-2 and MPEG-4.

The first major difference is that MPEG-4 enables the coding of individual objects within frames. Each object of a video stream can be encoded individually, even if it does not have a rectangular shape. Each part of the original picture is called a *Video Object Plane (VOP)*. So a video can consist of different video streams, e.g. one for the background, and one for the person in the front (see figure 1.5). Also different audio objects can be encoded individually. This provides the ability to encode the different audio streams at different rates and with different functionalities.

Because of the individual objects, the system part contains the "composition" function. Another difference is that the system part contains a standard technology to represent time-varying synthetic 3D information. Also a framework to deal with rights protection is provided. The last difference with MPEG-1 and MPEG-2 is that the file format is standardized.

MPEG-4 Video is seen as the standard for the next generation mobile communication. It is also utilized to develop solutions for video on demand.

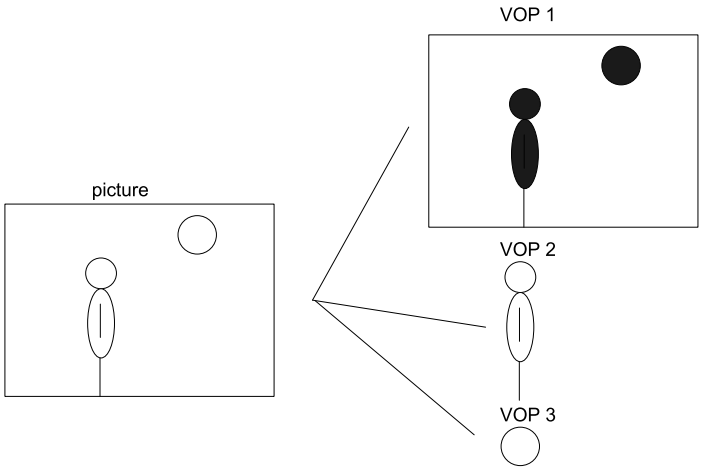


Figure 1.5: Segmentation of a picture onto VOP's

Next to MPEG-1, MPEG-2 and MPEG-4 there are a number of other standards that the Motion Picture Expert Group is working on. MPEG-7 "Multimedia Content Description Interface" is a standard that represents not the audio and video information itself but defines how this information has to be stored in XML format.

The MPEG-21 "Multimedia Framework" standard provides a multimedia framework. It sets out a vision for the future where different users can use content in multiple application domains.

Finally, there are a number of additional standards defined. For these standards only a list of parts is available, no description: the MPEG-A "Multimedia Application Formats", the MPEG-B "MPEG Systems Technologies", the MPEG-C "MPEG Video Technologies", the MPEG-D "MPEG Audio Technologies" and the MPEG-E "MPEG Multimedia Middleware" standards.

1.3 Coding process

In this section an overview of the steps within the MPEG-4 encoding process is given. It is easier to describe the encoding process because all the steps origin in compressing the data. So, in the encoding process the steps have a logical order. The decoding process is taking the inverse steps in reverse order.

Figure 1.6 depicts an overview of the MPEG-4 encoding process. The shape coding is not taken into account because the Simple profile used in this thesis does not support this.

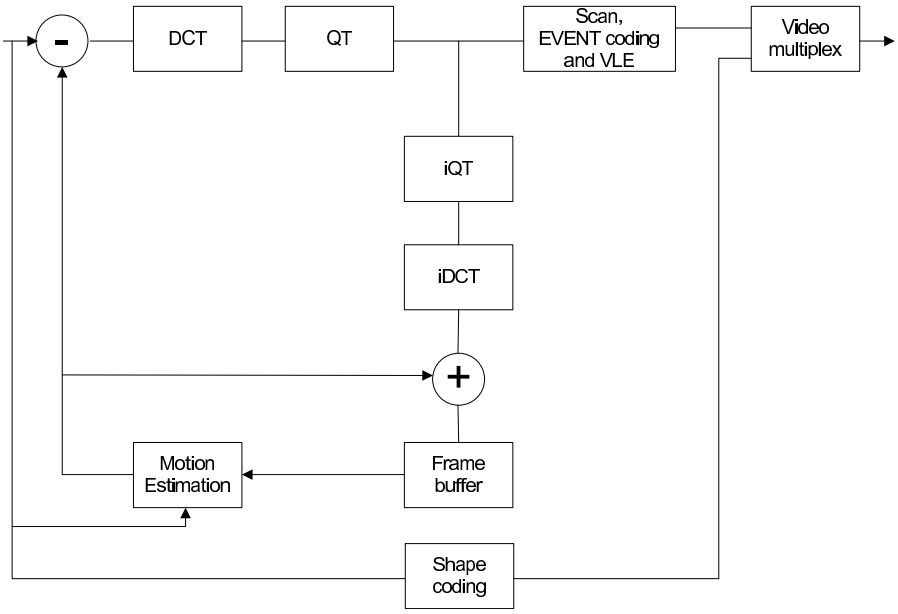


Figure 1.6: Basic block diagram of MPEG-4 Video Coder [2]

The basic block of a MPEG frame is 16x16 pixel *Macroblock (MB)*. Such a MB consists of 6 8x8 pixel blocks. There are 4 luminance blocks (for each pixel one luminance value), one Cb block and one Cr block (see 1.1.1). For every four chrominance values (Cb and Cr) there is only one weighted value stored. This is done by dividing the MB in groups of 2x2 pixels, the average chrominance value of this four pixels is stored.

These 6 pixel blocks are the basic units for the encoding and decoding. On each block the following steps are performed: A Discrete Cosine Transform (DCT), AC/DC prediction, Quantization, scan, Event encoding and variable length encoding (VLE) (Figure 1.7). Coding with this steps is called texture coding. On blocks of P- and B-frames also motion estimation is performed to take advantage of the temporal redundancy. [4] [6]

1.3.1 Motion Compensation

For the P-frames the encoding starts with motion estimation. Motion estimation is a way of describing the difference between consecutive frames in terms of where each

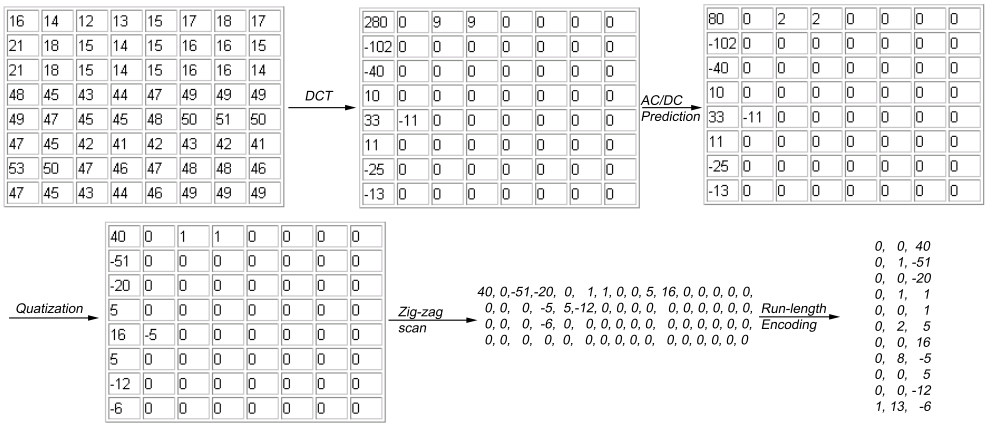


Figure 1.7: Texture encoding: DCT, AC/DC prediction, Quantization, scan (zig-zag), Event encoding and VLE

section of the former frame has moved to. For each MB a MV is calculated. This MV gives the general direction of the motion from that MB; the vector points to a 16x16 block from the reference frame that matches the best (see figure 1.8). The process of finding the best MV is called motion estimation. This process is a very intensive task.

A MV is pointing to a relative point in the previous frames. All pixels are shifted along this vector, so the block is mapped on 8x8 pixels in the previous frame (prediction block). The prediction block is not necessary a single block, it can for example consist of two half blocks. The prediction block can contain samples from at most four blocks. A MV consists of two values, one for the horizontal movement and one for the vertical movement. So, when a sample in a block has coordinate X,Y (coordinate in the whole frame) and the MV is (3,-2) the predicted value is the sample in the previous frame at coordinate X+3,Y-2.

In MPEG-4 it is also possible to use a global MV. The local MVs are relative to the global MV. The global MV can be used when the whole frame moves in the same direction, e.g. when the camera moves.

One, two or four MV(s) may be used per MB. The number of MVs used is set in the header. When one MV is used, all 6 blocks uses this vector. When two vectors are used, for all blocks the average of these vectors is used. In the case that four vectors are used, each vector is used for one luminance block. The chrominance blocks uses the average of the four vectors.

MVs can have three different accuracies: integer accuracy, half-pixel accuracy (in the middle between two pixels) and quarter-pixel accuracy. When a non-integer accuracy is used, the predicted value is calculated by a linear interpolation of the surrounding pixels.

When an integer MV is used this is called *INTRA prediction*, when non-integer MVs are used this is called *INTER prediction*. Whether inter or intra prediction is used is decided in the encoder based on a formula.

If INTER prediction is used, the MVs are coded differentially by performing prediction based on the three MVs of the neighboring MBs: the left, the top and the

right-top neighbors. The prediction is the average of this three neighboring vectors. If one neighbor is outside the VOP, that vector is set to zero. If two neighbors are outside the VOP, the value of the third is used as prediction. When all three neighbors are outside the VOP, the prediction is set to zero.

The difference between the prediction and the original frame is subtracted. This result is called the prediction error. This prediction error is calculated per 8x8 block (for each 6 blocks per MB). This prediction error is encoded in the same way as I-frames. So, for an I-frame there are 6 pixel blocks (with a lot of information) to be encoded and for a P-frame there are 6 pixel blocks (with only the difference) and the MV to be encoded. The pixel blocks are texture encoded (see the following steps), the MV is directly encoded in the video stream. [4] [7] [8]

Decoding

The decoder uses the MVs encoded in the headers to obtain the prediction. In the previous steps (the texture encoding) the prediction error is calculated. When the prediction is added to the prediction error the original frame is decoded. The prediction is acquired by calculating the values where the MV is pointing to for every MB. [4] [9] [7] [10]

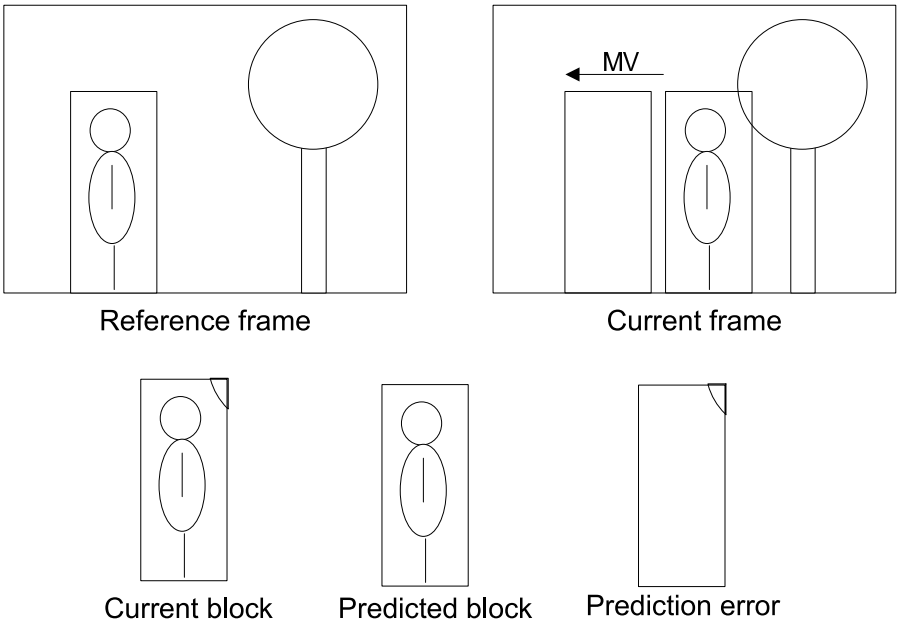


Figure 1.8: Motion estimation

1.3.2 Discrete Cosine Transformation

Next, the pixel blocks are encoded using a Discrete Cosine Transformation (DCT). This transformation transforms the data from the spatial domain to the frequency domain. This gives advantages because images have a strong ”energy compaction”

property: most of the signal information tends to be concentrated in a few low-frequency components of the DCT that is located at the left-top values of the resulting matrix (see figure 1.7). A P-frame will only contain the prediction error. This results in an even stronger energy compaction and compression (see figure 1.9). [11]

16	14	12	13	15	17	18	17
21	18	15	14	15	16	16	15
21	18	15	14	15	16	16	14
48	45	43	44	47	49	49	49
49	47	45	45	48	50	51	51
47	45	42	41	42	43	42	41
53	50	47	46	47	48	49	46
47	45	43	44	46	49	49	49

DCT →

280	0	9	9	0	0	0	0
-102	0	0	0	0	0	0	0
-40	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0
33	-11	0	0	0	0	0	0
11	0	0	0	0	0	0	0
-25	0	0	0	0	0	0	0
-13	0	0	0	0	0	0	0

(a) I-frame (11 EVENT codes)

-3	-2	-2	0	1	2	3	3
2	2	2	2	1	1	1	1
3	3	1	0	-1	-3	-4	-4
-1	-2	-2	-3	-4	-4	-5	-5
-3	-2	-2	-1	0	1	1	2
1	1	2	4	5	6	7	8
2	2	2	2	3	3	3	3
-1	-1	-2	-3	-5	-6	-7	-7

DCT →

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
14	-15	0	0	0	0	0	0
-15	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(b) P-frame (3 EVENT codes)

Figure 1.9: Difference between an I and a P frame after a DCT

Decoding

In the decoder the 8x8 blocks are inverse cosine transformed. The blocks are transformed from the frequency domain back to the spatial domain. The definition for the MPEG-4 iDCT is (there are four slightly different definitions for iDCTs) [10]:

$$f[x][y] = \frac{2}{8}\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)\times C(v)\times F[u][v]\times\cos\left(\frac{(2x+1)u\pi}{16}\right)\times\cos\left(\frac{(2y+1)v\pi}{16}\right)$$

Where F[u][v] are the input samples, f[x][y] the output samples and C(u) en C(v) are defined as follows:

$$C(u), C(v)\begin{cases}\frac{1}{\sqrt{2}} & ,for\ u,v=0\\ 1 & ,otherwise\end{cases}$$

1.3.3 Quantization

After the DCT the 8x8 pixels matrices are quantized. MPEG-4 quantization introduces a weighting factor into the process. The purpose of this is to exploit properties of the

human visual system. Since human eyes are less sensitive to some frequencies, these frequencies can be quantized with a coarser step-size then more important frequencies. This results in a matrix with as many as possible zeros with a minimal distortion in the picture (see Figure 1.7). [4, 12]

Quantization is done by dividing the DCT matrix by a fixed quantization matrix. Higer coefficients of the quantization matrix results in a coarser step-size of the quantization which will give higher compression. So, the quantization determines the trade-off between compression and loss of information. Quantization is the only lossy step in MPEG-4 encoding (RGB to YCbCr conversion loses data when not all chrominance samples are stored, but is officially not part of the standard).

Decoding

The inverse quantization consists of three different steps: the inverse quantization itself, the saturation and mismatch control (see Figure 1.10). In this figure QF[v][u] is the quantized input matrix value, the quant_scale_code is the scaler, W[v][u] is the value of the quantization matrix, F''[v][u] is the non-saturated value, F'[v][u] is the saturated value and F[v][u] is the output value.

There are two forms of inverse quantization used, the H.263 method and the MPEG-4 method. Which method is used is a parameter of the MPEG-4 header. Inverse quantization is different for DC and AC coefficients. For H.263 iQT the DC coefficient of a block in a I-frame is reconstructed as follows:

$$F''[0][0] = 8 \times QF[0][0]$$

And the rest of the coefficients:

$$F''[v][u] = \begin{cases} 0 & ,if\ QF[v][u] = 0 \\ 2 \times QP \times QF[v][u] + QP + 1 & ,if\ QF[v][u] \neq 0,\ QP\ is\ odd \\ 2 \times QP \times QF[v][u] + QP - 1 & ,if\ QF[v][u] \neq 0,\ QP\ is\ even \end{cases}$$

MPEG-4 inverse quantization is also different for I- and P-frames. The DC coefficient of the blocks of I-frames is obtained by multiplying it with a scaler. This scaler can be different for every MB and is encoded in the header. The non-saturated values are computed as follows:

$$F''[0][0] = scaler \times QF[0][0]$$

To reconstruct the other coefficients for both block of I- and P-frames (the AC coefficients and the DC coefficient from the blocks of P-frames) the following equation is used:

$$F''[v][u] = \frac{2 \times (QF[v][u] + k) \times W[v][u] \times scaler}{32}$$

Where k is 1 for blocks of I-frames and k = sign(QF[v][u]) for blocks of P-frames: negative samples stay negative. Samples of an I-frame are always positive, the P-frame contains the error and can therefore contain negative values. For P-frames another quantization matrix is used than for I-frames.

Next, the coefficients (both DC and AC) are saturated. The coefficients have to be in the range [-2048,2047]. So, the coefficients are saturated in the following manner:

$$F'[v][u] \begin{cases} 2047 & F''[v][u] > 2047 \\ F''[v][u] & -2048 \leq F''[v][u] \geq 2047 \\ -2048 & F''[v][u] < -2048 \end{cases}$$

Finally, the mismatch control is carried out to compensate the mismatch between DCT and iDCT. Only coefficient F[7][7] is compensated if necessary. First the sum of all coefficients from the block is calculated. If the sum is even, the coefficient at 7,7 is corrected in the following manner:

$$F[7][7] \begin{cases} F'[7][7] - 1 & ,if\ F'[7][7]\ is\ odd \\ F'[7][7] + 1 & ,if\ F'[7][7]\ is\ even \end{cases}$$

The rest of the coefficients are not changed. [10]

The result after the inverse quantization is an 8x8 matrix with coefficients in the range [-2048,2047].

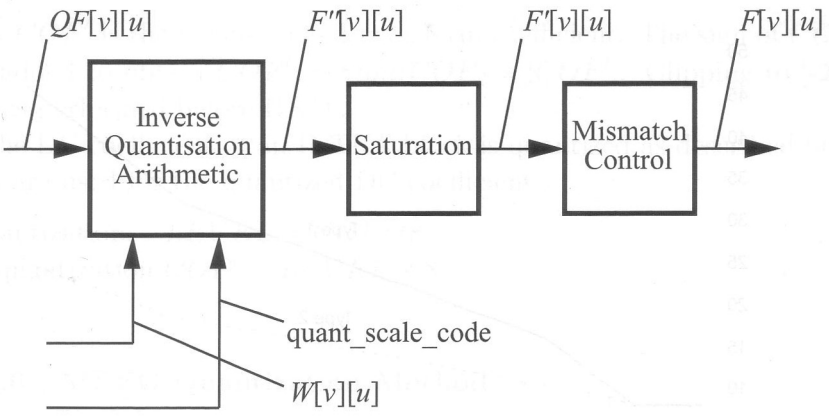


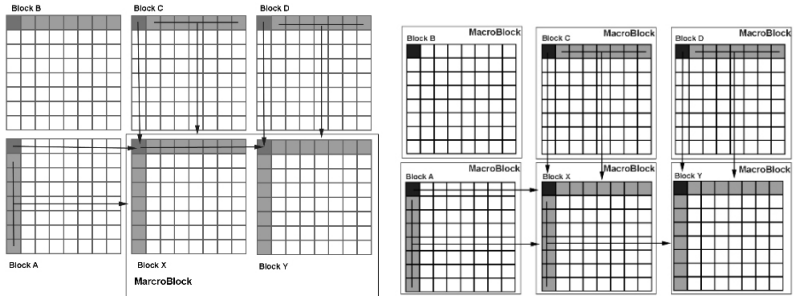
Figure 1.10: Inverse Quantization process [10]

1.3.4 AC/DC Prediction

A new feature in MPEG-4 encoding is the AC/DC prediction. All other steps described here are also used in the previous MPEG versions. To gain more compression, some of the values of the transformed matrix are predicted. The DC coefficient is the coefficient at position 0,0, the rest of the values are AC coefficients. There exists a statistical dependency between some of the AC and DC coefficients. Therefore, the value of one can by predicted from coefficients of neighboring blocks (see Figure 1.11). Prediction is only carried out on the first row or the first column of AC coefficients and the DC coefficient of each block. The prediction is alway based on the above or left neighbors. This is because in the decoding process these neighbors are already decoded, so the values are available. Luminance blocks are predicted based on luminance blocks and chrominance block predictions on chrominance blocks. These blocks can be in the same MB or in another MB. The Cr and Cb blocks are always predicted based on the Cr and Cb blocks from the previous MBs (see figure 1.11). [3] [9] [10]

Whether DC prediction is enabled is a parameter of the MB header. The predicted DC value is as follows obtained:

$$DC_{X'} = \begin{cases} DC_C & ,if\ |DC_A - DC_B| < |DC_B - DC_C| \\ DC_A & ,otherwise \end{cases}$$



(a) luminance blocks (b) chrominance blocks

Figure 1.11: AC/DC prediction based on adjacent blocks

Where DC_A is the DC value of the horizontal adjacent block (see Figure 1.11), DC_B the DC value of the left-top adjacent block, DC_C the DC value of the vertical adjacent block and $DC_{X'}$ is the predicted value. The differential value, $DC_X - DC_{X'}$, is stored. When an adjacent block is outside the frame the DC value of that block (for prediction) is set to zero.

Decoding

The decoding process is similar. The prediction value is obtained in the same way. The original value is the prediction value added to the differential (stored) value.

The AC prediction is also based on the horizontal or vertical adjacent blocks. The direction of the DC prediction (horizontal or vertical) is also used for the AC prediction. Because of that, AC prediction can only be enabled when DC prediction is also enabled. Besides, AC prediction is only used for intra frames (I-frames).

When the horizontal direction is used, only the first column is predicted, with vertical direction only the first row (accept the coefficient at 0,0, the DC coefficient).

The predicted values are calculated with the following formula:

$$\underset{\texttt{horizontal}}{QAC_{[i][0]X'} = \frac{QAC_{[i][0]A} \times QP_A}{QP_X}} \qquad \underset{\texttt{vertical}}{QAC_{[0][j]X'} = \frac{QAC_{[0][j]C} \times QP_C}{QP_X}}$$

Where $QAC_{[i][j]A}$ is the AC coefficient, QP_A is the AC quantifier of block A and $QAC_{[i][j]X'}$ is the predicted value. When an adjacent block is outside the frame, the corresponding AC values are set to 0 and the QP value is assumed to be equal to QP_X.

The differential values (original value minus predicted value) are stored. The decoder obtains the original value by adding the predicted value to the differential value.

The decision to enable or disable the AC prediction is based on criterion S. For horizontal prediction the criterion is calculated as follows:

$$S = \sum_{i=1}^{7} |QAC_{i0X}| - \sum_{i=0}^{7} |QAC_{i0X'}|$$

For vertical prediction the criterion is done in the same way but instead of prediction based on block A prediction based on block C is used.

For all blocks in the MB a common decision is to be made, so the sum of all criterions S is calculated and the AC prediction decision is made on the rules below:

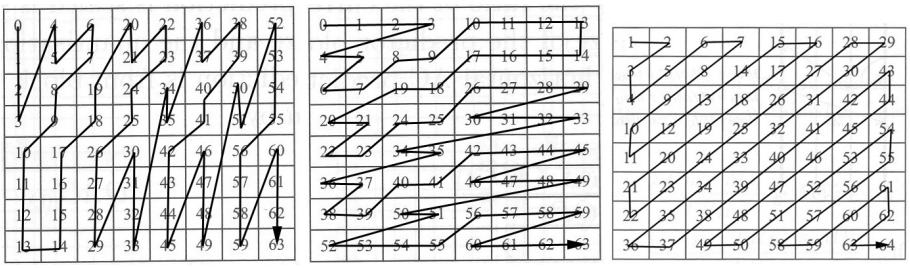
$$\begin{cases} \textit{enable AC prediction} & ,if \ \sum S > 0 \\ \textit{disable AC prediction} & ,otherwise \end{cases}$$

A flag in the header indicates whether AC prediction is enabled or disabled.

1.3.5 Scan methods, Event decoding en VLE

To store or transport the matrix it has to be mapped from an 8x8 matrix to a 1x64 vector. For some cases there are different scan methods (depending on the AC/DC prediction). In the end of the vector are long runs of zeros.

There are 3 scan methods employed to transform the 8x8 matrix to a 1x64 vector (see figure 1.12). The used method depends on the DC prediction direction. P- and B-frames always use the zig-zag scan. For I-frames, if the AC prediction is disabled, the zig-zag scan is selected. If AC prediction is enabled and the DC prediction is from the horizontal adjacent block the vertical-alternate scan is used, otherwise the horizontal-alternate scan is used. Which prediction mode and direction is used can be calculated from the preceding blocks (see previous section). [4]



(a) Alternate-horizontal scan (b) Alternate-vertical scan (c) Zig-zag scan

Figure 1.12: Scan methods

The 64 coefficients are EVENT encoded. Because of the scan, there are long runs of zeros. The length of zero runs together with a non-zero coefficient and a flag if it is the last non-zero coefficient are encoded as a triple (last, run, level):

```
LAST  0: there are more nonzero coefficients in the block, 1: This is
        the last nonzero coefficient
RUN   Number of zero coefficients preceding the current nonzero
      coefficient
LEVEL The coefficient itself
```

In most cases, this reduces the number of coefficients.

Finally, the most common events are variable length encoded using a special type of Huffman coding. Variable length encoding means that not all data items are encoded using the same amount of bits. More common data items are encoded with fewer bits than the ones that occur less frequently. The table with codes is a fixed table, so there are fixed variable length codes. The rare events that are not variable length encoded are just encoded with 22 bits with special escape bits, starting with an escape sequence [4, 10, 11]:

```
ESCAPE 7 bit escape sequence
LAST   1 bit
RUN    6 bit
LEVEL  8 bit
```

1.3.6 Headers

The video bitstream of MPEG4 is arranged in a structured composition of four layers: Picture Layer, Group of Blocks Layer (the MBs in a picture are divided in blocks), Macroblock Layer and Block Layer. Each layer has its own header. Some headers are optional and when there is a header some parts of the header are also optional. The most important parts of the headers are discussed in this paragraph, a complete description can be found in [4].

Each picture in the video bitstream has its own header. The header provides information about the size (QCIF, CIF, enz.), the type (I,B,P) and a couple of other defines like the type of MV and the prediction modes coded in the PTYPE field. The quantizer is defined in the PQUANT field. The data, the Marcroblocks, are in the GOB LAYER field. The EOS field is a flag that is used when all picture of a stream are send [9].

The Group of Blocks Header is not send for the first group of blocks in each picture. For the other GOBs, the header may be empty.

In the MB header is again defined to which kind of frame this MB belongs (I, P or B). There is also a differential quantizer coded (coding the difference in the multiplication factor). This differential quantizer indicates four possible changes in the quantizer (-1, -2, 1, 2). The MVs are also defined in this header.

The Block Layer does not contain a real header. This layer consists of two fields, the IntraDC field and the TOOEFF field. The IntraDC field contains the DC value in case of an I-frame and is otherwise empty. In an I-frame the DC value is not entropy encoded and quantized separately. The TOOEFF field contains the rest of the (coded) coefficients [9].

1.3.7 Color conversion

To convert the YCrCb values to RGB values a whole MB is required. For converting a luminance block to its RGB values, a quarter of a Cr block and a quarter of a Cb block are used (because only one-fourth of the chrominance samples were stored, see section 1.1.1). Converting the YCrCb values to RGB is done with the formulas mentioned in the first paragraph of this chapter:

$$
\begin{aligned}
R &= 1.164 \cdot (Y - 16) + 1.596 \cdot (Cr - 128) \\
G &= 1.164 \cdot (Y - 16) - 0.813 \cdot (Cr - 128) - 0.392 \cdot (Cb - 128) \\
B &= 1.164 \cdot (Y - 16) + 2.017 \cdot (Cb - 128)
\end{aligned}
$$

As mentioned above, for every coordinate there is an luminance sample available, but not for every chrominance vector. Because of there are one-fourth of the chrominance samples, every chrominance sample is used four times (1.13).

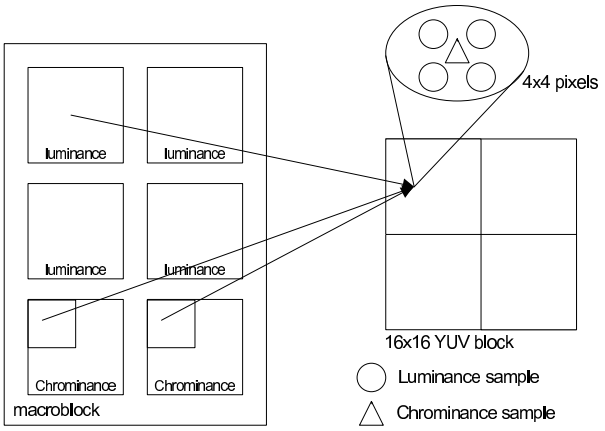


Figure 1.13: Location of YCrCb samples in a macroblock

CHAPTER 2

Reconfigurable and Heterogeneous Architectures

Modern day's embedded systems have a large variety of requirements like high performance, energy-efficiency, real time behavior, reliability and flexibility. The set of requirements for a specific system is quite often contradictory. For example, mobile devices like mobile phones, multimedia players and PDAs require both high performance and high energy-efficiency. The major opportunity to deal with those problems is the domain specific application of an embedded system.

In this chapter we start with a list of algorithm properties that are important for the architecture. The next two sections give an introduction into two special (often used) architectures in embedded systems: reconfigurable and heterogeneous architectures. The fourth paragraph gives a short introduction into the communication between different architectures. The chapter ends with a description of the used architecture for this project, the BCVP.

2.1 Algorithm properties

There are many different types of algorithms, each with their own properties. Algorithms can be computational intensive, memory intensive, control intensive, etc. For a good performance it is important to map an algorithm on a processor that is capable of executing an algorithm efficient. With efficient executing we mean that the processor is able to execute the algorithm in a energy-efficient way. For example, mapping an memory intensive algorithm on an architecture that has a high memory access time and/or a small memory gives a miserable performance [13]. One property that many algorithms have in common in mobile applications is that they are streaming DSP applications. Examples of streaming algorithms are HiperLAN/2, WiMaX, DAB, DRM, DVB, UMTS and multimedia processing like MPEG [14].

2.1.1 Streaming algorithms

Streaming DSP algorithms express computation as a data flow graph with streams of data items (the edges) flowing between computation kernels (the nodes) [14]. An example of such a flow graph is shown in Figure 2.1. The nodes represent the processes and perform the actual computation, the edges represent the communication lines. The data streams through the graph from node to node over the edges. In this thesis we assume that the graph is deterministic.

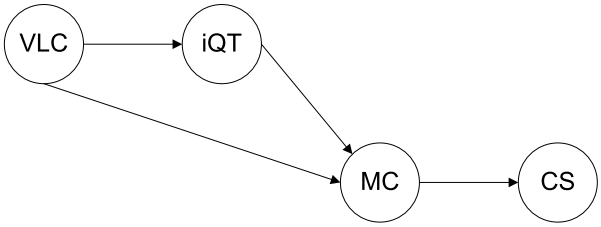


Figure 2.1: Streaming Process Graph

In a deterministic process graph, the data flows always in the same way through the nodes in a pipelined fashion. This gives a predictable temporal and spatial behavior, which can be used to map the algorithm on parallel processors. The local processing in the nodes is relatively simple, but there is an huge amount of data. The data rate (IO) can be very high in streaming applications. Therefore, the communication will dominate the energy costs rather than the processing. [14]

2.1.2 Parallelism

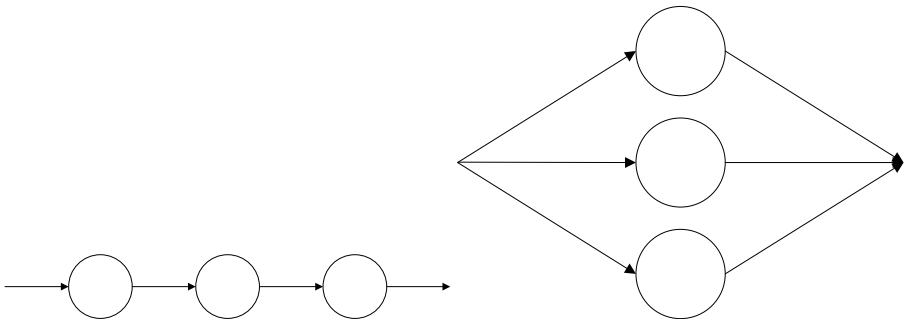
Parallelism means doing multiple things at the same time, or pretend to do multiple things at the same time. Personal Computers are pretending to handle multiple processes in parallel. In fact they do not really do two things at the same time. They divide the single processor in small time slices and the different processes can use alternately the CPU. This form of parallelism is called *functional parallelism*. When there are actual multiple processing units and there are really multiple processes running at the same time, this is called *architectural parallelism*.

For streaming algorithms parallelism means doing the calculation of multiple nodes at the same time, so architectural parallelism. There are two forms of architectural parallelism: spatial and temporal (Figure 2.2).

Temporal parallelism

Temporal parallelism looks like pipelining. There are multiple processing units, the first processing unit starts the calculation on a sample and sends the result to the next processing unit. While the next processing unit performs the calculation on the first sample (the result of unit one) the first processing unit can start the calculation on the next sample (Figure 2.2a). This form of parallelism can be used for every multiple node algorithm by mapping the successive nodes on successive processing units.

Another form of temporal parallelism is to implement the application twice. Mapping the process graph (or only the slowest part) twice on the architecture results



(a) Temporal parallelism (b) Spatial parallelism

Figure 2.2: Temporal and spatial parallelism

in two implementations that can process a sample in parallel. This form of parallelism can be used for every algorithm when there are no data dependencies and the processes are stateless.

Temporal parallelism does not decrease the latency of an algorithm, every processing step is done successive in time, just like it is done on only one processing unit. The gain is an increasing throughput.

Spatial parallelism

Spatial parallelism means that there are several calculation done on the same data item on the same time (Figure 2.2b). Spatial parallelism is not possible for all algorithms. The input of the different processes (nodes) has to be independent of each other, i.e. if an process uses the result of another process as input, these processes can not be calculated at the same time. Figure 2.3 shows a process graph, the processes in a box in the graph can be calculated at the same time. Spatial parallelism does not only increase the throughput, but also decrease the latency because several steps of the algorithm are calculated at the same time.

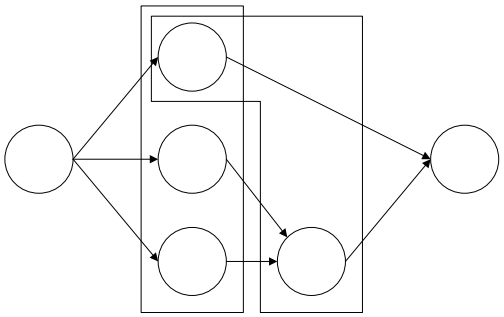


Figure 2.3: Parallelism in the process graph

2.1.3 Algorithm characteristics

There are four areas of specializations in a processor: the data paths that perform the actual operations, the memory architecture, the interconnect architecture and the control architecture [13].

In [15] a set of properties for algorithm characterization is proposed. The properties of this set required to elect the most suitable architecture for an algorithm are:

- *Size* depicts the number of nodes in the flow graph of a process, the bit width, the number of i/o operations, the number of memory accesses and the required amount of memory.
- *Concurrency* measures the number of operations that can be executed at the same time.
- *Temporality* captures information about the lifetimes of variables.
- *Spatial locality* characterizes the degree of natural clusters of computations in the algorithm, where amounts of computation can be done independently.
- *Regularity* is the amount of common patterns within the algorithm.
- *Control flow properties* determines the structure of the control flow and interaction between loops, e.g. the number of (conditional) jumps.

Important algorithm properties for the data path are concurrency and size. To exploit the concurrency, the data path must support computation of multiple operations at the same time. The size covers the bit width of the operations. Furthermore, the complexity of the operations and the support for complex operations by the data path are important (e.g. multiplications and division).

The size and temporality determine the requirements for the memory architecture. The lifetimes of the variables determine the requirements for the registers (number of registers) and caches. There must be enough memory available and the number of memory accesses determines the requirements for the memory architecture.

The concurrency, regularity and size determine the preferences for the interconnect. To exploit the concurrency the interconnect architecture must support the concurrent interconnect operations. The regularity determine how often the interconnect has to be reconfigured and how many configurations are required. The number of i/o operations determine requirements for the interconnect architecture to the i/o devices (e.g. speed).

Important for the control architecture are the size (number of different operations), regularity and control flow properties. The last two properties determine the flexibility and amount of control. The spatial locality is a degree for the spatial parallelism.

2.2 Reconfigurable Architectures

With a reconfigurable architecture we mean an architecture with a reconfigurable instruction set. The available instruction set after configuring is a subset of the total instructions that can be performed by the architecture. This results in an architecture which adapts to the algorithm.

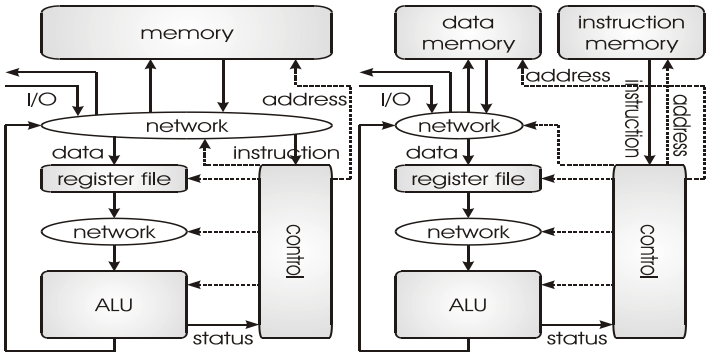
2.2.1 Introduction

The four most common processor architectures are the General Purpose Processor (GPP), the Field Programmable Gate Array (FPGA), the Digital Signal Processor (DSP) and the Application Specific Integrated Circuit (ASIC). All these architectures have their own advantages and disadvantages.

Classic Architectures

A GPP is the most flexible architecture. The GPP is used in all personal computers. It can be programmed to compute any algorithm, which is its major advantage. It has a simple instruction set available, but a sequence of instructions make complex operations possible. Because the instructions are stored in the main memory, there is in practice no limitation on the number of instructions used. Because this architecture is so often used, there are a lot of compilers available. Implementing an algorithm on such an architecture can be done very fast in a wide variety of languages. The disadvantages of the GPP are its poor performance and its energy-inefficiency.

Originally, GPPs where designed according to the *Von Neumann* architecture (Figure 2.4a). In a Von Neumann architecture the data and the instruction are in a single memory. Only one memory read can be done in one cycle on a memory. Therefore, in every memory clock cycle only a data item or an instruction can be read. The speed of the GPP has become many times faster than the speed of memory, so the memory is the bottleneck in the speed of the GPP. This bottleneck is called *the Von Neumann bottleneck*. One way to relieve this bottleneck is split memory into a data memory and an instruction memory. This is called the *Harvard architecture* (Figure 2.4b). The second way is to use a memory hierarchy with fast caches. [16]



(a) Von Neumann architecture (b) Harvard architecture

Figure 2.4: Von Neumann and Harvard architecture [16]

A DSP architecture resembles a GPP, but is optimized for digital signal processing operations. DSPs have a larger instruction set than an GPP, but the instruction set of an DSP is specialised on DSP algorithms. The DSP has more combinatorial hardware and less control hardware. DSPs are more energy-efficient and have a better performance than a GPP when they are used for digital signal processing [16].

An FPGA consists of a two dimensional array of *Logical Cells (LC)*. An LC is the processing unit of an FPGA, which can perform simple operations. In some FPGAs there are also memory components available in the LCs. The LCs can be connected via a configurable matrix of wires (see Figure 2.5). By configuring the interconnects between the LCs and the LCs itself, the FPGA performs a particular function after configuration. A configuration can be seen as an instruction. Therefore, FPGAs have an unlimited instruction set, but there is only one instruction stored.

Because the FPGA has to be configured for every wire and every LC, the amount of configuration information is considerable (order of Mb) and loading the configuration into an FPGA is slow. When another algorithm has to be computed, the whole FPGA has to be reconfigured, even if there is only a slight adjustment in the algorithm (some FPGAs support partially reconfiguration). So, an FPGA is not very flexible. Developing an algorithm for an FPGA is not very easy, because *Hardware Description Languages (HDL)* are used to describe the functionality. These languages are much more complicated than high level languages. Furthermore, compilation of HDL to configurations requires more time. Because of the architecture of the FPGA its capable of exploiting a lot of parallelism. An FPGA has a good performance for algorithms matching the hardware: algorithm that manipulate bits and have a lot of parallelism. An advantage of the FPGA is that its much more energy-efficient than a DSP [16].

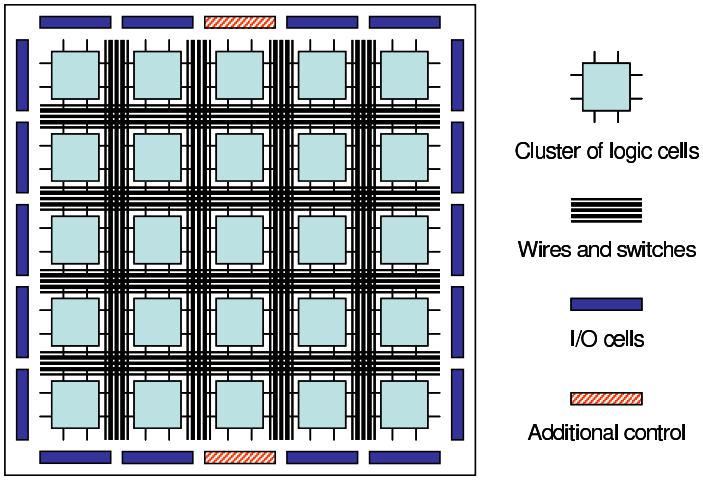


Figure 2.5: FPGA architecture [16]

An ASIC is an integrated circuit designed for a specific algorithm. An ASIC implementation of an algorithm is the fastest compared to the other architectures. But once an ASIC is produced, it can not be changed anymore. So, for every algorithm another ASIC has to be designed and produced, even for an update. Designing an ASIC is also done in HDL. Next to that an ASIC is very fast, it is also the most energy-efficient choice [16].

Reconfigurable Architectures

The conclusion is that a GPP is the most flexible architecture, but with the worst performance and energy-efficiency characteristics and an ASIC is not flexible at all but

has the best performance and energy-efficiency. So, a trade off is possible between flexibility, performance and energy-efficiency (Figure 2.6). However, for many applications both flexibility and energy-efficiency are required, for example for mobile devices [13].

There is a gap between the flexible, not so energy-efficient DSP and the energy-efficient, not flexible ASIC (see Figure 2.6). This gap is filled by reconfigurable architectures. One of these architectures is the FPGA, but the FPGA is optimized for certain algorithms.

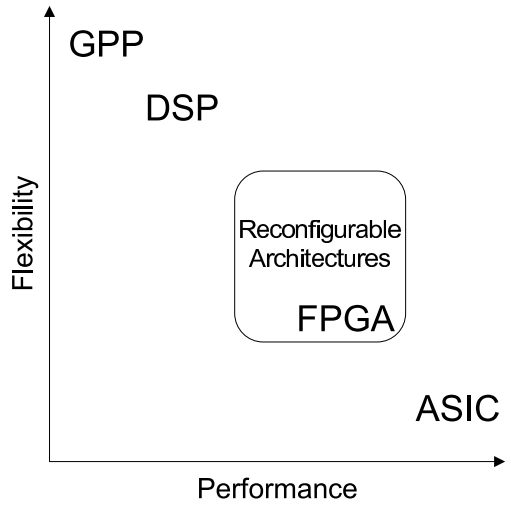


Figure 2.6: Tradeoff between flexibility and performance

So, the demands for a reconfigurable architectures are: flexibility, good performance, energy-efficiency and re-usability. Reconfigurable architectures try to find a compromise between these requirements. Developing an universal reconfigurable architecture is an illusion. To satisfy the requirements, domain specific reconfigurable architectures are used. These architecture are flexible, have a good performance and are energy-efficient within a particular algorithm domain. [17]

2.2.2 Reconfigurable Architecture properties

Reconfigurable architectures have specific properties. The most important properties are discussed below. [16]

Dynamic versus Static reconfiguration

Infrequent reconfiguration (e.g. once an hour or once a week) is known as static reconfiguration. Frequent reconfiguration is known as dynamic reconfiguration. For dynamic reconfiguration (e.g. once a second) the configuration information has to be small. There are some architectures that use two configuration spaces [16]. In this way the offline configuration space can be reconfigured while the online configuration space is used and when needed it can swap to the other configuration instantly. Dynamic configuration also allows time-sharing of hardware. When for example an

algorithm computes two multiplications, both multiplication could be done on one multiplier (when it fits in time).

Fine grained versus Coarse grained

The granularity of an architecture is defined as the width of its components in the datapath. We define a datapath of four bits or less as fine grained. Fine grained architectures need more configuration information and configuration overhead, because it can be configured on detailed level. Word-level operations in a fine grained architecture gives a lot of overhead, because for each word operation multiple bit configurations have to be set. Coarse grained architectures need less configuration information than fine grained architectures. However, bit manipulating algorithms can be done much more efficient on fine grained architectures.

Levels of reconfigurability

There are different levels of reconfiguration. In a reconfigurable architecture a subset of the total instruction set can be used without reconfiguration. On an FPGA this is only one (large) instruction, a MONTIUM (see Section 2.5.2) can use more instructions without reconfiguration. For the re-usability the configuration information should be as small as possible and it is preferable that the architecture could be reconfigured at run-time. Reconfiguration also should not take too much time. [17]

Design automation

For a reconfigurable architecture it is important that there is enough high-level tooling available. Low level languages like HDLs and assembly require a lot of knowledge of the underlying architecture and experience before the optimizations for standard code constructions (i.e. loops) can by fully exploited. Besides that, developing in a low-level language requires much more time than in a high-level language. On the other hand, compilers have sometimes difficulties with exploiting all the parallelism and optimizations for a particularly architecture, so for the best implementation the developer has to have some knowledge about the architecture. One solution is using a high-level language in which it is possible to have more control over the mapping on the hardware, for example with the *Configuration Description Language (CDL)* (see Section 2.5.2).

2.3 Heterogeneous architectures

A conclusion of the previous section is that different processors have all their own advantages and every processor is best in a particular application domain. Fine grained reconfigurable architecture are best in bit level operations, coarse grained reconfigurable architectures in word level operations and GPPs have the best performance in control intensive and irregular tasks (with a lot of different instructions). Combining different processors gives a system that can efficiently execute a large variety of algorithms.

2.3.1 Heterogeneous architectures

Today, many applications include algorithms or parts of algorithms that require (for best performance) different processors. For example, MPEG-4 contains very control and memory intensive parts (VLC, MC) and very regular, computation intensive parts (iDCT, CS). That is why *Heterogenous Architectures* become popular. Heterogeneous architectures are architectures that consists of different types of processors. For example, combining a GPP with a reconfigurable processor gives good performance for both control intensive and computationally intensive tasks.

When in an heterogeneous architecture the reconfigurable processor can be used for two different tasks (because that fits in time) it has to be reconfigured, otherwise it would require two such processors due to the lack of a reconfiguration ability. The advantage of combining a reconfigurable processor with another processor such as a GPP is this reconfiguration. The GPP can reconfigure the reconfigurable processor when necessary, without the need of an external signal or person. [13]

Besides high performance for a single application, mobile devices also require an architecture that can efficiently run different types of algorithms. Mobile phones are not only used to make phone calls, but also to listen MP3s, listen to the radio, watch videos and play games. So, there are a lot of different applications and therefore a large variety of algorithms. The workload on a mobile device is very dynamic. When the user watches for example a movie on his phone, the phone is downloading video (via UMTS), encodes (MPEG) it and shows it on the screen. When the phone is in standby, it is only contacting the telephone network now and then.

The algorithms have to be mapped on the processor that is the best suitable to fulfill the task to get a good performance and efficiency. Processors of the architecture that are temporarily not used (because of the variable demand of processing power) need to be switched off or switched to low-power mode.

Heterogeneous tiled architectures can be helpful to achieve the performance goal in an energy-efficiency way. By integrating different types of processors on the device it can get a good performance in case of high load. The control process maps the different algorithms to the architectures that are best suitable for it. When there is little load, parts of the tiled architecture can be switched off or switched to power save mode to get a considerable reduction in energy consumption. When load decreases and one or more processors are switched off it can be necessary to move a process from a processor which is switched off to another processor. Therefore, a run-time remapping tool is required. [14]

2.4 Network on Chip

When multiple processors are combined in one architecture, a communication infrastructure between these processors is required. This communication has to have enough bandwidth to transport the (streaming) data from one processor to another. When more than two processors are used, a programmable infrastructure is preferred. Which processors communicate with each other is related to the application.

Some kind of communication is required on a multi-processor platform. In the case of only two processors, a point-to-point link between this two processors satisfies. But when there are more processors, communication possibilities between

every two processors are required. When for every tuple of processors a point-to-point link would be implemented, it requires a lot of hardware. Therefore, some flexible solution is required. One solution is the use of a bus. Drawback of this solution is that only two architectures can communicate at the same time. A better solution is a *Network on Chip (NoC)*.

A NoC consists of point-to-point links and routers. Every processor is connected with a router through a point-to-point link. Routers are connected to each other. Every tuple of processors can communicate and there are multiple communication paths at the same time available. For the routers, just like for ethernet networks, scheduling and routing algorithms are required. And just like for ethernet networks, there are several methods and algorithms available. Which is the best algorithm is subject of current research.

2.5 Used architecture: BCVP

For this report the BCVP is used. This is a heterogeneous architecture with reconfigurable parts. It is the concept platform of the Smart Chips for Smart Surroundings (4S) project. [1]

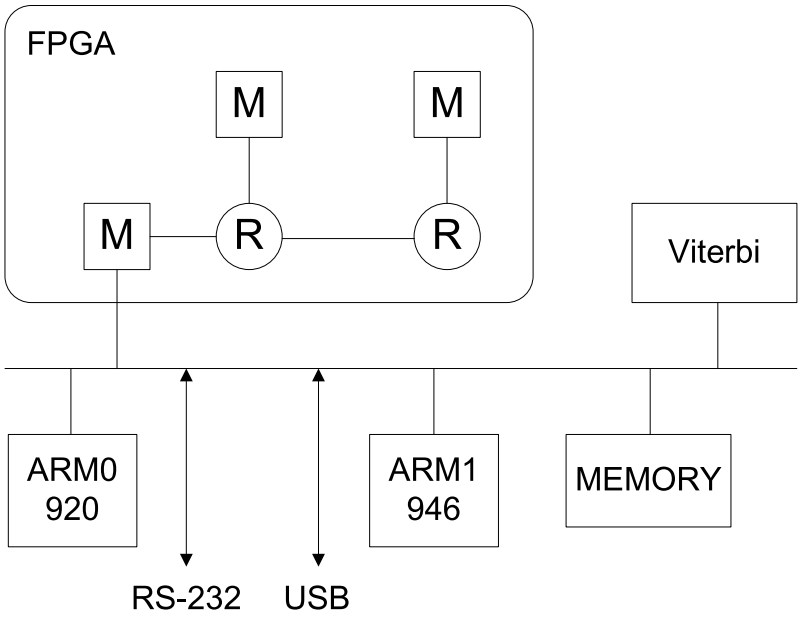


Figure 2.7: Basic Concept Verification Platform architecture

2.5.1 Layout

This architecture contains two GPPs, a fine-grained reconfigurable part (FPGA) and an ASIC tile [1]. In this master project the FPGA is configured with multiple Montiums (see Section 2.5.2) connected via a small NoC. These Montiums itself are coarse-

grained reconfigurable architectures. The configuration of the BCVP is depicted in Figure 2.7.

All the communication between the different parts is done via shared memory and the AMBA bus. Hardware drivers between the actual hardware and the memory handle the communication between the hardware and the memory.

For the General Purpose Processors two ARMs are available, the ARM 946E (ARM1) and the ARM 920 (ARM0). For our implementation, most of the tasks are performed by ARM1 (946) because this processor starts at boot time and this processor can access all memory on the chip. ARM0 is hardly used.

The FPGA on the BCVP is a Xilinx XC2V6000. Optionally this FPGA can be configured to emulate a couple of Montiums. On the BCVP there is also an ASIC available. This is a Viterbi decoder. This ASIC is not used in this project. On the ARM runs an Operating System that is developed at the University of Twente. [18]

2.5.2 Montium

The Montium is a reconfigurable architecture developed at the University of Twente by Paul Heysters [16]. The Montium contains five *Arithmetic Logical Units (ALUs)* with their own register file of 16 registers and ten 16-bit wide memories with a dept of 1024 words. Figure 2.8 depicts the total architecture of the Montium.

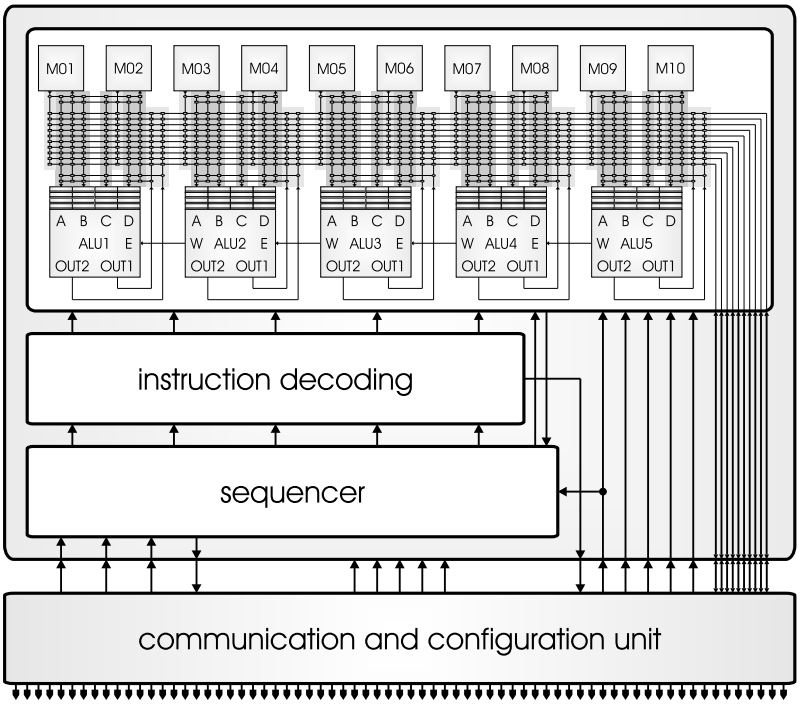


Figure 2.8: Montium architecture [16]

Each ALU is connected with a local interconnect to two memories. One combination of an ALU, its registers and the two memories is called a *Processing Part (PP)*. There are 5 such PPs available on one Montium. Such a set of PPs is called a *Processing*

Part Array (PPA). An ALU can do computation on data that is in the register file. The register bank is divided in four parts, every part can contain four data items. Every part is directly connected to an input of the ALU. Therefore the ALU can do calculation on 4 data items every clock cycle, but these data items have to be in different parts of the register. Next to the input data from the register bank is there an east-west connection between the PPs. This east-west connection is within the combinatorial datapath without registers.

Between the PPs and the memories there are 10 global busses for transport of data to another memory than the two local memories, for data transport between PPs and for data transport to and from the *Communication and Configuration Unit (CCU)*. The CCU is the interface to the NoC. There are four 16-bit lanes to the outside and four 16-bit lanes from the outside. The total width of the datapath (except some parts of the ALU) is 16-bit. [19]

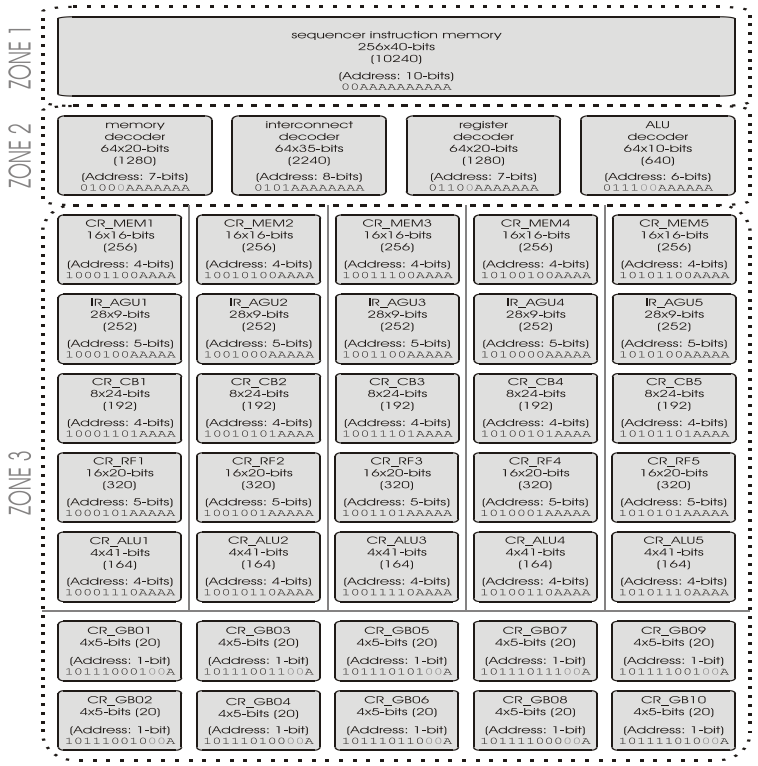


Figure 2.9: Montium configuration hierarchy [16]

A Montium is controlled by the sequencer. This sequencer contains an instruction memory with a depth of 256 instructions. In this memory there are the encoded instructions for the ALU, memories, registers and the interconnect. So, there are only 256 different sequencer instructions possible without reconfiguration. It is possible to perform jumps within the sequencer (i.e. at the end of the code jump back to the beginning to run the same code again), so an algorithm can also perform loops and more than 256 clock cycles can be run without reconfiguration.

The sequencer behaves like a state machine. The decoding of the state into hard-

ware instructions is done in stages. The sequencer instructions call a decoder, which internally selects the hardware instruction. Both the decoder instructions and the hardware instructions are stored in registers. Not all combinations of instructions fit in the registers. Therefore, only a subset of the total possible (combinations of) instructions can be used without reconfiguration. Figure 2.9 shows the staged memory hierarchy. The figure shows the configuration hierarchy of the MONTIUM implementation described in [16]. There are differences in the memory mapping with the current, used implementation, but the hierarchy principle remains the same.

The staged decoding of the hardware instructions result in a very small configuration space and thereby in a small reconfiguration time. The MONTIUM can be partially reconfigured. Drawback of these registers is that the number of different instructions (without reconfiguration) is small. For example, there is for every ALU an instruction register with a depth of 8 available. This results in 8 different instructions for every ALU. The instruction register for every global bus has a depth of four, so there are four configurations possible for one global bus.

The MONTIUM is capable to read 4 (16-bit) samples from the outside and write 4 samples to the outside every clockcycle. The FPGA implementation of the MONTIUM has an clockspeed of 6.6 MHz. This results in a (theoretical) communication bandwidth of 50 MB/s. The ASIC implementation of the MONTIUM has an expected maximum clockspeed of 100 MHz. This would result in a bandwidth of 750 MB/s. The combination of the high bandwidth and a small configuration makes the MONTIUM suitable for computationally intensive, regular tasks width a high communication bandwidth.

The ALU has four inputs and two outputs. The ALU itself is divided into two levels: a level with four functional units and a level with a multiplier and a butterfly structure with two adders/subtracters. Figure 2.10 shows this structure.

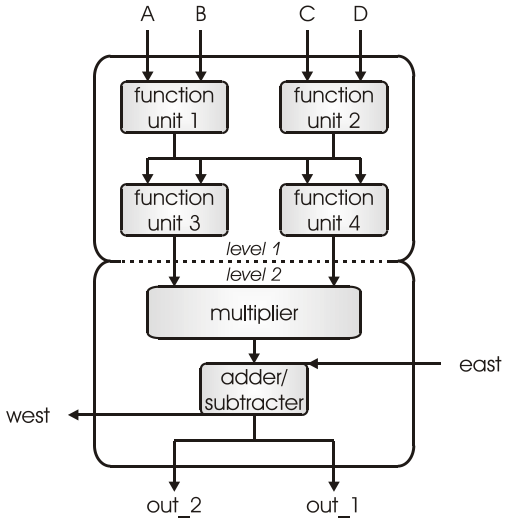


Figure 2.10: Montium ALU [16]

The functional units in level one of the ALU implement general arithmetic and logical operations that are available in languages like C [16]. Examples of these arith-

metic operations are (saturated) additions, minimum and maximum of two values and absolute values. The logical operations are operations like bitwise ”and” and ”or”, bitwise inversion and the negate operation.

Level two of the ALU is in fact a multiply-accumulate (MAC) structure with a butterfly structure below. This level is able to multiply two values, add a value to it and in the butterfly add and subtract another value.

The ALU can do complex computations in one clock cycle. But, especially with complicated computations, not all combinations of registers are possible. It is possible to multiply A and C and add and subtract D from it, but is not possible to multiply A and B and add and subtract C from it. The ALU can do integer calculations as well as fixed point calculations.

CHAPTER 3

MPEG-4 implementation on the BCVP

This chapter describes which parts of the MPEG-4 standard are implemented on the Montium. The first section starts with an argumentation on which processor of the BCVP (or type of processor) each part of the MPEG-4 standard could best be implemented. The next two sections describe the implementations of the iQT and CS on the Montium. The last section studies how MC could be implemented on a FPGA.

3.1 MPEG-4 processes

For this thesis we use the MPEG-4 implementation of the 4S project [1]. It implements the Simple Profile and Advanced Simple Profile. The format of the video stream is QSIF (176x144 pixels) and SIF (352x288 pixels). The input is a MPEG-4 video stream, the output is the RGB frames that can be displayed on a monitor.

The 4S implementation is a streaming implementation. The application consists of four processes: variable length coding (VLC), inverse quantization (iQT), motion compensation (MC) and color space conversion (CS). The first process, VLC, contains the VLC algorithm, header decoding, EVENT decoding, inverse scanning and AC/DC prediction. The iQT process exists of the iQT algorithm and the iDCT. The MC process contains the MC algorithm, the CS process executes the CS algorithm.

All processes are based on frames. The process starts when a whole frame is available. Therefore, the buffers of the pipes have to be big enough to contain a whole frame. Figure 3.1 shows the data rate between the processes per frame.

The VLC algorithm searches in lookup tables whether a code is available. The length of the code is variable. Therefore, it is not possible to map a code directly to a memory address. The algorithm starts searching with the minimum number of bits a code can have (length of the smallest codeword). If this code is not available it searches with one bit more, etc. There are optimizations for this problem, but most of them require even more memory [12]. Next to searching and accessing memory, and the memory usage it is an essentially sequential process. Only after a code is decoded

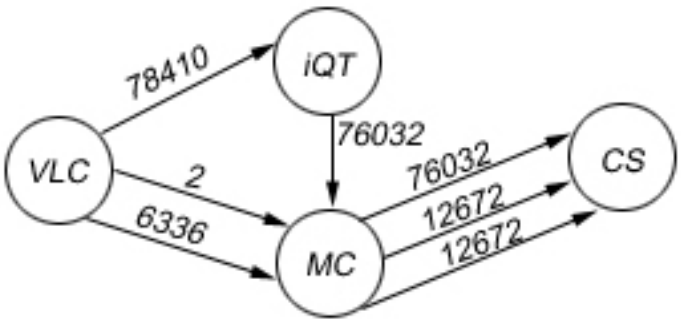


Figure 3.1: Data rate between MPEG-4 processes in bytes per frame

the next code can be decoded because the length of a code is variable. So the VLC algorithm can best be implemented on the ARM.

The iQT and the iDCT are very computational intensive algorithms. In particular the iDCT requires a lot of processing power [20]. Some algorithms described in literature for iDCT contain a lot of parallelism. Next to this, the algorithms only require input blocks of 64 bytes, 2 times 64 bytes for the inverse quantization and some operands for the iDCT. These algorithms seem to be suitable to be implemented on the Montium. Section 3.2 describes the implementation on the Montium and the problems implementing it.

MC is locating the reference block in the memory and adds the prediction error. These additions are simple integer additions (8 bit). For every input sample there is one addition. The most challenging part of MC is locating the reference block in the memory based on the motion vector (MV). These are not always 64 consecutive addresses. For example, when the image is stored row by row, the first eight samples of the reference frame are in consecutive addresses but the ninth sample is one row lower. Implementing MC on the Montium is not possible because there is by far not sufficient memory to save a complete reference frame. Furthermore, there is less parallelism possible because for almost every operation a memory access is required (to the same memory). The MC could be implemented on a processor with enough memory and processing power to generate the addresses, for example a FPGA. How it could be implemented on a FPGA is pointed out in Section 3.4.

The Color Space conversion is a straightforward algorithm consisting of five multiplications and seven additions per pixel. These operations can be (partially) done in parallel. Three input samples are used for every iteration and three output samples produced. Not all input samples are really streamed, a part of the samples is used multiple times. All together, this algorithm is suitable to be implemented on the Montium.

3.2 Inverse Quantization on the Montium

The process iQT consists of two algorithms, the iQT and the iDCT. The iQT algorithm is different for I- and P-frames and every frame has its own multiplication factor. The

iDCT is for all frames exactly the same. Therefore, the process requires next to the 64 samples of every block also a flag for the I- or P-frame selection and the multiplication factor as input.

3.2.1 Inverse Quantization

The iQT algorithm is described in Section 1.3.3. For an I-frame the iQT consists of two multiplications, for a P-frame of two multiplications and an addition (or subtraction). This last addition or subtraction is difficult. If the sample is negative, the sample must be decreased by one, if the sample is positive the sample must be increased by one and if the sample is zero it must stay zero. For I-frames, the same equation as described in Section 1.3.3 is used. For P-frames, the equation is a bit changed:

$$F''[v][u] = \frac{2 \times (QF[v][u] + k) \times W[v][u] \times scaler}{32} = (QF[v][u] + k) \times \frac{2}{32} \cdot W[v][u] \times scaler$$

The value stored in memory is not the matrix value W, but $\frac{2}{32} \cdot W$. The operations and their mapping on the Montium is shown in Figure 3.2. The mismatch control is not shown in this figure. For the mismatch control all samples are summed up during the quantization. This is done in parallel on PP4. Afterwards, checking whether the last sample has to be changed or not and change the sample is done on PP4 in two clock cycles. One for checking whether the sum of all samples is odd or even and one for adding or subtracting one of the last sample of the block.

I-frame	P-frame
1: $X_1 = sample[i][j] << 4$	1: $X_1 = \begin{cases} (sample[i][j] + 0) << 4 \text{ , } if\ sample[i][j] >= 0 \\ (sample[i][j] + 2) << 4 \text{ , } otherwise \end{cases}$
2: $X_2 = scaler \times matrix[i][j]$	2: $X_2 = scaler \times matrix[i][j]$
3: $X_3 = X_1 \times X_2$	3: $X_3 = \begin{cases} X_1 \times X_2 + X_2 \text{ , } if\ X_1 <> 0 \\ X_1 \times X_2 + 0 \quad \text{ , } otherwise \end{cases}$

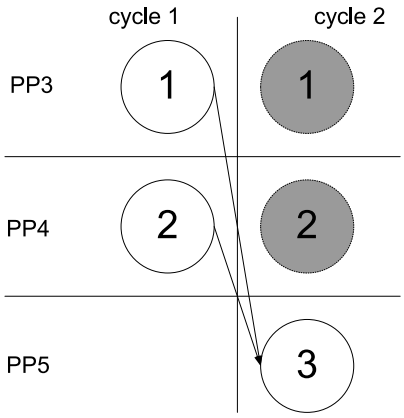


Figure 3.2: iQT mapping on the Montium, arrows represent data dependencies

The mapping is in such a way that the iQT for I-frames and for P-frames both use one clock cycle. This is the clock cycle which is also required to stream in the data

and store it on the right place in the memory for the iDCT. So, after implementing the iDCT, implementing the iQT hardly increases the number of clock cycles. Only the last sample (with the mismatch control) and the pipeline fill requires some extra clock cycles (3 in total).

The mapping shows how the addition/subtraction problem is solved. This is done by subtracting two if the sample is smaller than zero and nothing otherwise after multiplying the sample with two (node 1 in Figure 3.2). After multiplying the result of node 1 and 2 (in node 3) the result of node 2 is added another time if the result of node 1 is not zero. So, the implemented equations are:

$$\begin{cases} ((sample \times 2) - 0) \times (scaler \times matrixValue) + (scaler \times matrixValue) & , if\ sample > 0 \\ ((sample \times 2) - 0) \times (scaler \times matrixValue) + 0, & , if\ sample = 0 \\ ((sample \times 2) - 2) \times (scaler \times matrixValue) + (scaler \times matrixValue) & , if\ sample < 0 \end{cases}$$

3.2.2 Inverse Discrete Cosine Transform

Implementing an iDCT in a straightforward way is very slow. For every coefficient in the block there are 8×8 multiplications of 5 numbers. There are also cosine numbers among these numbers and for this cosine numbers itself there are also multiplications required (or they have to be stored in memory). Implementing the straightforward way with a few optimizations showed that 16384 multiply accumulate operations are required for each block. The definition of a 2D 8x8 iDCT is:

$$f[x][y] = \frac{2}{8} \sum_{u=0}^{7} \sum_{v=0}^{7} C(u) \times C(v) \times F[u][v] \times \cos\left(\frac{(2x+1)u\pi}{16}\right) \times \cos\left(\frac{(2y+1)v\pi}{16}\right)$$

Where F[u][v] are the input samples, f[x][y] the output samples and C(u) en C(v) are defined as follows:

$$C(u), C(v) \begin{cases} \frac{1}{\sqrt{2}} & , for\ u, v = 0 \\ 1 & , otherwise \end{cases}$$

There are a lot of different ways of implementing the iDCT algorithm. All these ways have one thing in common: the 2D iDCT is split in 16 1D iDCTs. For a 1D iDCT there are much more optimizations known than for a 2D iDCT [20]. The definition of a 1D iDCT is:

$$f[x] = \sum_{u=0}^{7} C(u) \times F[u] \times \cos\left(\frac{(2x+1)u\pi}{16}\right)$$

The 2D iDCT can be calculated by calculating the 1D iDCT on each row and on each column of the block.

One of the most optimized ways of implementing an iDCT is Chen's method [20,21]. Chen derived a way of implementing a 1D iDCT with 16 multiplications and 26 additions without complicated memory operations or divisions. The whole 2D iDCT requires 256 multiplications (16*16) and 416 additions (16*26). A flow graph of Chen's algorithm is given in Figure 3.3.

We have chosen to implement the method of Chen because this algorithm uses almost the least number of multiplications and additions and it uses a few constants

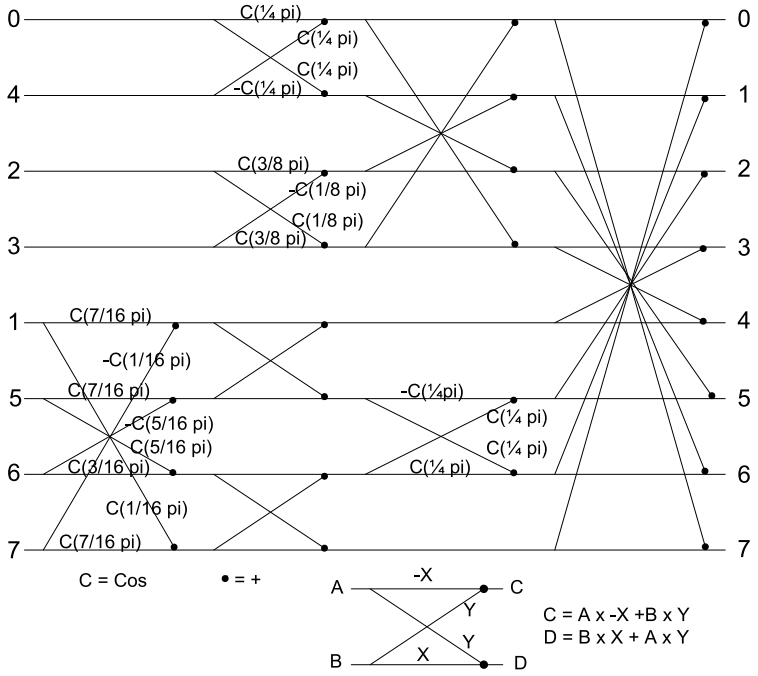


Figure 3.3: Flow graph Chen's iDCT algorithm

and no other operations (like bit shifts). Figure 3.4 shows the mapping of the algorithm on the Montium. An explanation of which operation each node performs and how the nodes are connected is given in appendix A.

The samples are multiplied by 16 in the iQT. In this way the samples become 12.4 fixed point numbers (12 bits for the representation of the digits before the radix point, 4 bits for representation of the digits after the radix point). This decreases the errors caused by rounding the intermediate values. Without this shifting the rounding errors are up to 10%. The samples are shifted four positions because there is underflow (smaller than zero) and overflow (bigger than 255) possible for the intermediate values. Before streaming the samples out the results are multiplied with 1/16. The multiplier rounds the result of the multiplication. Shifting the samples four positions to the right (flooring) results in errors with a maximum of one (too small). But, because the MC adds all samples, the errors are added.

3.2.3 Dataflow

The position of the samples in the memory is very important. The first two clock cycles of each 1D iDCT four samples of a row are read. These samples have to be in different memories because the Montium can read at most one word from the same memory per clock cycle. The first eight iDCTs are performed on the rows of the matrix, the next eight are performed on the columns. Four memories are used for the input and four for the output. Storing the results of the first iDCTs (on the rows) in such a way that the four samples required at the same time for the iDCT on the columns are in different

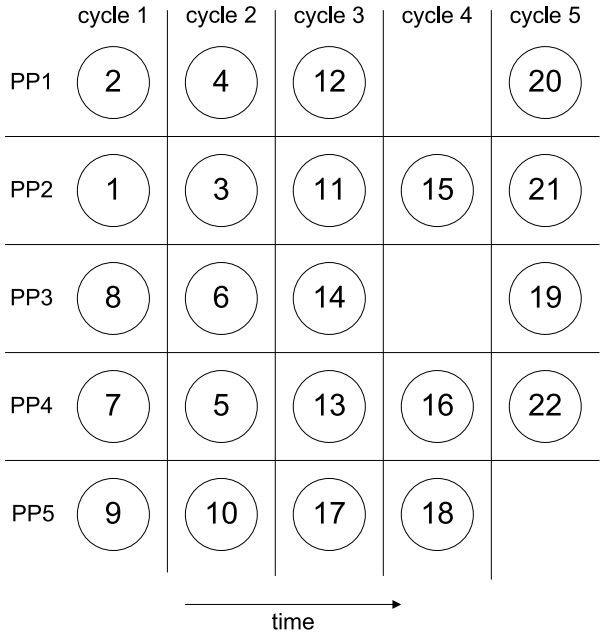


Figure 3.4: Scheduling iDCT algorithm on the Montium

memories is not possible. The values of one column are produced by the same PP, this PP should then store the results in different memories. This is very irregular and therefore there are not enough configurations for the interconnect. So, the matrix must be transposed between the row iDCTs and the column iDCTs. Consecutively the samples are read and the iQT is performed, eight 1D iDCTs are carried out, the matrix is transposed, another eight 1D iDCTs are performed, the matrix is transposed back and the samples streamed out.

The total number of clock cycles for the iQT required for one block is given in Table 3.1.

Configuring the Montium to execute the iQT process requires 1249 16 bit words. To load the operands, another 235 words are required. Every clock cycle one word can be loaded, so in total 1484 clock cycles are required to configure the Montium.

Streaming in and iQT	67
Eight 1D iDCTs	40
Transpose matrix	16
Eight 1D iDCTs	40
Transpose and stream out	64
TOTAL	227

Table 3.1: Number of clock cycles for the iQT algorithm on the Montium per block

3.3 Color Space Conversion on the Montium

The CS changes the color space from YCrCb to RGB. This requires three input samples (Y,Cr,Cb) and produces three output samples (R,G,B). For every *MacroBlock (MB)* there are four blocks of Y samples but only one block with Cr samples and one with Cb. So every Cr and Cb sample is used four times.

3.3.1 Mapping the algorithm

The formulas for the CS are:

$$R = 1,164 * (Y - 16) + 1,596 * (Cr - 128)$$
$$G = 1,164 * (Y - 16) - 0,813 * (Cr - 128) + 0,392 * (Cb - 128)$$
$$B = 1,164 * (Y - 16) + 2,017 * (Cb - 128)$$

The input samples are integer values but the operands are non-integer. The Montium has no floating point arithmetic. Therefore, we have to implement the formulas in fixed point notation. The input and output samples are eight bit values, so the point can be shifted seven bits to the left to prevent rounding errors (one bit overflow). Shifting the point to the left can be done by multiplying one of the operands of the multiplication by 128. The non-integer operands are multiplied, it become rounded integers (1.164 becomes 149). Before the results are send to the output, the results of the multiplications have to be shifted back and the results have to be clipped. Clipping is required because a result bigger than 255 does not fit in eight bits anymore. When only the least significant eight bits are used (by masking) without clipping, this results in big errors. The least significant eight bits of 256 results in 0 (and an error of 255). Results can become bigger than 255 because of the rounding errors.

Shifting to the right and clipping is done in one clock cycle. Overshoot of the results is caused by rounding errors, the result does not become much bigger than 255. Therefore, the samples do not become bigger than 511. Then the sample is nine bits and the most significant bit is set. Since the result is shifted to the left by seven, the most significant bit (bit 16) is set and therefore the result becomes negative. So, when the result is negative the result must be 255, otherwise the result has to be shifted seven to the right. The negative check and optional shifting to the right can be done on the Montium in one clock cycle. Figure 3.5 shows all required operations (operations in one node can be done in one clock cycle on a Montium PP) and their dependencies. In the dependency graph the mapping is already included.

3.3.2 Dataflow

In every iteration a new luminance sample is read from the input. The chrominance samples have to be stored because they are used four times. In the current MPEG-4 implementation the Y, Cr and Cb samples are streamed in the Montium over different lanes. The order of the samples is pixelwise row after row from the left to the right. Every chrominance sample is used two successive iterations and then at the next row again for two iterations (see Section 1.1.1). For the odd rows three samples are read

1: $X_1 = 149 \times (Y - 16)$
2: $X_2 = X_1 + 204 \times (Cr - 128)$
3: $X_3 = X_1 - 104 \times (Cr - 128)$
4: $X_4 = X_3 + 50 \times (Cb - 128)$
5: $X_5 = X_1 + 258 \times (Cb - 128)$
6-8: $X_{6-8} = \begin{cases} 255 & ,if\ X_i < 0 \\ X_i >> 7 & ,if\ X_i \geq 0 \end{cases}$
where X_i is the input value of the node

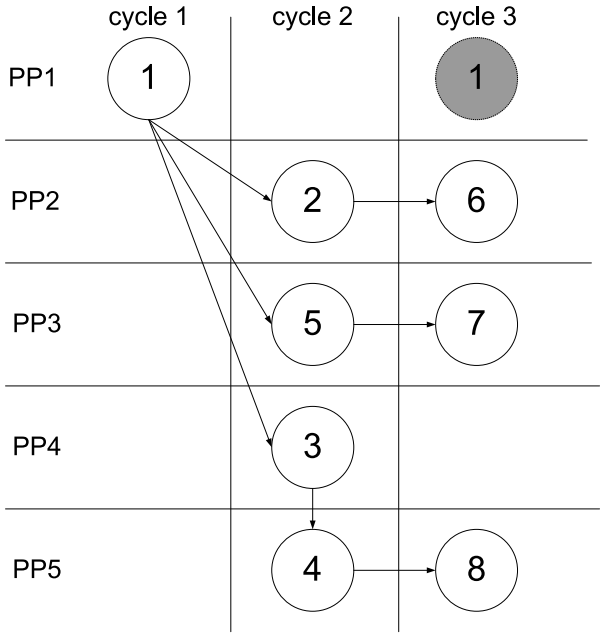


Figure 3.5: Mapping CS algorithm on the Montium

in the odd columns and the chrominance samples are stored in memory. In the even columns only a luminance sample is read and the chrominance samples from the previous iteration are used. For the even rows only a luminance sample is read, in the odd columns the chrominance samples are derived from the memory. In the even columns these chrominance samples are used again.

The total number of clock cycles required for the CS is 1(setup) + 2 x 256 (2 for each pixel) = 513 for every macroblock (MB).

To configure the Montium for this process, 416 words are required (16 bit). Loading the operands require 45 words. In total, 461 clock cycles are required to configure the Montium to perform the CS.

3.4 Motion Compensation

As mentioned above the MC can not be implemented on the Montium because of the large memory required. It could be implemented on a FPGA.

The basis of the implementation are two memories. One memory with the reference frame and one memory where the resulting frame is stored. For the next

frame, the resulting frame is the reference frame. After decoding a total frame the memories are swapped (or preferably only the pointers).

The samples have to be stored in such a way that the address generator can easily locate an arbitrary block of 8x8 samples when the left-top coordinate is known. The best way of storing the data is pixel wise row for row, first the luminance samples and then the chrominance samples. The adresses of an 8x8 block in the memory are, starting at the left-top coordinate, eight successive addresses, then skip as much addresses as there are samples in a row, than another eight successive addresses, etc.

The left-top coordinate of the reference block can be derived with the left-top coordinate of the current block in the frame and the MV. Adding this two vectors gives the left-top coordinate of the reference block. Storing the block in the resulting frame happens in the same way. The left-top coordinate of the block is known so the addresses where the samples have to be stored can be exposed as described above.

Because the memories have to be swapped after each frame there is a control signal required to signal that the end of a frame is reached. Next to a control signal for the end of the frame is a signal required for selecting an I-frame or a P-frame. Motion compensation is only performed on P-frames, not on I-frames. But an I-frame is a reference frame, it has to be stored in the memory. For keeping regularity, can this be solved by adding zero to every I-frame sample by using a multiplexer which decides whether the value from the reference memory is added or a zero is added. The complete implementation is shown in Figure 3.6.

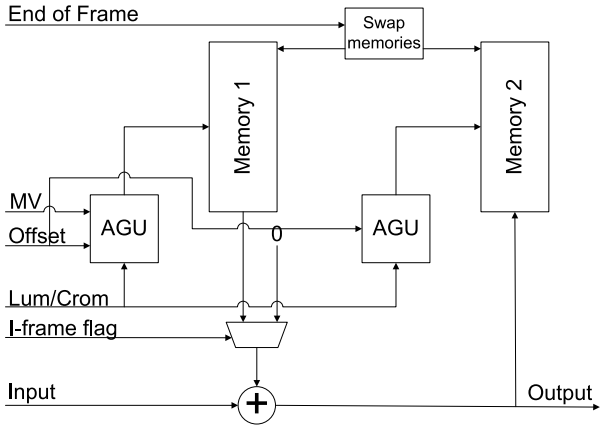


Figure 3.6: Possible MC implementation on a FPGA

This solution is suitable for the most basic form of MC, using only P-frames and integer MVs. When also B-frames are used, not all frames are reference frames and the frames are decoded out of order.

Nevertheless, almost the same implementation can be used. Despite that the frames are decoded out of order, the reference frames (I- and P-frames) are decoded in order, only the position of the B-frames in the stream is changed. The B-frames are decoded directly when both (previous and successive) reference frames are decoded. Decoding an I- and P-frame is done in the same way as described above. The result after decoding an I- or P-frames is that the latest I- or P-frame is in a memory and

the previous I- or P-frame is in the other memory. Both reference frames, which are required for decoding a B-frame, are available. If this B-frame is not stored and the memories are not swapped multiple B-frames which require these reference frames can be decoded. The next I- or P-frame gives no problems, this can be decoded in the standard way. So, there is only additional hardware required for reading the second memory instead of storing to the second memory (see Figure 3.7).

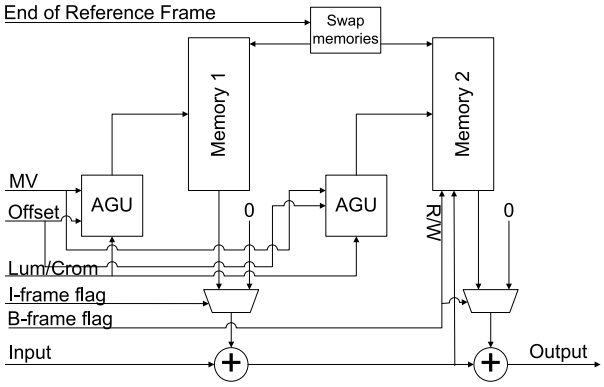


Figure 3.7: Possible MC implementation on a FPGA, also suitable for B-frames

Also non-integer MVs can be handled with a slightly different implementation. For a MV that points between two pixels, the average of this two pixels is used. A MV can point horizontally between two samples and vertically. The estimation can be a single pixel (MV does not point between pixels), the average of two pixels or the average of four pixels. Therefore, for implementing non-integer MVs a calculation unit has to be added which calculates the average of four pixel values. The resulting value goes to the multiplexer. The address generation unit must always generate four addresses. These can be the same addresses, two different addresses or four different addresses. The memories have to support reading four addresses at the same time. The rest of the implementation can stay the same (see Figure 3.8).

3.5 Conclusion

The VLC process is a control and memory intensive process. Therefore, it can be executed best on a GPP. MC requires a lot of memory, but it is a very regular process and the calculation is straightforward. A Montium does not have sufficient memory to store a complete reference frame. Therefore, the Montium is not suitable to execute the process. MC could be executed on a processor architecture with sufficient memory and enough processing power to generate the addresses, for example an FPGA. More complex versions of MC (B-frames, non-integer MVs) require only little changes of the implementation on a FPGA.

The Montium is suitable to execute the iQT and CS. Both processes are regular and require only a small amount of memory. For the iQT and CS implementation, the parallelism of the Montium can be exploited to speed up the process.

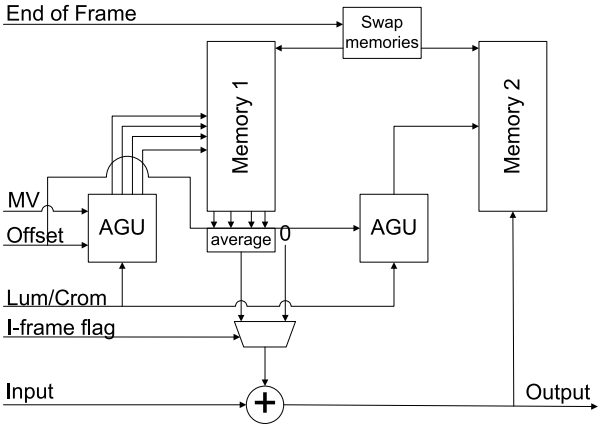


Figure 3.8: Possible MC implementation on a FPGA, also suitable for non integer MVs

CHAPTER 4

Interprocess communication on the BCVP

For implementing a streaming application on a particular architecture communication between processes is required (interprocess communication). Communication between processes on the same GPP can be handled in the OS of the tile. For communication between processes on different processors drivers are required. These drivers have to manage the hardware between the processors and the interfaces of the processors. The BCVP is a heterogeneous architecture, it consists of multiple (types of) processors. The University of Twente is developing an OS for the ARMs on the BCVP, BasOS [18]. Since the BCVP is intented for streaming applications, there has to be a possibility for streaming communication between processes. The usage of the communication must be very simple. Therefore, there is also an Controller developed, which configures the streaming communication.

The first section of this chapter gives a description of the requirements of the interprocess communication. The second section gives an overview of the implementation choices. The last section is a manual for the usage of the Controller. Implementation details of BasOS are described by van Sisseren [18].

4.1 Requirements

There are requirements defined for the interprocess communication on the BCVP. These requirements encompass the communication between processes, the configuration of the Montium and NoC and the Controller.

- Run processes on PC, BasOS and Montium
  The processes of an application can run on the PC, ARM or on the Montium. There has to be a communication possibility between these processors. Communication between the BCVP and the PC is done with an USB connection. For the communication with the Montium, the bridge from the ARM to the NoC, and the NoC are used. Therefore, it has to be possible to configure the NoC routers,

so the data can stream from and to the Montiums and between the Montiums. Because the Montium is a reconfigurable architecture, the Montium has to be configured. Next to communication between processes on different architectures, the processes running on the ARM must also be able to communicate with each other.

- Standardised communication interface between tasks on PC, ARM and Montium
  The communication interface of the processes running on different processors have to be standard. When a process sends data to another process, it makes no differences whether that process is running on the PC, ARM or Montium.

- Easy usage of communication between processes
  Communication between the processes has to be as simple as possible. Processes running on the Montium have a fixed way of communicating (read/write the lanes). The hardware handles the rest [16]. For processes on the ARM there has to be an easy, standardized way of communicating with other processes, independent whether they are running on the ARM, Montium or PC. Processes on the PC have to communicate over the USB connection in a determined way. How this is implemented on the PC is out of the scope of this project.

- Dynamically reconfigurable (run-time)
  The mapping of the processes of an application can change at run-time. The communication has the be set up in such a way that remapping a process to another processor can be done at run-time. A process can for example move from the ARM to the Montium while the application is running.

- Configuration System
  For a user friendly system a configuration system is required. This system initiates the processes, the communication between the processes and the configuration of the routers and Montiums in the correct way and the correct order based on a configuration. It has to be as easy as possible for the user to implement an application in the shape of a process graph to the system.

4.2 Implementation

This section gives a description of the implementation of the interprocess communication and the Controller on basis of the requirements described in the previous section.

4.2.1 Basic idea

An application is defined as a process graph: a set of processes, signals and communication channels (pipes). A process starts when it receives a signal. Signals can come from pipes or from other processes (4.2.4).

Processes can run on the ARM, Montium or the PC. The processes are signal driven. A process needs signals to start. If all signals connected to a process are received, the process starts. If there are no signals connected to a process it will not

start. When all input data is ready and there is enough free memory for the output, then the process starts it calculation (the developer has to make sure the processes receive signals at the right time). In this way, the process does not block. Non-blocking processes are in the first place required to guarantee the real-time constraints [18]. For replacing a process implementation by another implementation it is also required that a process terminates in finite time to a known state. When a process is non-blocking it is running or not. It is not blocked in the middle of the calculation. This means that when the process is not running it can be replaced with another process. The processes are stateless, they do not keep any state information. In this way the process can easily be replaced with another implementation (e.g. on another processor) without worrying about keeping the state information and initiating the replacing process in a particular state. In BasOS, it is possible to implement processes blocking. But, it is preferable for the real-time constraints and run-time remapping to implement processes non-blocking.

The data flows through the graph in a message based way: the streaming data is divided in message items. It is alway possible to define streaming communication as a message based communication. The processes always require a minimum amount of data to perform the calculation. For example, most processes in MPEG-4 can perform their operations on a MB, so the minimum amount of data required is one MB (the message size in the current implementation is one frame). This minimum amount of data can be defined as the message size. Defining the communication in a message based way makes the run-time remapping much easier. If communication is message based, it is possible to change the destination process of a communication channel between two messages. The previous iteration of the process runs on the old processor and the next iteration of the process runs on the new processor.

For the configuration system it is important to know on which processor a process is running. In the first place the programming language (and therefore the binary code) is different for an ARM and a Montium. Furthermore, it is required to configure the routers of the NoC for using the Montiums. When the application is initiated, there is knowledge required about the mapping of the processes. Connecting two ARM processes uses another initialization of the communication than connecting an ARM process with a Montium process. The goal is that there are no differences in interface for the processes itself. The Controller initiates the right (type of) communication, but there is also a difference in the configuration when for example one of the processes run on the Montium. One of the differences is that the routers have to be configured.

4.2.2 Processes

Processes on the ARM

Processes running on the ARM are in fact processes running on BasOS. All functions these processes can use (e.g. read/write standard out) are defined in BasOS [18]. Because BasOS is a real-time operating system, the processes are real-time or run in slack time [18]. The main difference between a real-time process and a non-real-time process is that a real-time process has a deadline. Furthermore, a real-time process has a period. A real-time process runs multiple times, when it finishes its execution it returns to the waiting queue until it receives signals and is executed again. A process running in slacktime runs only once, when it finishes its execution it starves [18]. If the

CONTROLLER is used, every process receives pointers required for the communication and signaling as argument every iteration. In this way the configuration can change, e.g. a process is remapped, without the previous or successor process mention it. Only the arguments change.

Processes on the Montium

To run processes on the MONTIUM, the MONTIUM has to be configured. To configure the MONTIUM a configuration is send over lane 1 (first lane). Therefore, when a MONTIUM is configured, there has to be a direct communication channel between the ARM and the MONTIUM. On run-time there might be no communication between the ARM and the MONTIUM (e.g. when the MONTIUM receives its data from another MONTIUM). Sometimes it is required to reconfigure the routers of the NoC after the configuration of the MONTIUM. The CONTROLLER handles this for the developer (see Section 4.2.6).

4.2.3 Pipes

For the communication between processes pipes are defined. A pipe is a FIFO (First In First Out) buffer. The size of this buffer is defined when the pipe is initialized. A pipe consists of the buffer and a read and write threshold (these thresholds are described below). For reading or writing the pipe only the pointer to the pipe is required, the pipe is not connected to a process. Because the CONTROLLER gives an argument struct with this pointer to each process every iteration it is easy to change the source or destination pipe of a process. Four functions are available for reading and writing the pipes [18]: a blocking and non-blocking write and a blocking and non-blocking read.

So, reading and writing the pipes can be done blocking or non-blocking. It is preferred to use non-blocking reads and writes. Non-blocking is required to guarantee the real-time constraints and for the run-time remapping.

Pipes can also generate signals. A signal can be generated when there is free memory in the buffer (not full) or when data is available in the buffer (not empty). When a buffer is not full can be defined by the write threshold. This is the number of bytes which has to be available in the buffer before the signal is generated. For the not empty signal there is a read threshold which defines the number of bytes available in the pipe before the signal is sent. Figure 4.1 shows the signals and thresholds.

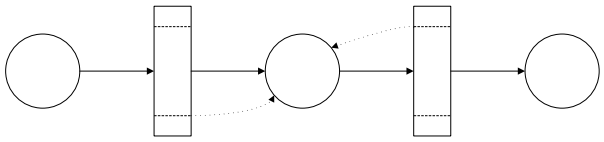


Figure 4.1: Signals generated by pipes

Montium Pipes

For the connection to the NoC there are eight fixed pipes available, four pipes in and four pipes out. The pipes are connected to a router through the bridge (see Section

2.5). It should be possible to connect an arbitrary pipe to the bridge/router. This is necessary for run-time remapping (see Section 4.2.5). The eight pipes connected to the routers are normal pipes, the size and thresholds can be defined.

For a pipe connected to the bridge it is necessary to define the type of communication. The driver needs to know how the bytes in the pipe have to be translated to packets for the NoC. A packet in the NoC exists of 16 data bits and 2 bits for the flit type. Such a packet is called a flit. A flit is a data unit that can be transported over one channel in one clock cycle. The flit type defines whether the flit is a configuration packet, data packet or the last data packet of the message (tail). Standard the data in the pipe is interpreted as 32 bits packets, the least significant 16 bits for the data and bit 16 and 17 for the flit type. The rest of the bits are discarded. There are three other ways of interpreting the data in the pipe: 16 bit, 32 bit horizontal dual and 32 bit vertical dual. For the 16 bit packets all bits are data. The flit type is the same for all data items, the driver adds the flit bits to the data. The flit type is standard set to data, but this is adjustable. If two pipes are dual, horizontal or vertical, both pipes are 16 bit and the data from both pipes is synchronically sent to the router. Just as for the 16 bit pipes the flit type for all data items is the same and added by the driver. The channels of the router are virtual mapped as an 2x2 matrix (see Figure 4.2). The horizontal option connects two horizontal neighboring pipes (pipe 1 and 2 or pipe 3 and 4). The vertical option connects two vertical neighboring pipes (pipe 1 and 3 or pipe 2 and 4).

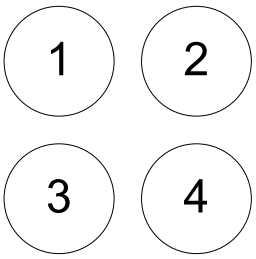


Figure 4.2: Channel layout of the bridge

USB Pipes

The USB pipes are fixed connected to the hardware. There are two pipes, one pipe in and one pipe out. For the USB pipes it is also preferable that an arbitrarily pipe could be connected to the USB driver. To get more USB pipes available there has to be a protocol defined on top of the USB connection to add information about the pipe.

4.2.4 Signals

Signals are used (and required) to trigger a process to set starting. They can be sent by another process or by a pipe. Signals can be stateful or stateless. A stateless signal signals once and then becomes low again. A stateful signal stays high as long as the signaling condition is valid (e.g. not full and not empty signals from pipes). Standard a process sends a stateless signal and a pipe a stateful signal. When a process receives

a signal a bit is set high for that signal. In this way a process remembers a trigger from a stateless signal. When bits of all signals of a process are set the process starts. When the process starts, all signal bits are cleared.

4.2.5 Run-time remapping

Run-time remapping is not implemented in the system so far. But the whole design of the system is such that run-time remapping is possible. The remapping itself can be started and executed in different ways. There has to be a trigger to remap. This can be a command, adding more applications to the system, etc. Moreover, there must be a manner to decide which processes are remapped and in which way. This can be an explicit command from a user to remap a process or an algorithm that decides what the best mapping is. What the trigger is and how the system decides what the best mapping is is out of the scope of this thesis. This thesis focuses on the steps required to supplant a process (remapping can be split out in objective supplanting processes).

A process consists of, next to the process or code itself, of a collection pipes in, pipes out, signals in and signals out. Because the processes are stateless, supplanting a process is in fact initiating another process with the same collection signals and pipes, remove the signals from the old process and change the scheduler (remove the old process and add the new). The old finishes its iteration (if it is running) and for the next iteration the new process starts. When the old process is running at the time of remapping it is required that one of the signals is stateless. Otherwise the new process also starts running (because the stateful signals are still high) and interferes with the old process.

This is not only a valid way for supplanting processes on the ARM, but also for processes on the Montium and the PC. But for supplanting a process from or to the Montium or PC in the current implementation, the original process has to be idle (and therefore no stateless signal is required). This is because the pipes from and to the process have to be changed, so the old process can not read and write the pipes anymore. Furthermore, the processes on the Montium and PC are data driven (they are not triggered by an signal but start executing when data is available). On the occasion of remapping a process to or from the Montium or PC the pipes where the previous process(es) reads from and the successor process(es) writes to are changed. This is done by changing the argument struct from these processes.

Before run-time mapping is implemented, it is preferred that the Montium and USB pipes are implemented as pipes which can be attached and detached to the bridge/router or USB driver. When there are multiple pipes from or to a process and there is data available in some of the pipes the process can not start its iteration because not all data is available yet. But when the pipes are completely replaced the data from the old pipes can not be read anymore, this data has to be copied into the new pipes.

Multiple iterations of the application can run in parallel (temporal parallelism). This is for example an advantage when a process on the Montium takes more calculation time than the previous and successor processes on the ARM. While the Montium is running, the next process can finish, but also the previous process can finish the next iteration. When the Montium finishes its calculation the data for the next iteration is already available, so the calculation can start directly. But, if multiple iterations of an

applications can run in parallel, supplanting a running process can the packets get out of order. For example, if the new process is faster and has therefore a shorter deadline (because the OS is *Earliest Deadline First, EDF* [18]). The new process is executed first. Therefore, an extra requirement is that the new process is not allowed to start before the old process finishes its last iteration). Also for processes on the ARM its easier to supplant them when they are not running.

For these reasons supplanting a process can be done best when the old process is idle. The new process has to be initiated with the right pipes and signals, the signals in have to be removed from the old process and the scheduler has to be changed (see Figure 4.3). Concerning the signals a bit more is required. For supplanting an ARM process with another ARM process the stateless signals which are already set in the old process has to be copied to the new process. If an ARM process is supplanted with a Montium or PC process, the signals from the previous process in the graph to this process triggering no process anymore (this is not a problem). But when a successor process is triggered by the process, this gives a problem. A Montium or a PC can not send signals. A developer has to keep this in mind while developing, for example by using only signals generated by the pipes.

4.2.6 Controller

All together there are a lot of options for the different parts of the system. Pipes have a size and thresholds and can trigger processes. Processes can be real-time or not and the triggers have to be connected to the process. The Montium and the routers have to be configured. This gives a lot of options during initiating of all the parts. And the initialization of the parts must happen in the right order. The Controller does all the initialization based on a configuration. This configuration is a character array (string), because constructing (for the developer) and parsing (in ANSI C) of a string is easy. This section describes which steps the Controller performs to store the configuration and start the application.

Configuration

The configuration is a string. All options described in the previous sections are supported by the Controller. The specification of the configuration, together with a description of the options for the different parts, can be found in appendix B. Appendix C shows an example of translating a process graph to a configuration.

Configuration structure

The Controller stores the configuration in a structure, the configuration structure. All processes, pipes, signals and routes are stored in this structure with their properties. In the first place this is necessary for the configuration itself. The Montium pipes are for example 32 bit during the configuration of the Montium, but after the configuration they are configured to what is specified (32 bit, 16 bit or 32 bit dual). Therefore, not all parts of the objects can be configured at the same time and the configuration has to be stored. In the second place the total configuration is required in case of remapping. Only then the pointers to the processes, pipes, etc. can be found. These pointers are used to alter the properties and to change pointers in the argument struct.

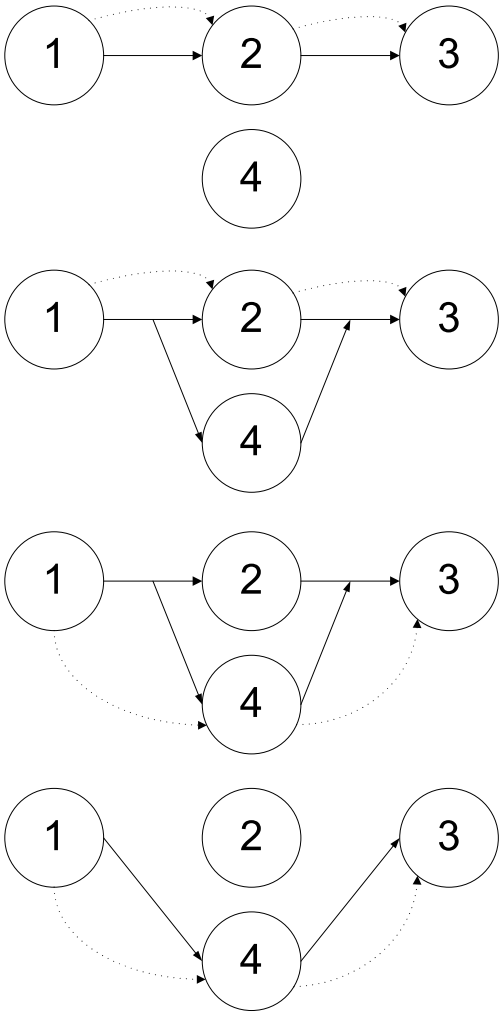


Figure 4.3: Run-time supplant a process

Every process on the ARM gets an (pointer to an) argument structure as argument. In this argument structure the pointers to the pipes in, pipes out and the signals out for that process are placed. The layout of the configuration structure and the argument structure can be found in appendix D.

The CONTROLLER performs the following steps (in this order):

1. Store routes
   The route information is stored in the configuration structure. The routers itself are configured after configuring the processes, because the configuration of the MONTIUMS requires a direct route from the ARM.

2. Store and initiate pipes
   The information about the pipes is stored in the configuration structure and the pipes are initiated (with all properties like size, thresholds and triggers).

3. Store and initiate MONTIUM and USB pipes
   The information about the MONTIUM and USB pipes is stored and these pipes are also initiated. The MONTIUM pipes are initiated as 32 bit pipes because they have to be used for the configuration. Later on they are reconfigured.
4. Store and initiate timers
   The timers are stored and initiated.
5. Store and initiate signals
   The signals are stored and initiated (to zero, non-signaling).
6. Store processes
   The information of the processes is stored. Also the pointers to pipes and signals are retrieved here and stored in the configuration structure.
7. Initiate processes
   The processes are initiated. ARM processes are created, flags are set, the argument structure initiated and the triggers are added to the processes. But the ARM processes can not start yet, because they are not added to the scheduler. For the processes on the MONTIUM the MONTIUMS are configured. The routers are temporarily configured from the ARM to the particular MONTIUM (routes are removed after the configuration). The MONTIUM processes can not start either before adding the ARM processes to the scheduler because the only way to receive data for a MONTIUM process is by an ARM process.
8. Configure routers
   After the MONTIUMS are configured the routers can be configured to the actual configuration.
9. Reconfigure MONTIUM pipes
   After the MONTIUMS are configured it is also allowed to configure the MONTIUM pipes to the right type (32 bit, 16 bit, etc.).
10. Add ARM processes to the scheduler
    Finally, the ARM processes are added to the scheduler so the processes (and the application) can start.

4.3 Usage of the Controller

This section gives a brief introduction to the usage of the CONTROLLER and implementing an application on the BCVP using BASOS and the CONTROLLER. Starting point of the implementation is a process graph with processes, pipes and signals. The implementation of the processes itself is also required and therefore the spatial mapping is required.

4.3.1 Layout of the application file

The application requires a file which contains all the processes and configuration data (eventually using includes). To use the CONTROLLER the file which contains the controller has to be included. The total layout of the file is in appendix E.

At the top of the file are the includes. These are standard libraries, libraries required for BasOS and the configuration file. Below the includes are the defines. For using Montiums or the USB connection defines are required: if this hardware is not used, the driver (causing overhead) is not loaded. The following includes are required for using Montiums or the USB:

```
#define USE_MONTIUM0 //Define to use Montium 0
#define USE_MONTIUM1 //Define to use Montium 1
#define USE_MONTIUM2 //Define to use Montium 2
#define USE_USB //Define to use the USB connection
```

When a Montium is used a configuration is required. This configuration has to be in an integer array with a pre-defined name. For Montium0 this is "montium0_cfg", for Montium1 "montium1_cfg" etc. Also the length of the array has to be defined: "montium0_cfg_size".

Next, the specification is required. This specification describes the processes, signals and the connection between the processes. Below the specification are the processes running on the specified ARM. Because it is a real-time system, there is for every process a configuration struct. This configuration structure contains also a name (as string). This name is used by the Controller to identify a process defined in the specification. Therefore, there has to be an array with these structure to search through. This array is placed under the the processes in the file. Finally, a predefined standard process is required which starts automatically and initiates the Controller to start. This process also starts the Montium and USB drivers because these drivers have to be started in kernel space [18].

4.3.2 Specification

The specification describes the process graph. For each object in the process graph there is a line in the specification. The formal definition of the specification can be found in appendix B. The configuration starts with a line specifying a name for the specification. This is used to identify the specification. This could be useful later on when multiple specifications are used. Next the lines that specify the configuration of the routers are placed. After that the more regular object specified (pipes, signals and processes) with their properties are given. There is one special type of signals that is not mentioned before, that is a timer. A timer is a signal that is fired periodically. The period can be defined in the specification. Appendix C shows an example of translating a process graph to a configuration.

4.3.3 Processes

The processes can run on three different processors, the PC, the ARM or the Montium. Processes running on the PC can communicate via the USB connection. They can use "libusb" for this. For more information see [18]. The processes on the ARM are processes running on BasOS and the processes running on the Montium require a Montium configuration.

ARM Processes

Processes on BasOS contains code for the process itself and a structure with the real-time specifications. The code for the process itself is ANSI C code and has the form of a function (but it are really processes). The processes can use functions provided by BasOS and libraries suitable for the compiler and processor architecture. [18]

For reading and writing the pipes four functions are available, blocking and non-blocking reads and writes. Another frequently used function is sending signals. These five functions are described below:

```
void SYS_write             (pipe *p, char* data, int data_size)
int  SYS_write_nb          (pipe *p, char* data, int data_size)
void SYS_read              (pipe *p, char* data, int data_size)
int  SYS_write_nb          (pipe *p, char* data, int data_size)
void SYS_event_send_signal (signal *s);
```

The functions with "_nb" added are the non-blocking functions. These functions return the number of bytes actual written. The pointers to the pipes and the signals are provided by the Controller through the argument structure.

Montium Processes

As mentioned before, the configuration for the Montiums is stored in integer arrays with pre-defined names. The configuration consists of data items of 32 bit. The least-significant 16 bits are the data bits and bit 16 and 17 defines the flit type.

4.4 Conclusion

BasOS provides drivers for streaming communication between processes running on the ARM, Montium and the PC. After initialization, the communication interface between two processes is independent of on which processor the processes run. Processes are defined in such a way that run-time remapping is possible. To make the usage of the system easier, a Controller is implemented. This Controller initiates the communication and configures the Montiums.

CHAPTER 5

Results

The 4S MPEG-4 is suitable for Osyres on top of eCos [22,23]. eCos is an open source, real-time OS intended for embedded applications. Osyres is a middleware layer between the OS and the application for inter-process communication. Osyres handles the communication between processes on the ARM and between processes on the ARM and the PC (using USB). It is not yet possible to handle communication between the ARM and the Montiums with Osyres. [1]

In this chapter the timing results of the different implementations are presented and compared. The first section describes the method; What is compared and in which way. In the second section are the results presented.

5.1 Method

The 4S implementation is adjusted to run on BasOS. As little as possible has been adjusted, only data sending and receiving is changed. Therefore, it is possible to compare this implementation with the Osyres implementation. For the implementation on BasOS, the Controller is used to validate whether the Controller works properly. The CS was not implemented in the 4S implementation, the implementation for this algorithm has been added to compare the different implementations. The 4S implementation using Osyres can also run on the PC. Results of the 4S implementation running on the PC are also presented.

The iQT and CS are implemented on a Montium as described in Chapter 3. The Controller initiates the communication between the ARM and the Montiums. The other processes stay exactly the same, regardless whether the iQT and CS processes are running on the ARM or on a Montium. If the iQT and/or CS are running on a Montium, the iQT runs on Montium0 and the CS runs on Montium1.

The results show the difference between the implementation running on BasOS, the Osyres implementation running on the ARM and the Osyres implementation running on the PC. For the timing on the PC all user tasks are canceled and the timing

functions of C are used. The used PC is a Pentium 4 running at a clock speed of 2.60 GHz. For the timing on the ARM (both for Osyres and BasOS) ARM1 of the BCVP is used. This is the ARM946E running at a clock speed of 86.016 MHz. For the measurements the clock count of the slow clock of the ARM is used. This clock runs at 32.75 kHz. The Montium timing is done by calculating the total number of clock cycles. The Montium runs at 6.6 MHz.

The BCVP is connected with an USB and a serial connection to a PC. The USB connection is used for the data transport to and from the BCVP. The serial connection is used to load the configuration into the BCVP, start the application and as standard out.

For all three implementations the total decoding time for a frame and the calculation times without communication per process are measured. The calculation times without the communication are for the ARM and PC based on the same function call, so the same piece of C code is executed. For the Montium implementations the time to execute a total process is calculated, because the communication and execution are done simultaniously (e.g. stream in the samples and the iQT algorithm is done at once).

The 4S implementation is based on Osyres and therefore optimized for Osyres. In Osyres a message variable is created and the data is copied to this message, sample by sample. The pointer to this message is send to the next process. The next process copies the data from the message into its own variables. On BasOS the data is copied into the pipe (and copied out the pipe), not only the pointer is sent. So, in the current MPEG-4 implementation first all data is copied into a temporal variable sample after sample, just like copying the data to the message in Osyres. Then the data is copied into the pipe in one burst.

In Section 5.2.3 the differences between BasOS implementations are presented. There are 4 different implementations: run all processes on the ARM, run the iQT on a Montium, run the CS on a Montium and run the iQT and the CS on a Montium. We determine whether the implementations using the Montium are faster and how much overhead the communication with a Montium causes.

Also the results of possible optimizations for MPEG-4 implementations are presented. For example connect the pipe from Montium1 (the CS) directly to the USB pipe.

Finally, in Section 5.2.5 the optimizations in energy performance using a Montium are estimated.

5.2 Results

During the implementation of MPEG-4 on BasOS the Controller worked well. All options of the Controller are used in the implementation (e.g. all different options for the Montium pipes).

5.2.1 Implementations on different processors

Figure 5.1 shows the total calculation time per process on the different processors and implementations. As mentioned before, this is for the ARM and PC without

communication. For the MONTIUM these times include the communication because this can be overlapped with other processes.

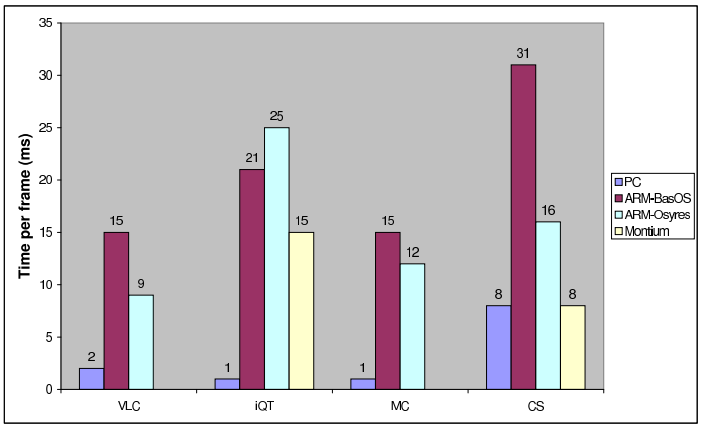


Figure 5.1: Calculation time per process per frame

The figure shows that the PC executes all algorithms the fastest. This is because the PC runs on the highest clock frequency.

Another notable fact is the difference between Osyres and BASOS, while the same C code is used. For some processes eCos is faster, for other processes BASOS is faster. This can be caused by differences in memory management, OS overhead and compiler options.

Figure 5.2 shows the total execution time for the different implementations.

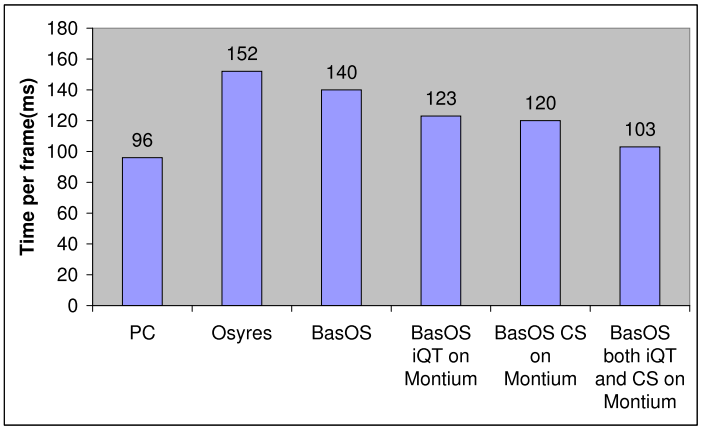


Figure 5.2: Total execution time per frame

The PC is overall also the fastest, but not ten times as in Figure 5.1. This could be caused by high overhead due to communication in windows. The difference between the Osyres and the BASOS ARM implementation is notable. When the calculation times of the processes are added, the sum is lower for Osyres than for BASOS. But the

total execution time per frame is lower for BasOS. The next subsection describes the difference between the two ARM implementations in more detail.

Outsourcing of the iQT and/or the CS to a Montium obviously gives a better performance. Per outsourced process the processing time per frame decreases with almost 14%. Section 5.2.3 describes the gain on performance and overhead in more detail.

5.2.2 Differences between ARM implementations

Figure 5.3 shows a decomposition of the total execution time per frame for the ARM implementations into calculation time, communication time and rest.

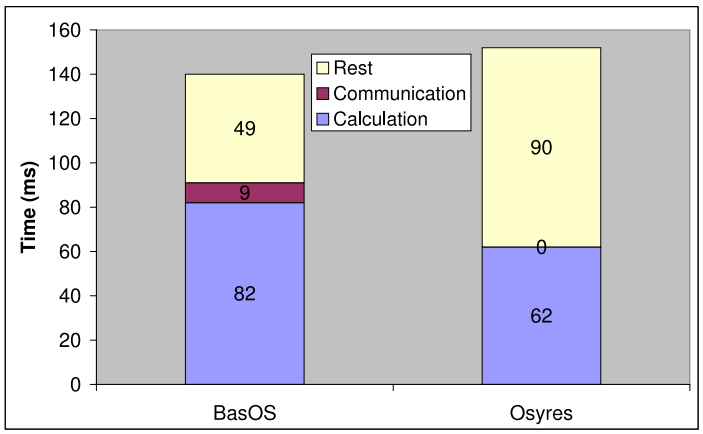


Figure 5.3: Execution time of ARM implementations split in calculation, communication and rest

For the Osyres implementation no communication time is specified, because we can not measure the communication time in Osyres. For BasOS there is a clear difference between the (for BasOS unnecessary) copying of data and the communication. Furthermore, we were able to measure the time required to read from and write to a pipe. Therefore, the rest time for Osyres consists of the communication and the OS overhead. For BasOS the rest time consists of copying data into the temporal variable and the OS overhead.

The figures shows that the overhead for Osyres is much bigger. The total time without the calculation time is for Osyres 90 ms and for BasOS 58 ms, while the implementation is based on Osyres. Especially copying the data one value at a time requires much execution time in BasOS, while copying all data at once can be done much faster. If the data is stored in another way we expect a much smaller execution time for BasOS: the data can be copied at once from the pipe before the calculation and to the pipe after the calculations. This requires in total 9 ms. There are another 82 ms required for the calculation and there is overhead for the OS. Measurements showed that the overhead for BasOS is smaller than 2%. This gives an estimated total execution time of $(82+9)\times1.02 = 93$ ms. Furthermore, more optimizations are possible in memory management (e.g. copying data samples one by one) or compilation to

decrease the calculation time even more for BasOS. One could expect that the total calculation time for BasOS could also decrease to about 62 ms, because exactly the same processor architecture is used.

Reading from a pipe and writing to a pipe is equally fast. Figure 5.4 shows that the time required to read from or write to a pipe seems to be linear. For three different amounts of data it is measured how much time is required to read from or write to the pipe. The read and write times are exactly the same (an equally number of slow clock cycles). Note that the measurement error of all measurements based on the slow clock is 1 clock cycle, 0.03 ms. For the smallest amount of data (12672 bytes), this is 0.03/0.3 = 10%.The formula for the time it takes to read or write an particular amount of data is:

$$T = 0.00001 \times B + 0,1638$$

,where T is the time in ms and B the amount of data in bytes.

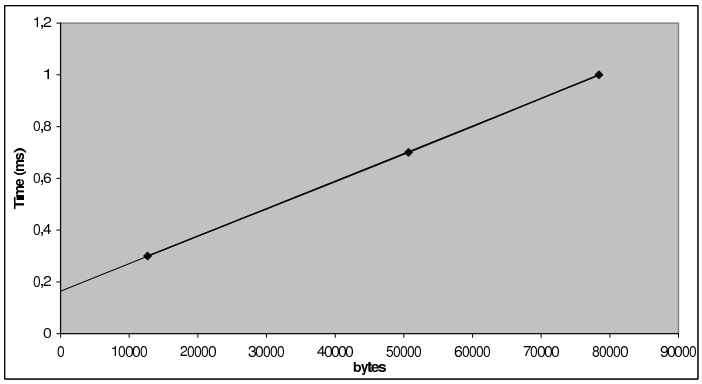


Figure 5.4: Time to read or write a pipe

5.2.3 Advantage and overhead using the Montium

Outsourcing processes to a Montium gives a reasonable reduction in the processing time. Table 5.1 shows the total execution times, the reduction of execution time and the time the outsourced processes would require on the ARM. The time the processes require on the ARM is the calculation time. It excludes the time required for the copying. The communication time stays the same because instead of the ARM process, the driver reads the data from the pipe and copies the data to the Montium.

The differences in execution time are smaller than the time required for executing the outsourced process on the ARM. This is caused by the overhead of communication with the Montium. Communication with the Montium on the BCVP is very slow: before the ARM is ready with sending data to the Montium, the Montium is already sending data back (the Montium is data driven, it starts executing directly after it receives the first data). The expectation is that in the next generation BCVP, the *Highly Integrated Concept Verification Platform (HiCVP)*, the communication with the Montium is much faster. This should give lower overhead costs and therefore even more advantage of outsourcing processes to a Montium.

	Total (ms)	Reduction (ms)	Required on ARM (ms)
Only ARM	140		
iDCT on a MONTIUM	123	17	21
CS on a MONTIUM	120	20	31
iDCT and CS on a MONTIUM	103	37	52

Table 5.1: Total execution time per frame, the decrease of execution time and the time the process(es) outsourced to a MONTIUM require on the arm

5.2.4 Optimizations for the BasOS implementation

We tried a couple of possible optimizations in the implementation to get a better overall performance: connect the pipe from the MONTIUM (executing the CS) directly to the USB, change a part of the data copying in the VLC and MC, use other compiler options and write directly in the pipe instead of copying it to a temporal variable first. The overall performance is calculated by measuring how long it took to decode a movie of 201 frames, including the USB communication. So, the measurement starts when the PC starts sending the first frame and stops when the last frame was received. The next frame is send when the previous is received. Table 5.2 shows the results of these optimizations. Connecting the MONTIUM pipe directly to the USB pipe gives a

	Frames per second
No optimizations	5.2
Connect MONTIUM pipe directly to USB pipe	stops half way, but slower
Optimize data copying	5.6
Optimize data copying and use more specific, optimized compiler options	5.6
Optimize data copying and write directly in pipe instead of into a temporial variable first	5.0

Table 5.2: Total execution time per frame, the decrease of execution time and the time the process would require on the arm

lower performance. If an ARM process is used to copy the data, this process waits until all data from the MONTIUM is available and copies it to a variable and then to the USB. The stop half way is probably caused by a pipe that is under its threshold, so the driver is not triggered anymore.

The changes to memory copying gives a higher performance. Instead of copying all data value by value, the data is copied in as big bursts as possible. Loops are unrolled to be able to copy the data at once. This gives a remarkable better performance. Copying the data directly into the pipe instead of to a variable first gives a longer execution time, while the same clustered way of copying is used. This is because of the overhead of writing to a pipe, this overhead is independent of the number of bytes written to the pipe. Using another compiler does not result in changes in performance.

5.2.5 Energy performance

In this subsection we estimate if the energy usage of the total system decreases when processes are outsourced to the Montium. Therefore, we make an estimation of the energy usage of the ARM and the Montium decoding a frame. Next, we compare the energy usage performing one 2D 8x8 iDCT and the maximum number of iDCTs per second per mm^2 on different processors architectures: an ASIC, a Montium, a DSP and a GPP.

Energy consumption per frame

When processes are outsourced to a Montium, the ARM finishes processing faster and therefore uses less energy. But, the Montium uses also energy and the data has to be transported to the Montium.

The power consumption of the ARM946 is 0.46 mW/Mhz [24]. The energy consumption per clock cycle is $\frac{0.46 \cdot 10^{-3}}{1 \cdot 10^{6}} = 0.46\ nJ$. The total energy usage decoding a frame completely on the ARM946 is $140 \cdot 10^{-3} \times 86 \cdot 10^{6} \times 0.46 \cdot 10^{-9} = 5.5\ mJ$. In the same way is the energy usage of the ARM calculated when the iQT is outsourced (4.9 mJ), when the CS is outsourced (4.7 mJ) and when both the iQT and the CS are outsourced (4.1 mJ).

The power consumption of the Montium is estimated on the basis of tests from [16]. We assume that clock gating is possible. Therefore, we assume the power consumption of the Montium when it is idling can be neglected. The power consumption of the Montium performing an iQT is estimated to be as much as the Montium uses during the FFT64 [16]. This algorithm has the same complexity and uses the same number of memories. Therefore, the energy consumption of the Montium performing an iDCT is 0.541 mW/MHz. Because the Montium is running on 6.6 MHz and it takes 15 ms to calculate the iQT, the total energy usage for the iQT per frame is $15 \cdot 10^{-3} \times 6.6 \times 0.541 \cdot 10^{-3} = 54\ \mu J$.

The power consumption of the Montium performing the CS is estimated to be as much as the Montium uses for the FIR5 [16]. This algorithm uses also two memories and has the same complexity or is even more complex. Therefore, the energy consumption of the Montium performing the CS is estimated to be 0.374 mW/MHz. The total energy usage of the Montium performing the CS is $8 \cdot 10^{-3} \times 6.6 \times .374 \cdot 10^{-3} = 20\ \mu J$ per frame.

Table 5.3 shows the total energy consumption of the processors per frame. It shows that the power consumption of the Montium is neglectable concerning the power consumption of the ARM (factor 100). The energy usage excludes the CCU of the Montiums, the memory access of the ARM and the communication from the ARM to the Montium. There are no numbers about the power consumption of the CCU available yet. We assume that the energy usage of the CCU is neglectable concerning the energy consumption of the ARM because the energy consumption of the whole Montium is also neglectable.

The energy numbers do not take the memory accesses of the ARM nor the communication between the ARM and the Montium into account. When a process is outsourced to a Montium, this decreases the number of memory accesses of the ARM (because not all intermediate values fit into the cache). The energy usage is increased due the communication with the Montium. Because the same amount of

	ARM (J)	Montium0 (J)	Montium1 (J)	**TOTAL(J)**
Only ARM	5.5 m	0	0	5.5 m
iDCT on Montium0	4.9 m	54 μ	0	5.0 m
CS on Montium1	4.7 m	0	20 μ	4.7 m
iDCT and CS on a Montium	4.1 m	50 μ	20 μ	4.2 m

Table 5.3: Total energy usage per frame

data is copied in both cases, we estimate that these two numbers are approximately equal: communication with the FPGA is done through a shared memory. The amount of data that has to be transported to the Montium is the same as the ARM needs for calculation. So, the same amount of data is transported over the AMBA bus. The energy usage of the NoC is not known. No energy consumption tests are done yet. We estimate that the transport of the data to and from the Montium requires not more energy than the calculation on the Montium itself. Therefore, the energy usage of the NoC is at most a few percent of the energy usage of the ARM.

Energy consumption per iDCT

To benchmark the energy usage and the processing power of the Montium, we compared the Montium with three other processors: an ARM, a DSP and an ASIC. These architecture are alternatives for the Montium. The ARM can do the MPEG-4 decoding also itself, so what is the gain of combining it with a Montium? The DSP is one of the most likely alternatives for a reconfigurable architecture. It is more flexible, but also faster and more energy hungry? The ASIC is the best choice in terms of energy-efficiency and processing power, but is not flexible nor reusable for other algorithms.

We compare the required energy and processing time performing an 2D 8x8 iDCT (and therefore the number of iDCTs per second). The energy usage for communication is for all processors not taken into account. The processing power is normalized by chip area (number of iDCT per second per mm^2). It is evident that doubling the chip area will increase the processing power (for example using two Montiums instead of one).

The approach is as follows. First the number of clock cycles required to perform an 2D 8x8 iDCT are determined. This is multiplied by the energy usage of one clock cycle. To make an fair comparison, the power characteristics for each processor are normalized to 0.13 μm technololgy and a nominal voltage of 1.2V. To benchmark the processing power, the number of iDCTs a processor can perform per second is determined (also normalized to 0.13 μm technology). This is divided by the area in mm^2.

Table 5.4 depicts the results of this comparison together with the characteristics of the processors. Figure 5.5 shows the same results in bar graphs. The energy usage is as expected. The GPP processor (ARM) uses the most energy, the ASIC uses the least energy. Notable is that the Montium behaves concerning energy usage much more like an ASIC than an DSP.

The ASIC can perform the most iDCTs per second per mm^2. The difference

between the ASIC and the Montium here is much larger than for the energy usage. The ARM is much slower than the rest of the processors, it can perform only 1/12 of the number iDCTs compared to a Montium and 1/80 compared to the DSP. But, because it has a very small area (1/40 compared to the DSP), the number of iDCTs per second per mm^2 differ only a factor 2 with the DSP.

The ASIC is, as espected, the best choice in terms of energy-efficiency and processing power. For energy usage, the Montium behaves like an ASIC. But, concerning processing power, the ASIC is four times faster than the Montium. The Montium is still much faster and more energy-efficient as a DSP and ARM. For performing an iDCT in MPEG-4 decoding in an heterogeneous architecture a (couple of) Montiums is the best choice. It is much more energy-efficient than an ARM (and DSP) and outsourcing the algorithm leads to both more processing power and less energy usage. An ASIC is even more energy-efficient and has more processing power, but especially for mobile devices that perform different algorithms the ASIC is too inflexible. The difference in energy usage is not that big compared to the energy usage of an ARM and a (couple of) Montium supply enough processing power to perform the iDCTs in MPEG-4.

In the remainder of this section we explain the results.

	ASIC	Montium	TI	ARM
Max. frequency (MHz)	154	100	720	200
Power @ max. frequency (mW)	635	50	1967	92
Technology (μm)	0.18	0.13	0.09	0.13
Nominal voltage (V)	1.8	1.2	1.2	1.2
Area (mm^2)	12.71	2.4	n/a	1.86
Cycles/2D 8x8 iDCT	30	112	82.5	2796
Normalized energy/clock cycle (mW)	1.34	0.541	3.95	0.46
Normalized area (mm^2)	6.1	2.4	≈79	1.86
Normalized max. frequency (MHz)	216	100	514	200
# 2D 8x8 iDCTs / second	7.2 M	0.9 M	6.2 M	0.072 M
Energy/2D 8x8 iDCT (nJ)	**40**	**61**	**325**	**1286**
# 2D 8x8 iDCTs/s/mm^2	1182 k	372 k	≈ 79 k	38 k

Table 5.4: Characteristics and benchmarking of 2D 8x8 iDCT in terms of energy and area on four different processor architectures

Montium

The Montium needs 112 clock cycles for one 2D iDCT (see Table 3.1). The energy consumption of a Montium performing an iDCT is 0.541 mW/MHz (see previous section). This is for a Montium in 0.13 μm 1.2V technologie [16]. Therefore, the energy consumption per clock cycle is $\frac{0.541 \cdot 10^{-3}}{1 \cdot 10^6} = 0.541\ nJ$. The total energy consumption performing an 2D iDCT is $112 \times 0.541 = 61\ nJ$.

The Montium can run on 100 MHz. Therefore, the Montium can execute $\frac{100 \cdot 10^6}{112} = 0.893\ M$ 2D iDCTs per second. In 0.13 μm technology, the Montium is 2.4 mm^2 [16]. Dividing by the area gives $\frac{0.893 \cdot 10^6}{2.4} = 372\ k$ 2D 8x8 iDCTS per second per mm^2.

ARM

We implemented the Chen algorithm also on the ARM946 using the DSP function (i.e. single cycle MAC). These DSP functions are not supported by the GCC compiler, so we added this assembly codes by hand (in C, letting the compiler handle the register mapping). The ARM needs 2796 clock cycles to perform one 2D 8x8 iDCT. Examining the assembly codes reveals that half of the clock cycles are used to load and store registers. There are only four registers available for calculation.

The energy consumption of the ARM946 is 0.46 mW/Mhz in 0.13 μm technology [24]. The energy consumption per clock cycle is $\frac{0.46 \cdot 10^{-3}}{1 \cdot 10^6} = 0.46\ nJ$. Therefore, the energy required for one 2D 8x8 iDCT is $2796 \times 0.46 = 1286\ nJ$. The maximum clock frequency of the ARM946 in 0.13 μm technology is 200MHz [24]. Therefore, the number of iDCTs per second is $\frac{200 \cdot 10^6}{2796} = 71530$. Because the ARM946E is 1.86 mm^2 [24], the number of iDCTs per second per mm^2 is $\frac{71530}{1.86} = 38\ k$.

ASIC

As ASIC implementation we examined the implementation presented in [25]. The used algorithm is the same as our implementation. This ASIC is implemented in 0.18 μm TSMC technology width an nominal voltage of 1.8V. It is possible to estimate the energy consumption as described in [26] in a smaller technology and a lower nominal voltage. The energy consumption of the ASIC is 634.5 mW per 156 MHz. Therefore, the estimated energy consumption per clock cycle in 0.13 μm technology is $(\frac{1.2}{1.8})^2 \times \frac{0.13}{0.18} \times \frac{634.5 \cdot 10^{-3}}{154 \cdot 10^6} = 1.32\ nJ$. The ASIC needs 30 clock cycles per 2D 8x8 iDCT [25], so the total energy consumption is $30 \times 1.32 = 40\ nJ$ per iDCT.

The area of the ASIC is 12.17 mm^2 in 0.18 μm technology [25]. The gate density in TSMC technology is 100K gates per mm^2 for 0.18 μm technology and 200K gates per mm^2 for 0.13μm [27] [28]. Therefore, the area is halved in 0.13 μm and becomes $\frac{200}{100} \times 12.17 = 6.09\ mm^2$. The clock frequency increases in newer technologies. According to the ITRS roadmap [29] the maximum clock frequency scale width a factor 1.4 per technology generation. Therefore, the maximum clock frequency will be around $154 \times 1.4 = 216$ MHz. The number of iDCTs per second is $\frac{216 \cdot 10^6}{30} = 7.2\ M$. This results in $\frac{7.2 \cdot 10^6}{6.09} = 1182\ k$ 2D 8x8 iDCTs per second per mm^2.

DSP

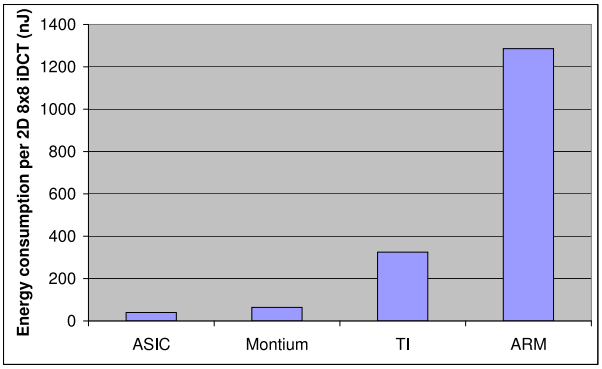
As DSP we have choossen for the state-of-the-art TMS320C6454-720 for comparison. According to [30] the power consumption is 1.18W at 1.2V width 60% utilization and a clock frequency of 720 MHz. We assume that the power consumption is linear depending on the utilization [16], so we expect $\frac{100}{60} \times 1.18 = 1.97$ W at 100% utilization. This DSP is implemented in 0.09 μm technology. It is possible to estimate the energy consumption as described in [26] in a larger technology. Therefore, the power consumption per clock cycle is $(\frac{1.2}{1.2})^2 \times \frac{0.13}{0.09} \times \frac{1.97}{720 \cdot 10^6} = 3.95\ nJ$.

TI provides a library with imaging functions. This library contains an 2D 8x8 iDCT function [31]. This function requires $72 \times n + 63$ clock cycles, where n is the number of iDCTs. We assume that it is realistic to execute 6 iDCTs, because there are 6 blocks per MB. So, on average $\frac{72 \times 6 + 63}{6} = 82.5$ clock cycles per 2D 8x8 iDCT are used. The energy consumption is $3.95 \times 82.5 = 326\ nJ$ per iDCT in 0.13 μm technology.

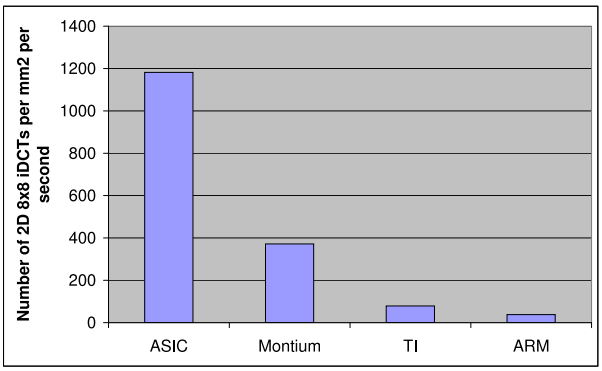
The area of the TMS320C6454-720 is not disclosed. Therefore, we estimate the the area. At the International Electron Devices Meeting in fall 1999, TI presented a roadmap [32], which reveals that a TMS320 core will contain about 100M transistors in 2005. Therefore, we assume that the TMS320C6454-720 has about 100M transistors. To estimate the number of transistor per mm^2, we investigated the the 0.13 μm TSMC technology [33]. For SRAM the density is 2.43 - 2.14 μm^2 for 6 transistors. This means there are 6 transistors per 2.34 μm^2 and therefore $\frac{6}{2.43 \cdot 10^{-6}} = 2.46\ M$ transistors per mm^2 (for SRAM memory). The gate density is 219k gates/mm^2, where a gate consists of 4 transistors. Therefore, there are $4 \times 218 \cdot 10^3 = 0.88\ M$ transistors per mm^2.

The DSP contains 8 Mbit SRAM. One bit SRAM consists of 6 transistors. Therefore, the memory on the DSP uses $6 \times 8 = 48\ M$ transistors. This requires $\frac{48 \cdot 10^6}{2.46 \cdot 10^6} = 20\ mm^2$. Because we estimated the TMS320C6454-720 contains 100M transistors, this leaves 52K transistors. This remaining transistors require $\frac{52 \cdot 10^6}{0.88 \cdot 10^6} = 59\ mm^2$. The total estimated area is 79 mm^2.

Normalization of the clock frequency gives a maximum frequency of $\frac{720}{1.4} = 514$ MHz. Because the iDCT requires 82.5 clock cycles per iDCT, the TMS320C6454-720 can execute $\frac{514 \cdot 10^6}{82.5} = 6.23\ M$ 2D 8x8 iDCTs per second in 0.13 μm technologie. With a area of 79 mm^2 this are $\frac{6.23 \cdot 10^6}{79} = 79\ k$ iDCTs per second per mm^2.



(a) Energy consumption for a 2D 8x8 iDCT on different processor architectures, normalized to 0.13μm technology and 1.2V



(b) Number of 2D 8x8 iDCTS per second per mm^2 for different processor architectures

Figure 5.5: Benchmarks of a 2D 8x8 iDCT on different processor architectures

CHAPTER 6

Conclusions and Recommendations

This chapter summarizes the most important conclusions of this thesis and gives some recommendations for future work.

6.1 Conclusions

1. It is possible to perform MPEG-4 processes on reconfigurable hardware. Not all tasks can be performed efficiently, the VLC process can be executed best on a GPP. The iQT and CS process can be executed on a Montium efficiently. The MC can be executed on an FPGA.

2. The communication bandwidth is large enough to perform processes on different processors. The communication overhead for the ARM is smaller than the reduction of required processing time caused by outsourcing processes. In the next generation platform, the HiCVP, the communication between the ARM and the Montiums is faster. Therefore, the communication overhead will be even smaller.

3. An architecture consisting of an ARM and a Montium increases performance in both processing power and energy-efficiency for MPEG-4.

4. The Montium is, performing an 2D 8x8 iDCT, much more energy-efficient than a TI DSP and an ARM. An ASIC is more energy-efficient, but the Montium behaves for energy-efficiency more like an ASIC than like a DSP.

5. An ASIC is, normalized to area, significantly faster than a Montium performing an 2D 8x8 iDCT. The Montium is much faster than a DSP and ARM.

6. The Montium is a good choice for a heterogeneous architecture combined with an GPP. It is for calculation intensive algorithms much faster and more energy-efficient than an ARM and DSP and more flexible than an ASIC.

7. A MONTIUM is a good solution for computational intensive tasks like iDCTs and CS, but it has also its limitations. Especially the limited amount of memory (MC) and the limited (and fixed) number of configurations (iDCT transpose).

8. The communication between processes running on the ARM, the MONTIUM and the PC works fine. Initializing the communication and configure the NoC and MONTIUM is much easier when the CONTROLLER is used.

6.2 Recommendations

6.2.1 Change MPEG-4 implementation

To get a faster demo the MPEG-4 implementation has to be changed:

1. The data storage and communication have to be changed to optimize the communication for BASOS and get rid of the double copying of data.

2. Processes are frame based, they perform their operation on a whole frame. When processes are block (or Macroblock) based, the buffers in the communication between the processes and the required memory for the processes itself decrease.

3. Processes on the ARM are already data driven, but the first process starts when the last process is finished. When back pressure is added (a process starts only when there is enough free buffer memory to write its results), the first process can start before the last process is finished when there is enough buffer memory and free CPU time.

6.2.2 Implement run-time remapping

The implementation of the communication between processes in BASOS is almost suitable to support run-time remapping. Only a few changes have to be made:

1. Change the MONTIUM and USB pipe implementation, so an arbitrary pipe can be attached or deattached to these devices.

2. Implement the run-time remapping as described in 4.2.5.

Bibliography

[1] 4S report D1.1. Ambient systems requirements (update march 2005). Internal project report, 2005.

[2] The mpeg home page. `http://www.chiariglione.org/mpeg/`.

[3] Koenen R. (2002). Overview of the mpeg-4 standard. `http://www.chiariglione.org/mpeg/standards/mpeg-4/mpeg-4.htm`.

[4] B.G. Haskell, A. Puri, and A.N. Netravali. *Digital Video: an Introduction to MPEG-2*, chapter 6-10. Chapman & Hall, 1997. ISBN: 0-412-08411-2.

[5] B.G. Haskell, A. Puri, and A.N. Netravali. *1997*, chapter 5. Chapman & Hall, Digital Video: an Introduction to MPEG-2. ISBN: 0-412-08411-2.

[6] J. Kneip, S. Bauer, J. Vollmer, B. Schmale, P. Kuhn, and M. Reimann. The mpeg-4 video coding standard - a vlsi point of view. In *IEEE Workshop on Signal Processing Systems*, pages 43–52, 1998.

[7] K.N. Ngan, T. Meier, and D. Chai. *Advanced Video Coding: Principles and Techniques*, chapter 1. Elsevier, 1999. ISNB: 0-444-82667.

[8] M. Berekovic, H.J. Stolberg, and P. Pirswch. Implementing the mpeg-4 as profile for streaming video on a soc multimedia processor. In *3rd Workshop on Media and Streaming Processors*, 2001.

[9] B.G. Haskell, A. Puri, and A.N. Netravali. *1997*, chapter 17. Chapman & Hall, Digital Video: an Introduction to MPEG-2. ISBN: 0-412-08411-2.

[10] K.N. Ngan, T. Meier, and D. Chai. *Advanced Video Coding: Principles and Techniques*, chapter 6. Elsevier, 1999. ISNB: 0-444-82667.

[11] V. Bakker and M. Schoemaker. Mpeg decoding on multiple architectures. *Embedded Computer Architecture University Twente*, 2006.

[12] D. Ishii, M. Ikekawa, and I. Kuroda. Parallel variable length decoding with inverse quantization for software mepg-2 decoders. In *IEEE Workshop on Signal Processing Systems*, pages 500–509, 1997.

[13] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh. A quick safari through the reconfiguration jungle. In *Design Automation Conference*, pages 172–177, 2001.

[14] G.J.M. Smit, A.B.J. Kokkeler, P.T. Wolkotte, M.D. van de Burgwal, and P.M. Heysters. Efficient architectures for streaming dsp application. In *Dagstuhl Seminar Proceedings*, 2006.

[15] L. Guerra, M. Potkonjak, and J. Rabaey. System-level design guidance using algorithm properties. In *Proc. of the VLSI Signal Processing Workshop*, pages 73–82, 1994.

[16] P.M. Heysters. *Coarse-Grained Reconfigurable Processors: Flexibility Meets Efficiency*. PhD thesis, University of Twente, 2004. ISBN: 90-365-2076-2.

[17] R. Hartenstein. A decade of reconfigurable computing: a visionary retrospective. In *Design, Automation, and Test in Europe*, pages 642–649, 2001.

[18] B.A. van Sisseren. Design of a lightweight real-time streaming kernel. Master's thesis, University of Twente, 2007, to appear.

[19] M.D. van de Burgwal, G.J.M. Smit, G.K. Rauwerda, and P.M. Heysters. Hydra: an energy-efficient and reconfigurable network interface. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 171–177, 2006.

[20] C. Loeffler, A. Ligtenberg, and G.S. Moschytz. Practical fast 1-d dct algorithms with 11 multiplications. In *International Conference on Acoustics, Speech, and Signal Processing*, pages 988–991, 1989.

[21] W. Chen, C.H. Smith, and S.C. Fralick. A fast computational algorithm for the discrete cosine transform. *IEEE Transactions on communications*, 25(9):1004–1009, 1977.

[22] `http://www.ti-wmc.nl/downloads/OSYRES_Product_Sheet_rev_2_0.pdf`.

[23] `http://ecos.sourceware.org/`.

[24] Arm946 technical data. `http://www.arm.com/products/CPUs/ARM946E-S.html`.

[25] R. Swamy, M. Khorasani, Y. Liu, and Bates S. Elliot, D. and. A fast, pipelined implementation of a two-dimensional inverse discrete cosine transform. In *Proceedings of Canadian Conference on Electrical and Computer Engineering*, pages 665–668, 2005.

[26] Committee on Networked Systems of Embedded Computers. *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers*. National Academy Press, 2001. ISBN: 0-3090-7568-8.

[27] Tsmc 0.13-micron technology. http://www.tsmc.com/download/enliterature/013_bro_2003.pdf.

[28] Tsmc 0.18-micron technology. http://www.tsmc.com/download/enliterature/018_bro_2003.pdf.

[29] International technology roadmap for semiconductors. http://www.sia-online.org/downloads/Issue_ITRS.pdf.

[30] Gustavo Martinez. *Application Report: TMS320C6455/C6454 Power Consumption Summary*. Texas Instruments, September 2006. document: SPRAAE8A.

[31] *TMS320C64x+ DSP Image/Video Processing Library Programmers's Reference*. Texas Instruments, march 2006. Literature Numuber: sprueb9.

[32] http://www.elecdesign.com/Articles/Index.cfm?AD=1\&ArticleID=1004.

[33] Tsmc advanced technology overview. http://www.tsmc.com/download/english/a05_literature/Advanced_Technology_Overview_Brochure_2006.pdf.

APPENDIX A

Chen's iDCT algorithm on the Montium

The 1D iDCT algorithm of Chen can be divided into 22 operations that can be performed in one clock cycle on a Montium. Table A.1 shows an overview of the 22 operations. Values starting with an I are the input samples for the iDCT, values with an O are the output values, values with an X are intermediate values and values starting with an C are constants. The flow graph of this 22 operations is shown in Figure A.1.

1:	$X_1 = I_7 \times C_1$ -	9:	$X_9 = I_4 \times C_4$ -	15:	$X_{15a} = X_{10a} + X_{14}$ $X_{15b} = X_{10a} - X_{14}$
2:	$X_2 = I_1 \times C_7 - X_1$ -	10:	$X_{10a} = I_0 \times C_4 + X_9$ $X_{10b} = I_0 \times C_4 - X_9$	16:	$X_{16a} = X_{10b} + X_{12}$ $X_{16b} = X_{10b} - X_{12}$
3:	$X_3 = I_3 \times C_5$ -	11:	$X_{11} = I_2 \times C_6$ -	17:	$X_{17} = X_{4b} \times C_4$ -
4:	$X_{4a} = X_2 + (I_5 \times C_3 - X_3)$ $X_{4b} = X_2 - (I_5 \times C_3 - X_3)$	12:	$X_{12} = X_{11} - I_6 \times C_2$ -	18:	$X_{18a} = X_{6b} \times C_4 + X_{17}$ $X_{18b} = X_{6b} \times C_4 - X_{17}$
5:	$X_5 = I_5 \times C_5$ -	13:	$X_{13} = I_2 \times C_2$ -	19:	$O_0 = X_{15a} + X_{6a}$ $O_7 = X_{15a} - X_{6a}$
6:	$X_{6a} = X_8 + (I_3 \times C_3 + X_5)$ $X_{6b} = X_8 - (I_3 \times C_3 + X_5)$	14:	$X_{14} = I_6 \times C_6 + X_{13}$ -	20:	$O_1 = X_{16a} + X_{18a}$ $O_6 = X_{16a} - X_{18a}$
7:	$X_7 = I_1 \times C_1$ -			21:	$O_2 = X_{16b} + X_{18b}$ $O_5 = X_{16b} - X_{18b}$
8:	$X_8 = I_7 \times C_7 + X_7$ -			22:	$O_3 = X_{15b} + X_{4a}$ $O_4 = X_{15b} - X_{4a}$

Table A.1: Operations of the iDCT algorithm

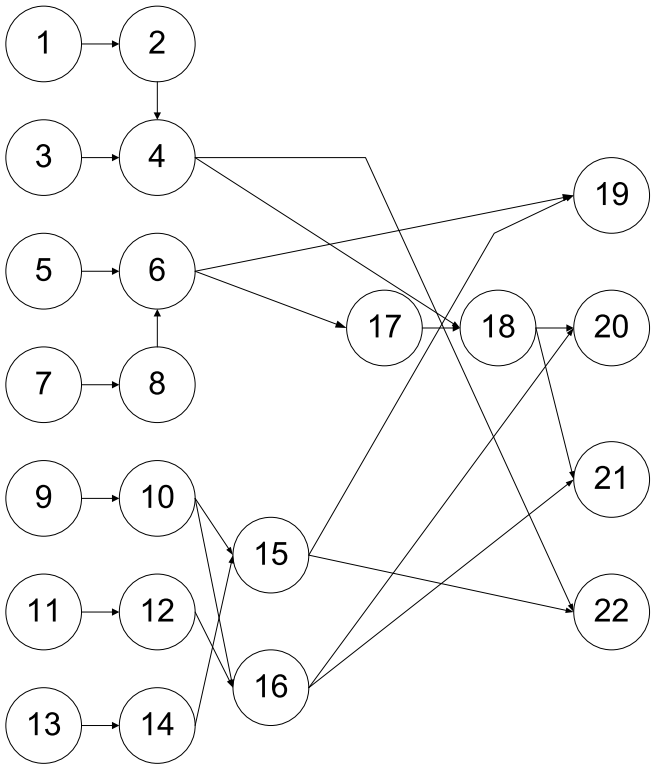


Figure A.1: Flow graph iDCT

APPENDIX B

Configuration specification

Graph definition

```
<DEFINITION> ::= <GENERAL_CONFIG> [<ROUTES>] [<PIPES>]
                 [<MONTIUM_PIPES_IN>] [<MONTIUM_PIPES_OUT>]
                 [<USB_PIPES_IN>] [<USB_PIPES_OUT>] [<TIMERS>]
                 [<SIGNALS>] <PROCESSES>
<GENERAL_CONFIG> ::= 'configuration' <NAME> '\n'
```

Start of the configuration, <NAME> is the identifier of the configuration.

Configuration of the routers

```
<ROUTES> ::= 'route' <ROUTER> <DEVICE> <PORT> <DEVICE> <PORT>
             '\n'[<ROUTES>]
<ROUTER> ::= 'router0' | 'router1'
<DEVICE> ::= 'ARM' | 'montium0' | 'montium1' |
             'montium2' | 'router1' | 'router2'
<PORT>   ::= <DIGIT>
```

Configuration for the routers of the NoC. There are two routers in the NoC, router0 and router1, <ROUTER> defines for which router this config rule is meant. The first <DEVICE> and <PORT are the source device and port, the second <DEVICE> and <PORT> define the destination.

Definition of pipes

```
<PIPES>        ::= 'pipe' <NAME> <PIPE_SIZE> [<RD_THRESHOLD>]
                   [<WR_THRESHOLD>] {<PIPE_EVENT>} '\n' [<PIPES>]
<PIPE_SIZE>    ::= <DIGIT> {<DIGIT>}
<RD_THRESHOLD> ::= 'rd_threshold' <DIGIT> {<DIGIT>}
```

```
<WR_THRESHOLD> ::= 'wr_threshold' <DIGIT> {<DIGIT>}
<PIPE_EVENT>   ::= <NAME> 'not full' | 'not empty'
```

Data can send between processes through pipes. The name of every pipe has to be unique. The <PIPE_SIZE> parameter defines the size of the buffer. For a proper working pipe this has to be equal than or bigger than the biggest packet size.

A pipe can create an event. This can be done when the pipe is not empty anymore (so a process can read from the pipe) or when the pipe is not full anymore (so a process can write into the pipe). The name of every event (pipe-events, timers and signals) has to be unique. This pipe events are defined by the <PIPE_EVENT> property.

Standard are the triggers for not empty and not full initiated on 1, so when there is one bit free in the pipe, the not full trigger is fired. Even so the not full trigger is fired when there is one bit in the pipe. These thresholds can be changed. The read threshold (for the not full trigger) can be changed by <RD_THRESHOLD>, the write threshold (for the not empty trigger) can be changed by <WR_THRESHOLD>.

Definition of Montium pipes

```
<MONTIUM_PIPES_IN>  ::= 'montium_pipe_in' <NAME> <PIPE_NUMBER>
                        <PIPE_SIZE> [<PIPE_TYPE>] [<RD_THRESHOLD>]
                        [<WR_THRESHOLD>] {<PIPE_EVENT>} '\n'
                        [<MONTIUM_PIPES_IN>]
<MONTIUM_PIPES_OUT> ::= 'montium_pipe_out' <NAME> <PIPE_NUMBER>
                        <PIPE_SIZE> [<PIPE_TYPE>] [<RD_THRESHOLD>]
                        [<WR_THRESHOLD>] {<PIPE_EVENT>} '\n'
                        [<MONTIUM_PIPES_OUT>]
<PIPE_NUMBER>       ::= <DIGIT>
<PIPE_TYPE>         ::= '16bit' | '32bitH' | '32bitV'
```

Montium pipes are the pipes connected to the bridge to the NoC. These structure of the pipes is exactly the same as the structure of the pipes between processes. The pipes to the Montium have all options that normal pipes also have (name,size, thresholds, events). The name of all the pipes have to be unique, so also Montium pipes and pipes between processes need different names.

The Montium pipes in are the pipes to the Montium, so ARM processes can write data to the Montium through these pipes. The Montium pipes out are pipes from the Montium.

An extra parameter of Montium pipes is <PIPE_NUMBER. This parameter defines the number of the pipe, the port of the router to which this pipe is connected. There are four pipes from and four pipes to the Montium. The pipes are connected to router0, the router configuration configures in which way the pipes are connected to the Montiums.

The second extra (optional) parameter is <PIPE_TYPE>. This parameter defines the type of the pipe, the length of one data item. This can either be 32 bit, 16 bit or connected with an other pipe (32bitH and 32bitV). The pipes are standard 32 bit, 16 bits for the data (least significant) and 2 for the flit type (bit 16 and 17). The rest of the bits are discarded. The pipes are standard 32 bit when there is no pipe type defined.

When the pipe is defined to be 16 bit, all 16 bits are data. The flit type is than for all data items the same, the driver adds the flit bits to the data. The flit type is standard set to data.

When to pipes are connected, both pipes are 16 bit and the data from both pipes is synchronically send to router. Just as with the 16 bit pipes is the flit type for all data items the same and added by the driver. The 32bitH option connect to pipes horizontal, pipe 1 and 2 or pipe 3 and 4. The 32bitV option connect to pipes vertical, pipe 1 and 3 or pipe 2 and 4.

Definition of USB pipes

```
<USB_PIPES_IN>  ::= 'usb_pipe_in' <NAME> <PIPE_SIZE> [<RD_THRESHOLD>]
                    [<WR_THRESHOLD>] {<PIPE_EVENT>} '\n'
<USB_PIPES_OUT> ::= 'usb_pipe_out' <NAME> <PIPE_SIZE> [<RD_THRESHOLD>]
                    [<WR_THRESHOLD>] {<PIPE_EVENT>} '\n'
```

The USB pipes have the same options as normal pipes. There is only one USB pipe in and one USB pipe out. The USB pipe in goes to the USB drivers (so to the PC), the USB pipe out comes from the PC.

Definition of timers

```
<TIMERS>         ::= 'timer' <NAME> <TIMER_INTERVAL> '\n' [<TIMERS>]
<TIMER_INTERVAL> ::= <DIGIT> {<DIGIT>}
```

A timer can periodically create an event. The name of every event (pipe-events, timers and signals) has to be unique. The <TIMER_INTERVAL> parameter is the period of the event in milliseconds.

Definition of signals

```
<SIGNALS> ::= 'signal' <NAME> '\n' [<SIGNALS>]
```

With a signal a process can create an event, which can activate another process. The name of every event (pipe-events, timers and signals) has to be unique.

Definition of processes

```
<PROCESSES>       ::= 'process' <NAME> 'signals' <PROCESS_SIGNALS>
                      'pipes' <PROCESS_PIPES> 'options'
                      <PROCESS_OPTIONS> '\n'[<PROCESSES>]
<PROCESS_SIGNALS  ::= '(' [<SIGNALS_IN>] ')' '(' [<SIGNAL_OUT>] ')'
<SIGNALS_IN>      ::= <NAME> [',' <SIGNALS_IN>]
<SIGNALS_OUT>     ::= <NAME> [',' <SIGNALS_OUT>]
<PROCESS_PIPES>   ::= '(' [<PIPES_IN>] ')' '(' [<PIPES_OUT>] ')'
<PIPES_IN>        ::= <NAME> [',' <PIPES_IN>]
<PIPES_OUT>       ::= <NAME> [',' <PIPES_OUT>]
<PROCESS_OPTIONS> ::= '(' <REALTIME> ',' <RUNNING> ',' <DEVICE> ')'
<REALTIME>        ::= '1' | '0'
```

```
<RUNNING>          ::= '1' | '0'
<DEVICE>           ::= 'ARM' | 'Montium' | 'PC'
```

A process is defined by its <NAME>. This name is also defined in the task specification struct, there has to be a process in the task specification array with this name for every ARM process.

The signals (in and out) and the pipes (in and out) has to be defined in the previous configuration lines. Signals in (events) can be pipe-events, timers and signals. A process is triggered when it received all signals in.

The <PROCESS_OPTIONS> define if a process is real time or runs in slack time (for an ARM process), if it is initially running (for an ARM process) and on which device it is running. For a process running on the Montium defines <REALTIME> on which Montium the process is running.

Primitives

```
<NAME>   ::= <LETTER> {<LETTER> | <DIGIT> | <SYMBOL>}
<LETTER> ::= [a-zA-Z]
<DIGIT>  ::= [0-9]
<SYMBOL> ::= [-_]
```

APPENDIX C

Convert a process graph to a configuration

In this appendix we give an example of converting a process graph to a configuration. The process graph is shown in Figure C.1.

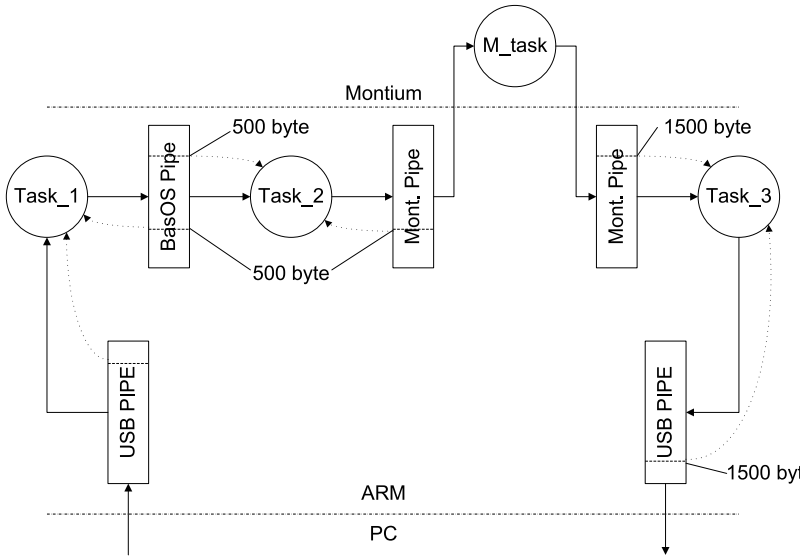


Figure C.1: Example process graph

The configuration starts with a definition, a name for the configuration. Next are the lines for configuration of the routers. Which lane the MONTIUM uses for streaming in and out data depends of the implementation of the task on the MONTIUM. In this example the MONTIUM streams data in over lane 3 and streams data out over lane 1 and the MONTIUM used is MONTIUM0. On the side of the ARM is for the input as well as for the output the first pipe used (pipe 0). The first lines over the configuration are:

```
configuration testapp
route router0 ARM 0 montium0 3
route router0 montium0 0 ARM 0
```

Next are the pipes defined. First the pipes between ARM processes are defined, then pipes to the MONTIUM and then the USB pipes. All pipes are 2000 bytes, the MONTIUM pipes are 16 bit:

```
pipe task1_task2 2000 rd_threshold 500 wr_threshold 500 event1 not_full
   event2 not_empty
montium_pipe_in to_montium 0 2000 16bit wr_threshold 500 event3 not_full
montium_pipe_out from_montium 0 2000 16bit rd_threshold 500 event4
   not_empty
usb_pipe_in to_usb 2000 wr_threshold 1500 event5 not_full
usb_pipe_out from_usb 2000 event6 not_empty
```

There are no timers and signals generated by processes, so these configuration lines can be skipped. Now the processes are defined. For MONTIUM processes its not required to define the signals and pipes connected to the process, but it can be helpful:

```
process task_1 signals (event1, event6) () pipes (from_usb)
   (task1_task2) options (1,0,ARM)
process task_2 signals (event2, event3) () pipes (taskl_task2)
   (to_montium) options (1,0,ARM)
process M_task signals () () pipes (to_montium) (from_montium)
   options (0,0,Montium)
process task_3 signals (event4, event5) () pipes (from_montium)
   (to_usb) options (1,0,ARM)
```

Now the configuration is complete. MONTIUM0 is used so the `#use_montium0` define has to be set. The `#use_usb` define must also be set because the USB connection is used. Then there is a MONTIUM configuration required in variable `montium_cfg_0` and there must be three ARM tasks: `task_1`, `task_2` and `task_3`.

APPENDIX D

Configuration structure Controller

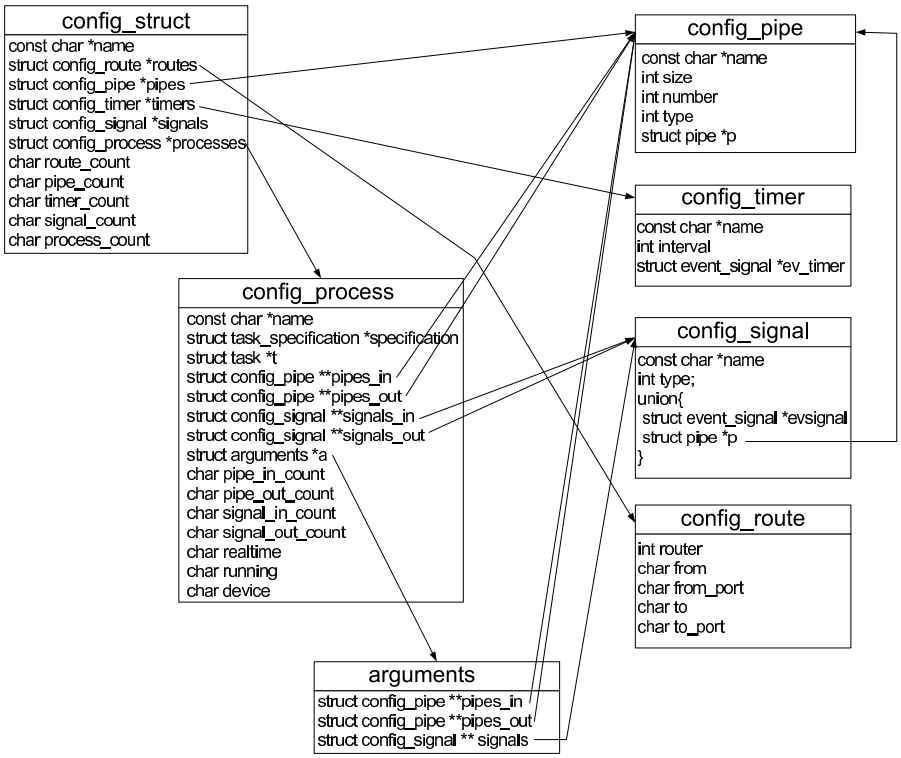


Figure D.1: Configuration structure

APPENDIX E

Layout file Controller