
UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

# A configurable processing platform and an application to video coding

Miguel Ribeiro
(*Licenciado*)

Dissertação para obtenção do Grau de Mestre em Engenharia Electrotécnica e de Computadores

Orientador: Doutor Leonel Augusto Pires Seabra de Sousa

**Membros do juri**

Presidente: Doutor Leonel Augusto Pires Seabra de Sousa
Vogais: Doutor José Carlos dos Santos Alves
Doutor José Teixeira de Sousa

Julho de 2007

# Abstract

Motion estimation is a central operation in video processing and coding, but it is also one of the most computational intensive components. Several fast search algorithms have been proposed for motion estimation based on block matching. These algorithms iteratively apply different search patterns and usually define individual initial search locations. They are not easily implemented in hardware, because the processing depends on the data and on the adopted search pattern. Therefore, hardware architectures proposed so far are hardwired to a single basic search algorithm. This thesis proposes a new mechanism, and a new architecture, to implement an hardware motion estimation processor that supports most of the actual known fast search algorithms. Based on these proposals, a motion estimation processor was designed and a prototype was implemented in a Field-Programmable Gate Arrays (FPGA). Experimental results show that the processor is efficient and flexible, able to configure the search in run-time for different conditions. Moreover, it allows real-time block matching motion estimation in CIF images by applying any of the fast search algorithms, such as the three-step, the four-step or the diamond search.

# Keywords

Video coding; fast search algorithms; motion estimation; configurable architectures.

# Resumo

A estimação de movimento é uma operação central em processamento e codificação de vídeo mas também um dos componentes mais computacionalmente intensivos Foram já propostos vários algoritmos de pesquisa rápidos para estimação de movimento. Estes algoritmos aplicam iterativamente padrões de pesquisa diferentes e definem áreas de pesquisa distintas.

Não são facilmente implementáveis em hardware, uma vez que o desenrolar do processamento depende dos dados de entrada e do padrão de pesquisa seleccionado. Assim sendo, as arquitecturas em hardware propostas até à data estão fixas a um único algoritmo de pesquisa básico. Este trabalho propõe um novo mecanismo e uma nova arquitectura para implementar e, hardware um processador para estimação de movimento que suporta a maioria dos algoritmos de pesquisa rápidos actuais. Foi desenvolvido um processador para estimação de movimento, tendo sido implementado um protótipo numa FPGA.

Os resultados experimentais mostram que o processador é eficiente e flexível e capaz de alterar a configuração do algoritmo de pesquisa em tempo real de modo a adaptar-se a diferentes situações.

O processador permite efectuar estimação de movimento baseada em blocos em imagens CIF em tempo real utilizando algoritmos de pesquisa rápidos (três passos, diamante, quatro passos).

# Palavras Chave

Codificação de vídeo; algoritmos rápidos de pesquisa; estimação de movimento; arquitecturas reconfiguraveis.

# Acknowledgments

Several people contributed to the research presented in this thesis, and that I would like to thank.

First of all, I would like to thank Prof. Leonel Sousa for his guidance, support and the invaluable help with all the documentation associated with this work.

My thanks also to all my colleagues at *Instituto de Sistemas e Computadores* (INESC), especially José Germano and Svetislav Momcilovic for bearing with my many requests for remote assistance, enabling me to perform most of the work off site. I would also like to thank Tiago Dias for his splendid work that inspired the "Video Coding Overview" section on this thesis.

Miguel Ribeiro

# Contents

# List of Acronyms

**3SS** Three-Step Search

**ACE** Advanced Configuration Environment

**ASIC** Application Specific Integrated Circuit

**ASIP** Application Specific Instruction-set Processor

**BM** Block Matching

**BRAM** Block RAM

**CF** Compact Flash

**CIF** Common Intermediate Format – $352 \times 288$

**CoDec** Coder/Decoder

**CPLD** Complex Programable Logic Device

**DCT** Discrete Cosine Transform

**DDR** Double Data Rate

**DFD** Displaced Frame Difference

**DMA** Direct Memory Access

**DS** Diamond Search

**DSP** Digital Signal Processing

**EDK** Embedded Development Kit

**FPGA** Field-Programmable Gate Arrays

**fps** frames per second

**FSBM** Full-Search Block-Matching

**FSS** Four-Step Search

**HDL** Hardware Description Language

**HVS** human visual system

**IDE** Integrated Drive Electronics

**IPIF** IP Interface

**IP** Intelectual Property

**LE** Logic Element

**LC** Logic Cell

**LUT** Look-up Table

**MAC/PHY** Medium Access Control / Physical layer interface

**MB** macroblock

**MC** Motion Compensation

**ME** Motion Estimation

**MVFAST** Motion Vector Field Adaptive Search Technique

**MV** motion vector

**PM** personality module

**PLB** Processor Local Bus

**PLL** Phase Locked Loop

**QCIF** Quarter Common Intermediate Format – $176 \times 144$

**SAD** Sum of Absolute Differences

**SDU** Search Decision Unit

**SoC** System-on-a-Chip

**USB** Universal Serial Bus

**VHDL** Very High Speed Integrated Circuit (VHSIC) Hardware Description Language

**VLC** Variable Length Code

**XPS** Xilinx Platform Studio

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Researches in the early '80s gave rise to a new kind of devices that have the ability to change its internal configuration in order to implement arbitrary logic circuits. These devices, from which the most well known are Complex Programable Logic Devices (CPLDs) and FPGAs, have since then been growing in size and features and are currently able to implement all but the most demanding systems.

The emergence of these devices led to the development of reconfigurable processing platforms that can either be integrated into an existing computer system (e.g. connected to a PCI bus any other interface – USB, parallel port, firewire, etc) or can themselves form an autonomous computer system capable of running an operating system, as is the case with Xilinx's ML3x0 series boards.

The tight integration that is achieved between the reconfigurable resources and the development and test tools both greatly simplify the development stages and allow a faster and more thorough test and validation of the circuits and systems.

The availability of reconfigurable resources to the operating system and applications has the effect of enabling the development of systems that reconfigure the hardware resources into specialized units in order to perform certain tasks more efficiently. This approach is further propelled by recent FPGA technologies that allow the on-the-fly reconfiguration of part of the hardware resources of a live system.

This thesis focuses on natural video coding which is a class of applications that, for the last decade, has been able to keep its processing requirements one step ahead of the computing power offered by mainstream general purpose processors.

Motion Estimation (ME) is one of the major and most demanding components in recent video coding standards, such as H.26x, MPEG-x and

H.264/AVC [1]. All these coding standards adopt the Block Matching (BM) approach, typically using a macroblock (MB) size of $16 \times 16$ pixels, but advanced video coding supports variable MB size and sub-block partitioning modes for more accurate motion estimation.

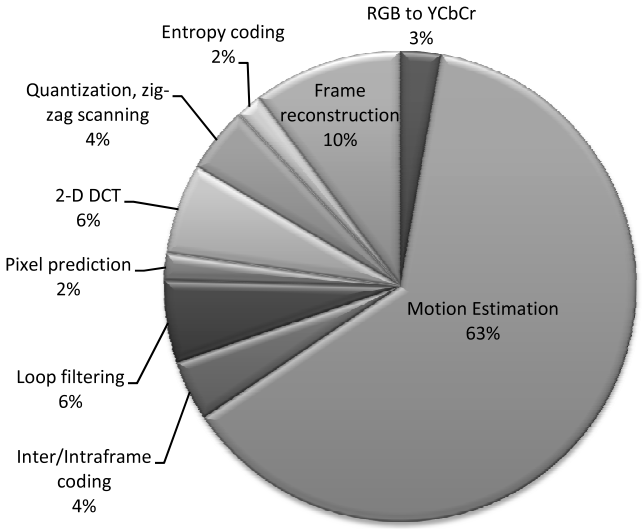


Figure 1.1: Distribution of the processor operations by the different tasks in H.261 compression.

As an example, Figure 1.1 shows the distribution of the processor operations needed by the different tasks of an encoding process using the H.261 codec on a Common Intermediate Format – $352 \times 288$ (CIF) sequence, and using the Three-Step Search (3SS) fast search algorithm [2]. For the full search algorithm, the widely used Sum of Absolute Differences (SAD) distance metric has to be computed for each candidate block. Therefore, fast search algorithms have been proposed to reduce the search space, by guiding the search procedure only through the locations where the SAD is probably lower. Examples of these algorithms are the 3SS [3], the Diamond Search (DS) [4], and more recently the Motion Vector Field Adaptive Search Technique (MVFAST) [5], that exploits information about motion vectors and previously computed distances in order to predict and adapt the actual search space. Figure 1.1 clearly shows that, even using a fast search algorithm, the motion estimation task is the dominant one in video coding.

Up until now, these algorithms have been mainly applied in pure software implementations, or by designing Application Specific Integrated Circuit (ASIC) that support a single search algorithm [6]. This type of hardwired

processors is mainly used for mobile devices, where energy restrictions are quite severe. However, these hardware accelerators are very inefficient in terms of reproduction and adaptation costs. Recently, an Application Specific Instruction-set Processor (ASIP) has been proposed for motion estimation [7, 8], but the programmability of this processor requires the usage of program memory, which leads to non-optimal solutions both in terms of circuit area and power consumption.

This thesis proposes a new mechanism for designing hardware processors that allows to configure the search procedure in run-time. This mechanism corresponds to express search patterns and initial locations trough lookup tables that can be integrated in hardwired control units to guide the search. With this controller and a single data-path, a simple and efficient hardware architecture is proposed to design motion estimators, with the flexibility of programmable approaches.

FPGA technology has been adopted in this work not only because it is well suited for prototyping but also because it allows to reconfigure the hardware, namely the data path, according to the specifications. The FPGA is integrated in a development board that can also allow the creation of a system where the software is able to automatically configure the co-processor hardware resources in run-time.

## 1.1 Original Contributions

The main contributions of this thesis are:

- the introduction of a novel architecture for motion estimation that is very efficient in terms of hardware usage and leads to the implementation of high performant motion video coding processors. This architecture is capable of performing most of the well known fast search algorithms, such as the 3SS, FSS, DS, etc,.
- the usage of a modern FPGA-based prototyping system as a reconfigurable computing platform and its configuration as a Linux operating system, namely for video encoding.
- the development of a prototype complete coding system based on an existing coding standard which uses the developed hardware motion estimator running on the configured linux system on the prototyping board.

## 1.2 Thesis Outline

This thesis is organized in six chapters, including those regarding to the introduction and conclusions – the first and sixth chapters, respectively. The introductory chapter describes the background and the motivation to this work. The second chapter presents a brief overview of video coding technology, focusing especially on the motion estimation process. Chapter 3 describes the reconfigurable hardware platform used throughout this work. Besides presenting the characteristics and resources of the platform, it also provides information about the developed base peripheral module and a driver that allows the interface between the module and the linux system. Chapter 4 introduces a novel motion estimation search algorithm configuration mechanism and an efficient hardware architecture that implements this mechanism. Chapter 5 shows a prototype of a processor with this architecture, a hybrid encoder, and the experimental results obtained with this system. Chapter 6 concludes this thesis and presents future work that will further improve the proposed system.

# Chapter 2

# Video Coding: The Target Application

A digital video signal is basically a 3D array of pixel values where two dimensions provide the spatial characteristics of the frames and the third represents the passage of time.

The number of pixels in each frame, the amount of information encoded in each pixel and the number of frames per second depend heavily on the application. Typical values for digital television are framesizes of 720 x 480 pixels using 24 bits per pixel with a framerate of 30 frames per second (fps), which amounts to a raw data rate of 248.83 Mbit/s (roughly 1.7GB per minute). Data rates of this magnitude are incompatible with widespread data channels and media, and consequently efficient compression techniques must be used in video applications in order to make them feasible.

The main goal of video coding is to minimize the bit rate of digital video signals, keeping the perceived image quality as high as possible.

Fortunately, video signals are most suitable for compression due to two major factors. First, there is a considerable amount of information that is irrelevant from a perceptual point of view. Second, due to the high degree of data correlation that exists in an image sequence, both within each image (spatial correlation) and or between different images (temporal correlation). Moreover, the RGB color space used for polychromatic video sequences adds an extra layer of redundant information, which can also be exploited to achieve further compression.

### 2.0.1 Exploiting data redundancy

A video signal can be modeled as a sequence of still images which are taken at fixed time intervals. Supposing that the time interval between images is

relatively short, it is reasonable to expect that scene information does not vary too much between neighbor images. Consequently, a high degree of temporal redundancy between consecutive pictures can be expected.

This kind of signal, according to signal processing theory, is well suited to predictive coding schemes, which are the most efficient to compress strongly correlated data [9].

In conventional predictive coding, both ends of the system share a model that provides an estimation for the evolution of the signal. The output of this model is called the predicted signal. The transmitting end codes and transmits the difference between the original signal and the predicted one, which is then used by the receiving end to reconstruct the signal. Consequently, the better the prediction the smaller the prediction error, which results in lower transmission bit rates.

In video coding, and for most sequences, consecutive frames are mostly similar so a fairly good prediction for a given pixel is the equivalent pixel on the previous frame. A better prediction can be achieved if the moving objects on the frame sequence are identified and used on the prediction model. A technique known as ME is described on section 2.0.2. There are a number of variations but the common aim is to provide information about the motion present on the sequence. This information is then used to enrich the prediction model by means of a procedure known as Motion Compensation (MC).

A compression process that uses both the MC and predictive coding techniques provides a very good compression of video sequences, because the prediction error is minimized.

Typical unprocessed images exhibit characteristics that can be exploited by a compression scheme. On the one hand, neighboring pixels in an image are highly correlated and consequently present a high degree of redundancy. On the other hand, images have significant amounts of information that is not perceptible to the human visual system (HVS).

A technique commonly used to reduce the amount of irrelevant information in images is the transformation of color space. Although most video capturing devices operate using the $RGB$ color space, in which every color is defined by means of a linear combination of only three primaries – red (R), green (G) and blue (B) – a different color space is used in digital video coding: the $YC_BC_R$ color space. In this color space, the Y channel is called luma and conveys the brightness information for the image. The $C_B$ and $C_R$ channels are the blue and red chroma components and carry the color information. The great advantage of $YC_BC_R$ color space is that some of the information can easily be discarded in order to reduce bandwidth. The human eye has fairly little color sensitivity: the accuracy of the brightness information of the luminance channel has far more impact on the image discerned than that

of the other two. Hence, the amount of data consumed by the chrominance channels can be considerably reduced, leaving the eye to extrapolate much of the color. For example, by sub-sampling the two chrominance channels by a factor of two the number of pixels required to represent each picture in the video sequence is halved.

Note that by removing the irrelevant information from the video signal, some amount of distortion (fidelity loss) is introduced in the coded video signal, which is why these compression processes are known to be lossy [2].

Although lossy image compression schemes can be *sample-based coding* or *block-based coding*, only the latter is usually applied to video coding. The main reason for this is that block-based coding schemes yield better compression ratios for the same level of distortion an is computationally less demanding than sample-based coding schemes [2].

Most video coding systems use the so-called transform-domain block coding, where pixels from an image are grouped into blocks which are then transformed into another domain, such as the frequency domain, in which a more compact and uncorrelated representation of image data can be achieved.

Although several functions can be used for the transformation of an $N \times N$ image block from the spatial domain to the frequency domain, the transform function used by most of the standards is the Discrete Cosine Transform (DCT) or an approximation of this transform, due to its properties and many benefits:

- compactness efficiency close to the KLT's;
- image independency;
- efficient data irrelevancy exploitation.
- orthogonality;
- separability, which simplifies the hardware implementation;
- can be performed using fast algorithms.

The DCT transform function, whose forward and inverse expressions are depicted in Equations 2.1 and 2.2, respectively, is usually applied to blocks of up to $8 \times 8$ pixels for the following reasons. Firstly, because an $8 \times 8$ block size does not impose significant memory requirements. Secondly, because the computational complexity of an $8 \times 8$ DCT is still manageable by most computing platforms. Thirdly, because block sizes greater than $8 \times 8$ do not offer significantly higher compression ratios, since according to rate distortion theory, for typical images the covariance function decays rapidly at distances longer than 8 pixels [2].

$$DCT(i,j) = \frac{C(i)C(j)}{\sqrt{2N}} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x,y) \cos\left(\frac{(2x+1)i\pi}{2N}\right) \cos\left(\frac{(2y+1)j\pi}{2N}\right) \tag{2.1}$$

$$pixel(x,y) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \frac{C(i)C(j)}{\sqrt{2N}} \cos\left(\frac{(2x+1)i\pi}{2N}\right) \cos\left(\frac{(2y+1)j\pi}{2N}\right) \tag{2.2}$$

The use of the DCT, by itself, does not provide any kind of data compression. In order to compress data, the frequency domain coefficients obtained through the use of the DCT must be quantized. While higher compression ratios can be achieved using coarser quantization steps, this causes the loss of more information, which lowers the quality of the coded video signal. Thus, the HVS characteristics must also be taken into account in this quantization process. By quantizing the lower frequency DCT coefficients, to which the HVS is much more sensitive, with lower quantization steps and the higher frequency DCT coefficients with greater quantization steps, good compression ratios can be achieved without introducing significant degradation in the image quality

Using the described compression techniques most data redundancies can be removed from video signals, leading to digital video signals with much lower bit rates, which can be further reduced by using entropy coding techniques [2].

In entropy coding compression schemes, symbols from video signals are mapped into codewords according to the probability model of the signal. These probability models can be described either from the image data or from *a priori* assumptions related to the data. The key idea is to use small codewords for symbols that occur with high probability and long codewords for symbols which occur with low probability. This equates to using a Variable Length Code (VLC) [2] to map symbols into codewords and produces by itself a fair amount of compression without any degradation of the image quality.

### 2.0.2 Motion estimation

As it was mentioned in section 2.0.1, in order to maximize the compression obtained with the video encoding process, the displacement of moving objects from one image to another must be compensated. To this end, the encoder must first compute the motion trajectories of the objects along the image

sequence through a process known as Motion Estimation (ME). This task defines some basic concepts namely the motion model, the ME criterion and the search strategy to be used.

### Motion model

In any given image sequence, each object has an associated motion trajectory which is defined only during the time interval in which its points are visible [10], and that must be suitably modeled so that the ME and Motion Compensation techniques can be successfully applied.

Identifying and tracking moving objects is a complex task because of the factors that can be at work in a scene, such as noise, light sources or occlusions. Nevertheless, the assumption that is usually made is that objects maintain the same intensity along the motion trajectory. Although this assumption is seldom completely valid, it provides a good model for the dominant properties of natural image sequences.

In the latter years several approaches have been proposed that model the behaviour of objects in a scene with different levels of detail, e.g., the shape of the object, the type of motion (translation, rotation, acceleration...), object shape transformation, etc. These represent a scene by a set of parameters that describe the motion of objects. The number of parameters that each model uses is closely related to the accuracy of the model.

The appropriate level of model accuracy will be determined by the application. While a security video surveillance application may require that each moving object's motion is tracked through a space with many degrees of freedom, a video coding application requires only that the scene information is efficiently conveyed, not necessarily using an accurate object identification process.

Most existing video coding standards use the block translational motion model [11] that uses fixed rectangular blocks of pixels and only two parameters – two MV coordinates that describe the blocks' translation. Although this motion model is not capable of describing arbitrary 2-D motion fields, it is used due to its simplicity and good compression characteristics.

### Estimation criterion

The estimation criterion is a measure of how well the model is approaching the input sequence.

Depending on the motion model that is used, different estimation criteria must be used. While for a complex motion model that identifies and models the edges of moving objects a sophisticated estimation criterion is required

which, e.g., takes into account the uncertainty of edge detection process, for a basic block translational model a simple distance metric produces good results.

The application of the constant-intensity assumption leads to an important class of estimation criteria, which consider that a frame is the result of a displacement of the previous frame, or regions within it[1], and therefore measure the difference between the original frame and the one obtained by means of these displacements.

Nowadays the majority of video Coder/Decoders (CoDecs) use the SAD metric as the estimation criterion because it is not a very computationally intensive procedure and has a low sensitivity to noise.

The SAD function expression is depicted in Equation 2.3, in which $R(u, v)$ denotes the set of pixels that belong to the reference area and $S(u + c, v + l)$ the set of pixels regarding to the search area with a common displacement of $(c, l)$.

$$SAD(c, l) = \sum_{v=0}^{N-1} \sum_{u=0}^{N-1} |R(u, v) - S(c + u, l + v)| \tag{2.3}$$

**Search strategy**

The search strategy is the heuristic that is used to find the parameter values that when used by the model match the input sequence. The choice of the search strategy to be used in the ME stage is decisive to the global performance of the coding process.

Most modern video coders are based on the block translational motion model and implement BM search strategies [2, 11], by estimating the amount of motion in an image on a block by block basis.

Given a search picture and a block in the reference picture, the objective of the block matching ME algorithms is to find a similar block in the search picture that best matches the block in the reference picture, according to the estimation criterion in use. Usually, this class of algorithms defines a square geometry for the image blocks and the SAD as the estimation criterion.

Although the optimum solution would be found by searching the whole search picture for the best match, this approach is very computational demanding and therefore impractical for real-time applications. Consequently, the search range is usually restricted to a search region around the original

[1]When the whole frame is considered the criteria are known as Displaced Frame Difference (DFD). When regions within the frame are considered the criteria are known as *displaced region difference*

location of the block in the reference picture. This has only a small impact on the quality of the estimation, since statistically the best match for a block is found close to the center of the search area.

Several different search strategies have been proposed [12, 13, 14, 15, 16, 17], which fall into one of two classes:

- **Optimal search strategies**, in which motion-compensated predictions for all possible motion candidates are compared with the original image within the region of support of the model. The candidate yielding the best match for the selected criterion is the optimal estimate. Although this search strategy is the best one in terms of the predicted image quality and algorithm simplicity, it is also the most computationally intensive.

- **Sub-optimal search strategies**, aimed at alleviating the computational expenses of optimal search strategies. This complexity reduction of the search procedure is usually attained either by reducing the number of search locations [18, 19], or by using less pixels to compute the estimation criterion [20, 21, 22]. Even though these strategies are computationally more efficient, requiring less computation time and hardware resources, there is no guarantee they provide the optimal solution [2].

The Full-Search Block-Matching (FSBM) is commonly used on dedicated hardware encoders, because of its very regular nature and simplicity. This is an optimal search algorithm and thus is guaranteed to find the best match within the search area which is important on high-quality or low bit rate applications. The usual implementation scans all the possible locations on the search area in a raster motion. For a standard $[-8;-8]-[7;7]$ search window, 256 matches are performed for each MB per search frame. Figure 2.1 depicts a possible implementation for the FSBM algorithm.

A large number of sub-optimal strategies have been proposed. One of the simplest is the 3SS which is described on Figure 2.2.

The main idea behind this algorithm is to reach a good estimation by using a small subset of the possible search locations. This is achieved by dividing the search process into three steps, each one testing a small number of possible locations. The first step tests the center position and eight positions located a $9 \times 9$ pixel grid centered on the initial position. The test that produces the smallest error is then used as the center point for the second step of the search which tests the eight positions located on a $5 \times 5$ pixel grid. As before, the test that produces the smallest error is used as the center the

$(x, y) \Leftarrow (0, 0)$ {motion vector initialization}
$SAD\,(x, y) \Leftarrow \infty$
**for** $c = -p$ to $p$ **do** {$(2p + 1) \times (2p + 1)$ search area}
  **for** $l = -p$ to $p$ **do**
    $SAD\,(c, l) \Leftarrow 0$ {SAD similarity measure initialization}
    **for** $u = 0$ to $N$ **do** {$N \times M$ reference macroblock}
      **for** $v = 0$ to $M$ **do**
        $SAD\,(c, l) += |R\,(u, v) - S\,(c + u, l + v)\,|$
      **end for**
    **end for**
    **if** $SAD\,(c, l) < SAD\,(x, y)$ **then**
      $(x, y) = (c, l)$ ; $SAD\,(x, y) = SAD\,(c, l)$
    **end if**
  **end for**
**end for**
**return** $(x, y)$ {motion vector = $(x, y)$}

Figure 2.1: FSBM algorithm using the $SAD$ similarity function.

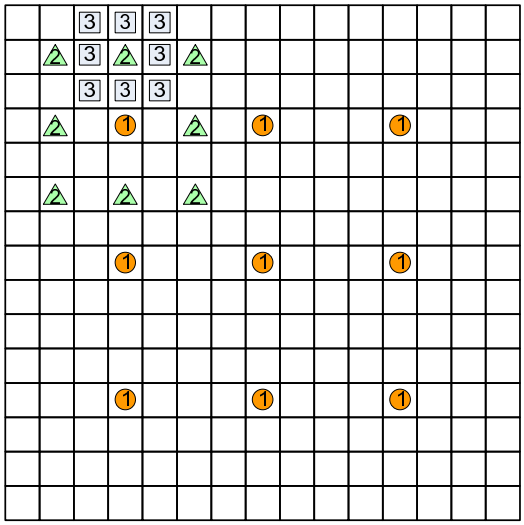


Figure 2.2: Three Step Search Algorithm

third and last step. This tests the eight positions on a $3 \times 3$ pixel grid. This algorithm tests only 25 possible positions on a $[-7; -7] - [7; 7]$ window (225 possible positions). However, TSS uses a uniformly allocated search pattern

in its first step, which is not very efficient to catch small motions appearing in stationary or quasi-stationary blocks.

Another well known algorithm is the Four-Step Search (FSS), which performs better on these conditions, because the initial search step uses a finer grid. This algorithm is composed of four steps, where the first three use a $5 \times 5$ grid and the final one uses a $3 \times 3$ grid. This algorithm uses two important optimizations:

- On each of the first three steps, if the best match is found on the center of the search grid, the algorithm proceeds directly to the last step ($3 \times 3$ window). This means that for small motion situations, the algorithm can use as little as 17 matches.
- The second and third steps only need to perform 5 tests when the previous best match was found in a corner or 3 matches if is was found in the middle of the vertical or horizontal axis of the search window, because all the other positions have already been tested on the previous step.

This way, in the worst case, the algorithm performs 27 matches, and only 17 matches in he best case.

Yet another algorithm, the Diamond Search (DS) uses diamond shaped windows instead of squared ones. The major difference on this approach is that the number of steps is not restricted, a priori. The initial step uses a 9 position diamond grid and chooses the one that yields the best match to center the next step, as depicted on Figure 2.3. The following steps use the same grid, but only perform the matching process on the positions that haven't been tested yet, which means 3 matches if the motion was diagonal and 5 if the motion was horizontal or vertical.

This procedure is recursively repeated until the best match is found on the center position. When this happens the algorithm switches to the final step, which uses a 4 position pattern. Figure 2.3 shows a possible evolution for the DS algorithm. Although only 3 steps are depicted, the process can run until either the search window or motion vector size limits are reached.

### 2.0.3 Picture types

Video standards define different picture types that are organized in classes according to the compression techniques which are used to exploit the redundant information contained in those pictures.

Pictures that are coded using techniques that only exploit the spatial redundancies, i.e., without referencing any other neighbor pictures, belong to the INTRA class. Thus, INTRA pictures (I-pictures) only provide moderate

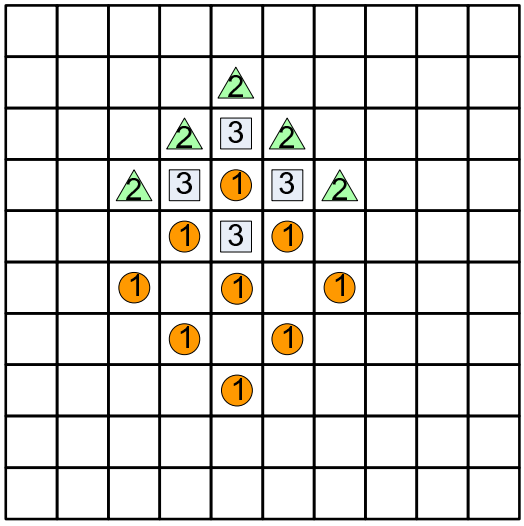


Figure 2.3: Diamond Search Algorithm

compression ratios. Nevertheless, data compressed using this intraframe coding allows for fast random access, which is important in many digital video applications, such as video recording, editing and interactive television.

On the other hand, if motion-compensated prediction is used, the picture is said to be of the INTER class. Depending on the MC scheme that is applied to code the picture, different compression ratios can be achieved. If the frame is coded using either past and/or future pictures (B-pictures), the compression ratios are higher than those achieved with a prediction using only past pictures (P-pictures). Consequently, INTER type pictures provide much higher compression ratios than INTRA type pictures.

Figure 2.4 shows the relationship between the three main picture types in a video sequence composed by seven frames. Pictures p1 and p7 are I-pictures whereas p4 is a P-picture. The remaining ones are B-pictures. As it can be seen from this figure, P-pictures are coded using only past I- and P-pictures. However, B-pictures can be coded using both past and future I- and P-pictures so that temporal redundancy can be better exploited. This can be clearly seen in this example, in which p3 is coded using motion compensation prediction from p1 and p4.

### 2.0.4 Typical video CoDec architectures

The typical architecture of a video coder that uses the techniques described in the previous sections is presented in Figure 2.5. It should be noted that since video coding standards only define the syntax and semantics of

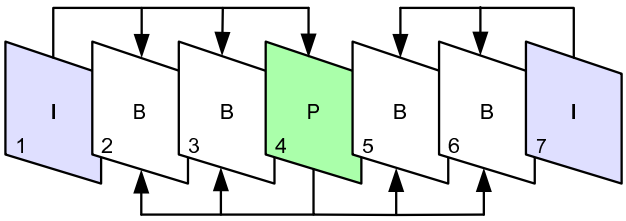


Figure 2.4: Example of inter-dependence among I-, P- and B-pictures in a video sequence.

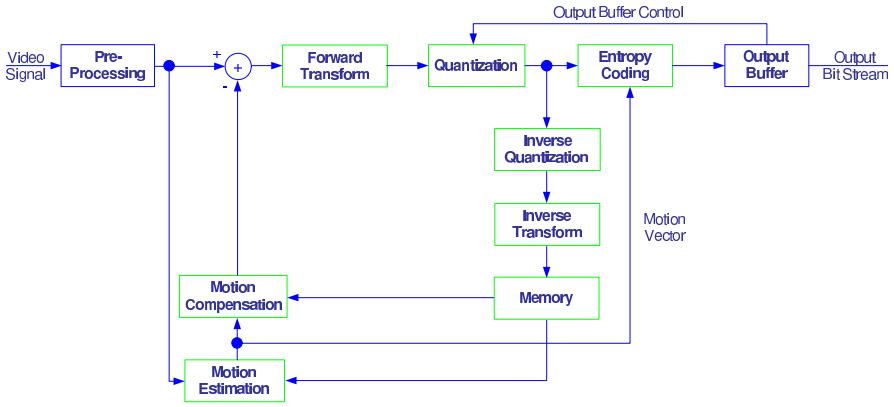


Figure 2.5: Typical architecture of a video coder.

the coded bit stream, as well as the video decoding process, manufacturers of video coders are not compelled to implement the specific architecture depicted in Figure 2.5.

As it can be seen from Figure 2.5, the encoding process usually begins with some pre-processing of the video signal. Color conversion to the $YC_BC_R$ image format, pre-filtering, sub-sampling and format translation from an interlaced to progressive picture formats are some of the tasks that may be done in this stage.

In the next stage of the video coding process, the encoder selects the coding type for the input picture. The picture is then coded either as an I-, P- or B-picture. While for I-pictures the coder directly encodes the pixels of each macroblock using the transform function, for P- and B-pictures the transform function is applied only to the prediction error obtained from the MC procedure. Consequently, for each macroblock in the reference picture the motion estimator must determine the coordinates of the macroblock that best matches its characteristics in a search picture. While for P-pictures this ME process is done using only past pictures, for B-pictures the process is done twice: firstly for a past picture and lastly for a future picture. Consequently,

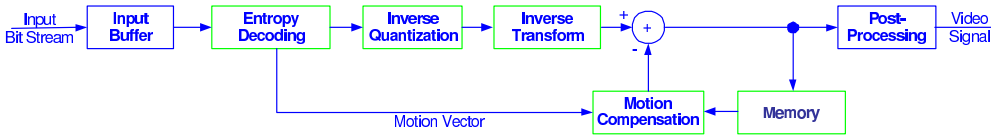


Figure 2.6: Typical architecture of a video decoder.

the prediction error macroblock for P-pictures is obtained using only a candidate block, while for B-pictures the prediction error is computed either from one of the two candidate blocks or from their average. After the transform function is applied to these prediction error terms, the transform coefficients are quantized, encoded using entropy coding techniques and stored in the output buffer. Furthermore, in applications that require a constant output bit rate, a buffer regulator is also used to adjust the quantization step for the transform coefficients, so that only slight variations exist in the output bit rate of the compressed bit stream.

The inherent lossy nature of this compression scheme makes it necessary to include part of the decoder blocks in the coder architecture, in order to guarantee that the distortion introduced by the encoding process is compensated by the decoder and that the quality of the encoded image does not significantly diverge from the original one. This is implemented through a feedback loop in the coder, where the quantized transform coefficients are inverse quantized and inverse transformed so that a copy of the encoded picture, as seen by the decoder, can be used for future predictive coding (see Figure 2.5).

The video decoding process follows the opposite scheme of the coder. Therefore, the block diagram of a typical video decoder is very similar to the feedback loop of the encoder circuit, as it can be seen from Figure 2.6. The decoder first applies entropy decoding to the input bit stream and determines the picture type from the header information. The transform coefficients of all picture macroblocks are then inverse quantized and transformed into the spatial domain by applying the inverse transform function of the one used in the coder. If the decoded picture is of the I- type, motion compensation needs not to be performed and data can be directly stored in the output buffer. On the contrary, if the decoded image is of the P- or B- type, motion compensation must be performed before data can be saved into the output buffer. This task consists in adding the area(s) of the the reference picture(s) pointed by the motion vector (MV)(s) to the decoded data.

In the next section the reconfigurable platform used in this work is presented and in chapter 4 a new configurable motion estimation architecture will be presented.

# Chapter 3

# Reconfigurable Computing Platform

## 3.1 FPGA Introduction

The historical roots of FPGAs are in complex programmable logic devices CPLDs of the early to mid 1980s. Both these devices include a relatively large number of programmable logic elements: CPLD logic gate densities range from the equivalent of several thousand to tens of thousands of logic gates, while FPGAs typically range from tens of thousands to several millions. The primary differences between CPLDs andFPGAs are architectural. A CPLD has a somewhat restrictive structure consisting of one or more programmable sum-of-products logic arrays feeding a relatively small number of clocked registers. The result of this is less flexibility, with the advantage of more predictable timing delays and a higher logic-to-interconnect ratio. The FPGA architectures, on the other hand, are dominated by interconnect. This makes them far more flexible (in terms of the range of designs that are practical for implementation within them) but also more complex to design and with higher power consumption.

The programmable logic components on an FPGA can be programmed to independently assume different configurations, e.g. duplicate the functionality of basic logic gates such as AND, OR, XOR, NOT or moderately complex combinational functions such as decoders or simple mathematical functions. The logic cell architecture varies between different device families. Generally speaking, each logic cell combines a few binary inputs to one or two outputs according to a boolean logic function specified in the user program. In most families, the user also has the option of registering the combinatorial output of the cell, so that clocked logic can be easily implemented. The cell's

combinatorial logic may be physically implemented as a small look-up table memory (LUT) or as a set of multiplexers and gates. LUT devices tend to be a bit more flexible and provide more inputs per cell than multiplexer cells at the expense of propagation delay. As an example, Figure 3.1 shows a high level diagram of a Virtex-II Pro slice, which are organized into two different logic blocks.

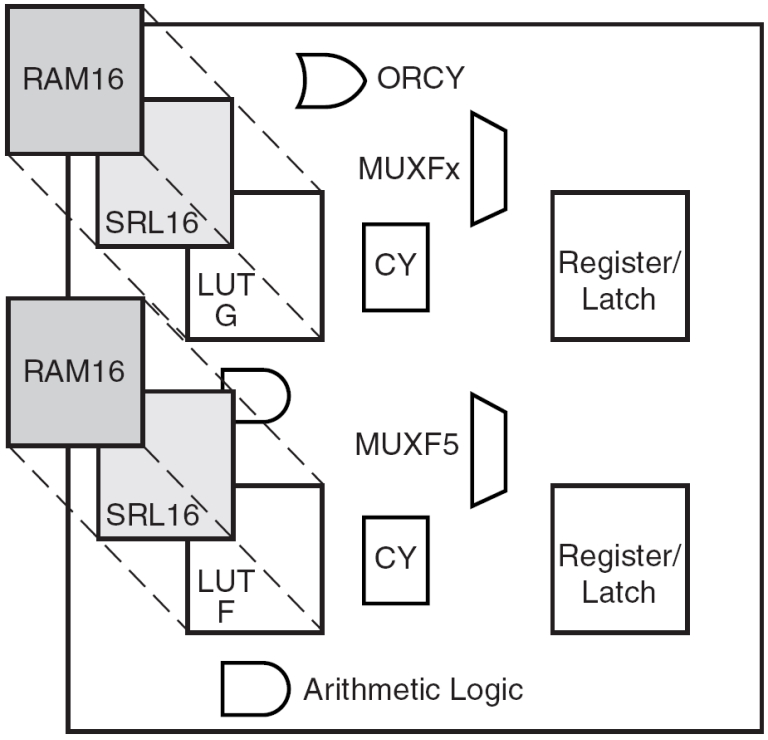


Figure 3.1: Virtex-II Pro Slice configuration

Each slice includes two 4-input function generators, carry logic, arithmetic logic gates, wide function multiplexers and two storage elements. As shown in Figure 3.1, each 4-input function generator is programmable as a 4-input LUT, 16 bits of distributed memory, or a 16-bit variable- tap shift register element. The output from the function generator in each slice drives both the slice output and the D input of the storage element.

The delay of a circuit implemented in an FPGA is mostly due to routing delays, rather than logic block delays, and a significant part of an FPGA's area is devoted to programmable routing. A hierarchy of programmable interconnects allows the logic blocks of an FPGA to be interconnected as needed by the system designer, somewhat like a one-chip programmable breadboard.

Like the logic blocks, these interconnects can be programmed after the

manufacturing process by the designer (hence the term "field programmable", i.e. programmable in the field) so that the FPGA can perform whatever logical function is needed, only restricted by the capacity.

Traditionally, the FPGA routing is unsegmented. That is, each wiring segment spans only one logic block before it terminates in a switch box. By turning on some of the programmable switches within a switch box, longer paths can be constructed. For higher speed interconnect, recent FPGA architectures use longer routing lines that span multiple logic blocks.

Figure 3.2 shows the routing resources available in a Virtex-II Pro FPGA, the FPGA used in this work.

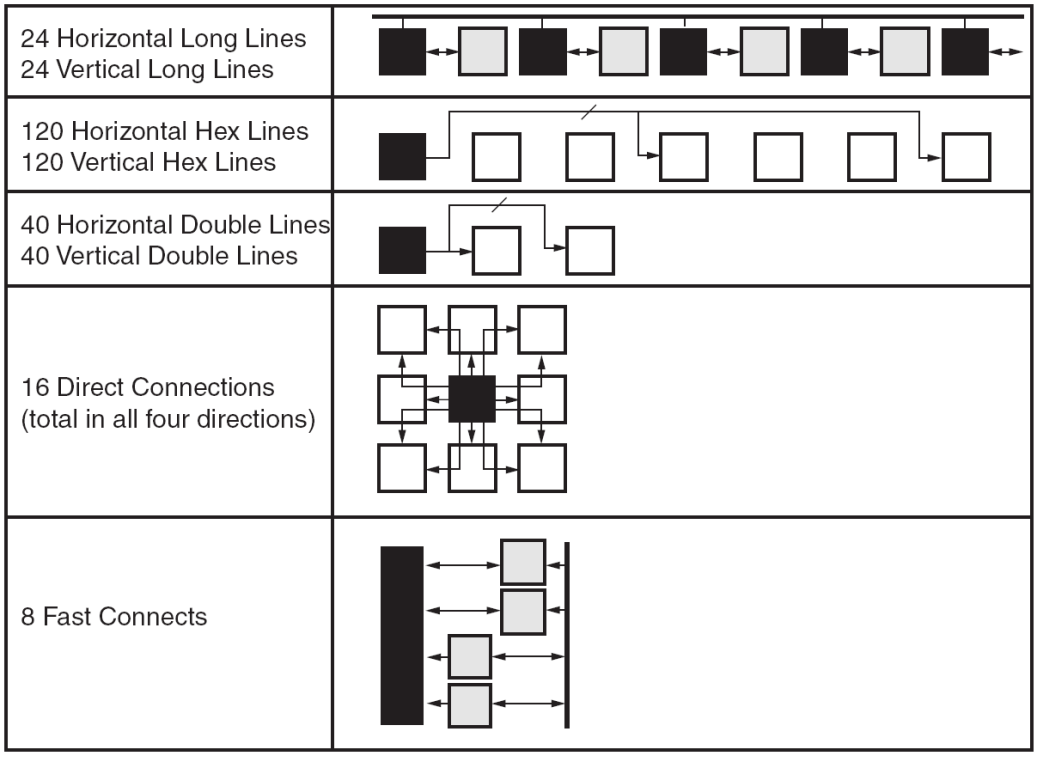


Figure 3.2: Virtex-II Pro Routing Resources

The long lines are bidirectional wires that distribute signals across the device. Vertical and horizontal long lines span the full height and width of the device. Hex lines, double lines and direct connect lines route signals to the neighboring blocks up to three, two and one blocks away from the source, respectively. The fast connect lines are the internal CLB local interconnections from LUT outputs to LUT inputs. Clock signals (and often other high-fanout signals) are normally routed via special-purpose dedicated routing networks in commercial FPGAs.

Modern FPGA families expand upon the above capabilities to include higher level functionality fixed into the silicon. Having these common functions embedded into the silicon reduces the area required and gives those functions increased speed compared to the alternative of building them from primitives. Examples of these include multipliers, generic DSP blocks, embedded processors, high speed IO logic and embedded memories.

Currently, the two largest players in the FPGA market are Xilinx and Altera. Altera's and Xilinx's basic building block contains a 4-input Look-up Table (LUT), a flip-flop and some additional circuitry. Altera calls this block a Logic Element (LE) while Xilinx calls it a Logic Cell (LC).

Altera low cost product line is the Cyclone series. While optimized for the smallest possible die size these devices offer up to 120k LEs (approximately 120k LUTs) and up to 4Mb of block memories, embedded $18 \times 18$ multipliers, Phase Locked Loops (PLLs) and external memory interfaces. As for Xilinx, the low cost product line is the Spartan series. This device family offers up to 75k LCs (approximately 150k LUTs) and up to 2Mb of block memories. Besides $18 \times 18$ Multipliers and PLLs, this family offers five platforms that include resources aimed at different applications rather than one device that combines all characteristics – Digital Signal Processings (DSPs), Non volatile memory, I/O bias, high density and pin count, logic bias.

At the high end range, Altera offers the Stratix series, which includes devices with up to 338k LE and up to 20Mb memory. These devices improve on the embedded features of the low cost family by including high-speed DSP blocks. Xilinx's high end devices are found in the Virtex family. These include embedded features such as hardware PowerPC405 cores, DSP slices and ethernet Medium Access Control / Physical layer interfaces (MAC/PHYs) units. A short summary of the main features of the high end series of both Altera and Xilinx is present on Table 3.1.

Table 3.1: FPGA features

| **Device** | **Process** | **LUTs** | **Bram (kb)** | **Other features** |
|---|---|---|---|---|
| **Virtex-II Pro** | 135 nm | 2.8k-88k | 216-8k | PPC405 Core |
| **Virtex 4** | 90 nm | 12.2k-178k | 648-9.9k | PPC405 Core or DSP Blocks |
| **Virtex 5** | 65 nm | 19.2k-207k | 1.2k-11.7k | DSP Blocks, 6-input LUTs |
| **Stratix** | 135 nm | 10.5k-79k | 898-7.2k | DSP Blocks |
| **Stratix II** | 90 nm | 15.6k-179k | 409-9.1k | DSP Blocks |
| **Stratix III** | 65 nm | 48k-338k | 1.8k-16.2k | DSP Blocks |

## 3.2 ML310 Hardware Characteristics

The Xilinx ML310 Embedded Development Platform based on a Virtex-II Pro XC2VP30-FF896 FPGA is suited for rapid prototyping and system verification.

In addition to the more than 30,000 logic cells, over 2,400 Kb of Block RAM (BRAM), dual PowerPC 405 processors and RocketIO transceivers available in the FPGA, the ML310 provides an on-board Ethernet MAC/PHY, Double Data Rate (DDR) memory, multiple PCI bus slots, and standard PC I/O ports within an ATX form factor board. An integrated System ACE Compact Flash (CF) controller is deployed to perform board bring-up and to load applications from the included 512 MB CompactFlash card.

The main resources provided by the ML310 board are:

- XC2VP30-FF896-6 with dual PPC405 cores;
- 256 MB DDR DIMM Memory;
- System ACE CF Controller;
- 512 MB CompactFlash card;
- Intel Ethernet/NIC PCI Device;
- 4 PCI slots (3.3V and 5V);
- LCD character display and cable;
- FPGA serial port connection;
- Personality module interface for RocketIO and LVDS access;
- Standard JTAG connectivity;
- ALi Super I/O (M1535D+ PCI Device);
  - 1 parallel and 2 serial ports;
  - 2 Universal Serial Bus (USB) ports;
  - 2 Integrated Drive Electronics (IDE) connectors.

Figure 3.3 shows an high-level block diagram of the ML310 board and the connections between the different hardware blocks. As it can be seen on this, the Virtex-II Pro FPGA is connected to the different peripherals either directly or indirectly via the PCI Bus. Each connection on the FPGA requires the implementation of an interface core on the FPGA logic cells. Table 3.2 lists the required core for the most relevant connections, which are provided as Intelectual Property (IP) cores. Table 3.3 shows a summary of the hardware available on the XC2VP30-FF896 FPGA that is used to implement the different IP cores and user modules.

The PowerPC hard cores on the Virtex FPGA allow the implementation of complex hardware/software systems and System-on-a-Chip (SoC) designs. They also provide a convenient way of running test applications on hardware

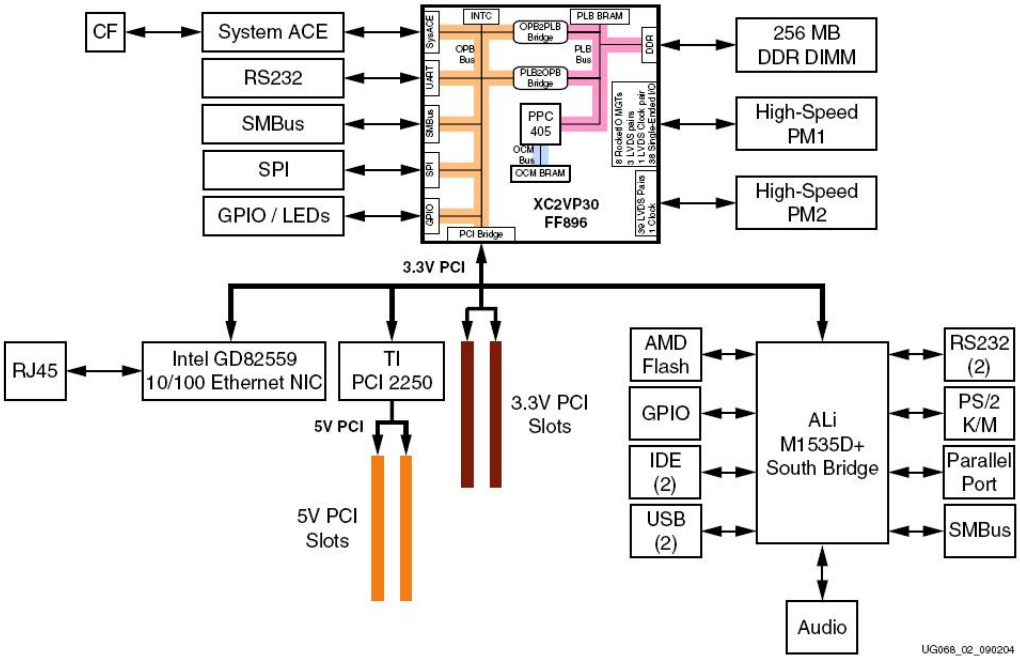


Figure 3.3: ML310 High-Level Block Diagram

Table 3.2: ML310 Peripheral Connection

| **Peripheral** | **Connection** |
|---|---|
| DDR DIMM Memory | DDR |
| FPGA RS232 | UART |
| System ACE | SysACE |
| LEDs / LCD | GPIO |
| PCI Bus Interface | PCI Bridge |
| ALi M1535D+ PCI Device | |
| Intel Ethernet/NIC PCI Device | |
| SMBus/IIC | SMBus |

Table 3.3: Virtex-II Pro (XC2VP30-FF896) Specifications

| **Component** | **Amount available** |
|---|---|
| Logic Cells | 30,816 |
| PPC405 | 2 |
| MGTs | 8 |
| BRAM (Kbits) | 2,448 |
| Xtreme Multipliers | 136 |

modules implemented on the FPGA fabric. The embedded PPC405 core is a 32-bit Harvard architecture processor with low power consumption (0.9 mW/MHz). Figure 3.4 presents a block diagram of this processor core, which includes:

- Cache units;
- Memory Management unit;
- Fetch Decode unit;
- Execution unit;
- Timers;
- Debug logic unit.

The processor operates on instructions in a five pipeline stages, the fetch, the decode, the execute, the write-back, and the load write-back stages. Most of the instructions execute in a single cycle, including loads and stores.

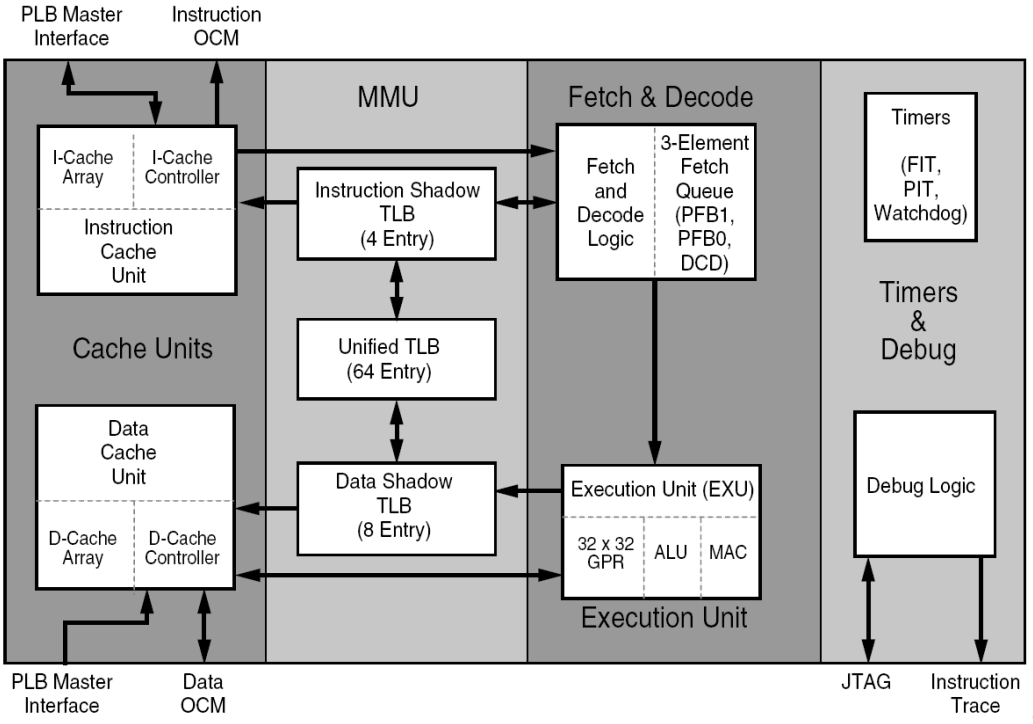


Figure 3.4: PowerPC 405 Block Diagram

The processor's features include:

- Thirty-two 32-bit general purpose registers;
- 16 KB two-way set-associative instruction cache;

- 16 KB two-way set-associative data cache;
- Memory Management Unit (MMU);
- 64-entry unified Translation Look-aside Buffers (TLB);
- Variable page sizes (1 KB to 16 MB);
- Dedicated on-chip memory (OCM) interface;
- Hardware multiply/divide unit;
- Supports IBM CoreConnect bus architecture.

Typically, a SoC contains numerous functional blocks requiring a huge number of logic gates. Such designs are best realized in a macro-based approach. Macro based design provides numerous benefits during design and verification phases, but the ability to reuse intellectual property is often the most significant. From generic serial ports to complex memory controllers and processor cores, each SoC generally requires the use of common macros. For promoting reuse, macro interconnectivity is accomplished by using common buses for inter-macro communications.

The IBM CoreConnect architecture provides three buses for interconnecting cores, library macros, and custom logic:

- Processor Local Bus (PLB);
- On-Chip Peripheral Bus (OPB);
- Device Control Register (DCR) Bus.

Figure 3.5 illustrates how the CoreConnect architecture is used to interconnect macros on the Virtex-II Pro.

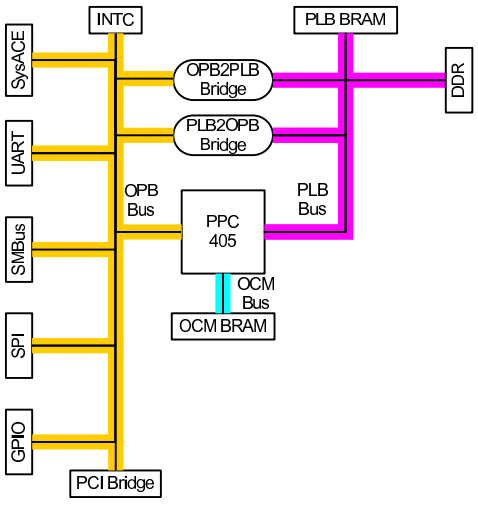


Figure 3.5: CoreConnect Buses on the Virtex-II Pro

The PLB and OPB buses provide the primary means of data flow among macro elements. Because these two buses have different structures and control signals, individual macros are designed to interface to either the PLB or the OPB.

The prime goal of PLB is to interconnect high-bandwidth devices such as processor cores, external memory interfaces and DMA controllers, which are the main producers and consumers of the bus transaction traffic. The PLB addresses the high performance, low latency and design flexibility issues needed in a highly integrated SOC through:

- decoupled address, read data, and write data buses with split transaction capability;
- concurrent read and write transfers yielding a maximum bus utilization of two data transfers per clock;
- address pipelining that reduces bus latency by overlapping a new write request with an ongoing write transfer and up to three read requests with an ongoing read transfer;
- ability to overlap the bus request/grant protocol with an ongoing transfer.

Moreover, the PLB offers designers flexibility based on the following features:

- support for both multiple masters and slaves;
- four priority levels for master requests allowing PLB implementations with various arbitration schemes;
- deadlock avoidance through slave forced PLB rearbitration;
- master driven atomic operations through a bus arbitration locking mechanism;
- byte-enable capability, supporting unaligned transfers;
- a sequential burst protocol allowing byte, half-word, word and double-word burst transfers;
- support for 16-, 32- and 64-byte line data transfers;
- read word address capability, allowing slaves to return line data either sequentially or target word first;
- DMA support for buffered, fly-by, peripheral-to-memory, memory-to-peripheral, and memory-to memory transfers;
- guarded or unguarded memory transfers allow slaves to individually enable or disable prefetching of instructions or data;
- slave error reporting;
- architecture extendible to 256-bit data buses;

- fully synchronous.

Table 3.4 shows the performance of the PLB bus, which can offer a maximum bandwidth of 2.9 GB/s.

Table 3.4: PLB bus performance; * estimated

| | **CoreConnect 32** | **CoreConnect 64** | **CoreConnect 128** |
|---|---|---|---|
| **PLB width** | 32-Bit | 64-Bit | 128-Bit |
| **Max Frequency** | 66 MHz | 133 MHz | 183 MHz* |
| **Max Bandwidth** | 264 MB/s | 800 MB/s | 2.9 GB/s* |

The On-Chip Peripheral Bus (OPB) is a secondary bus designed to alleviate system performance bottlenecks by reducing capacitive loading on the PLB. Peripherals suitable for attachment to the OPB include serial ports, parallel ports, UARTs, GPIO, timers and other low-bandwidth devices. The OPB provides the following features:

- a fully synchronous protocol with separate 32-bit address and data buses;
- dynamic bus sizing to support byte, half-word and word transfers;
- byte and half-word duplication for byte and half-word transfers;
- a sequential address (burst) protocol;
- support for multiple OPB bus masters;
- bus parking for reduced-latency transfers.

The DCR (Device Control Register) Bus is used for removing device configuration slave loads, memory address resource usage and configuration transaction traffic from the main system busses. Most traffic on the DCR bus occurs during the system initialization period, however, some elements such as the Direct Memory Access (DMA) controller and interrupt controller cores use the DCR bus to access functional registers during its operation.

## 3.3 Software Overview

The development environment makes use of several software tools:

**Xilinx Foundation ISE** – ISE Foundation is a complete programmable logic design environment. It includes timing driven implementation tools available for programmable logic design, along with design entry, synthesis and verification capabilities.

**Design Entry** – ISE provides support HDL and schematic entry, integration of IP cores as well as robust support for IP re-usage.

**Synthesis** – Synthesis is one of the most essential steps in the design methodology. It takes the conceptual Hardware Description Language (HDL) design and generates the logical or physical representation for the targeted silicon device. ISE includes Xilinx proprietary synthesis technology, XST.

**Implementation and Configuration** – Programmable logic design implementation assigns the logic created during design entry and synthesis into specific physical resources of the target device. The term "place and route" has historically been used to describe the implementation process for FPGA devices and "fitting" has been used for CPLDs. Implementation is followed by device configuration, where a bitstream is generated from the physical place and route information and downloaded into the target programmable logic device.

**Xilinx EDK/XPS** – The Embedded Development Kit (EDK) bundle is an integrated software solution for designing embedded processing systems. This pre-configured kit includes the Xilinx Platform Studio (XPS) tool suite as well as all the documentation and IP required for designing Xilinx Platform FPGAs with embedded PowerPC hard processor cores and/or MicroBlaze soft processor cores.

**Mentor Graphics Modelsim** – ModelSim is a complete PC HDL simulation environment that enables the verification of HDL source code and functional and timing models of designs.

## 3.4 Base System Configuration

The ML310 is supplied with a working Linux system by means of an Advanced Configuration Environment (ACE) file and a partition on the provided flashcard. The ACE file configures the FPGA hardware and programs the processor's BRAMs with the kernel image. However, in order to allow for custom hardware definition and kernel configuration, the default ACE file was not used in this project.

The base hardware configuration was performed using the reference design that includes the PCI bus supplied by Xilinx. This design configures the FPGA in order to implement several Xilinx IP cores that are used by the Linux system. This is a XPS design, and it produces the bitstream (download.bit) file that describes the hardware configuration on the FPGA

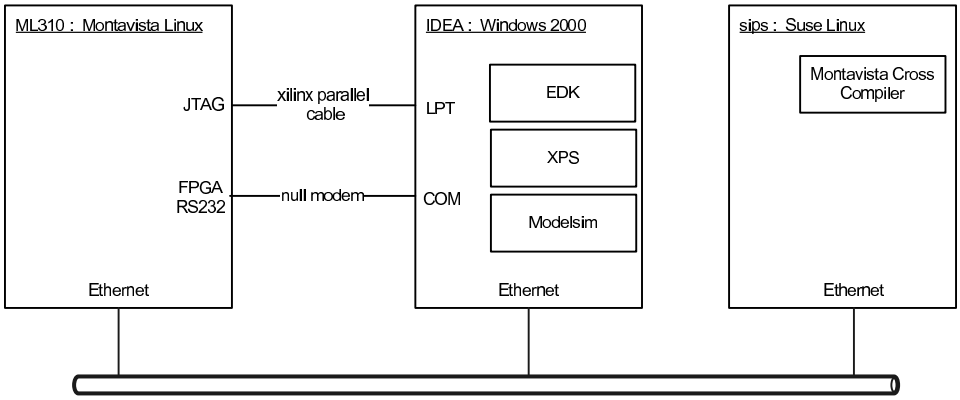


Figure 3.6: Computer platform used for development.

and processors' BRAM initialization data. Bit files are Xilinx FPGA configuration files generated by the Xilinx FPGA design software. They are proprietary format binary files containing configuration information.

The software initialization file used was a customized Linux kernel. The partition on the flash card was also replaced by a hard drive so that the Linux system has more disk space available [23].

Because the software development was not performed natively on the ML310, a cross compiler was required. This tool is able to generate binary code for a different platform than the one it is running on. The Montavista Linux Preview Kit for Professional Edition 3.1 was used as the cross compiler system on this work. This free version has limited functionality and integration and is only available for Linux platforms.

The Montavista Linux Preview Kit for Professional Edition 3.1 provides a kernel source tree that has been patched for PowerPC processors and is appropriate for the ML310. These sources were used to create the custom kernel, as described on [23].

The steps required in order to configure the prototype board's hardware resources into a full computing system running a Linux operating system are described in further detail in Annex B.

Figure 3.6 displays the arrangement of the computers on the development platform. The ML310 is connected by the Xilinx Parallel cable to a Windows 2000 computer where the Xilinx and Modelsim tools reside, while the cross compiler was installed on a Linux computer. All the computers are interconnected by an Ethernet network.

## 3.5 Base Peripheral Module

The main objective of this work is to implement hybrid systems that are composed of software running on the PowerPC and application specific hardware implemented in the reconfigurable part of the FPGA.

A generic testbench was created as a PLB peripheral module [24] in order to integrate it in the CoreConnect architecture. This task was greatly simplified by using the design tools' capability to generate the logic responsible for interfacing the Processor Local Bus (PLB) bus: the PLB IP Interface (IPIF) [25]. This is a logic block that implements the basic bus interfacing protocol as well as more advanced features, such as DMA and IRQ. The IPIF takes care of all the communications with the PLB bus and provides a simple set of signals to the user logic. Figure 3.7 shows the simplified block diagram of the peripheral module and the connection to the PLB. The created module contains two single port 32KB memories (AR0 and AR1) and sixteen 64-bit registers that can be used to control user defined hardware units (e.g. a motion estimator).

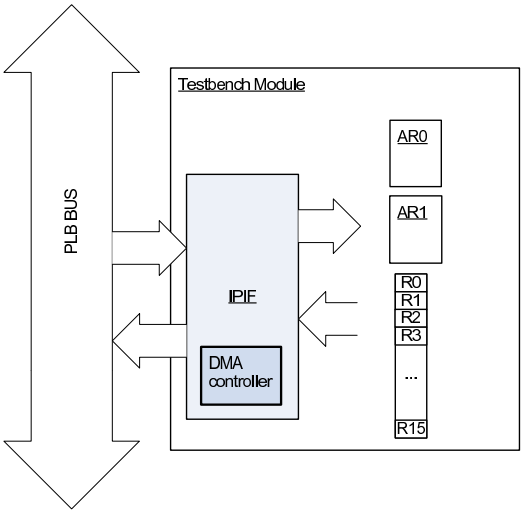


Figure 3.7: Block Diagram of the Base Testbench Peripheral.

The created IPIF includes a DMA master subsystem that can be used to significantly improve the speed and the autonomy of data transfers that involve the testbench module's memories. The DMA subsystem is controlled by a set of registers on the hardware module. A summary description of this registers is presented on Table 3.5.

The DMACR register controls the type of transaction that is performed by selecting the source and destination locations and address increment be-

Table 3.5: DMA Control Registers Summary

| **Register** | **Bit(s)** | **Description** |
|---|---|---|
| DMACR | | DMA/SG Control Register |
| | 0 | Source address increment (SINC flag) |
| | 1 | Destination address increment (DINC flag) |
| | 2 | Source is local (SLOCAL flag) |
| | 3 | Destination is local (DLOCAL flag) |
| SA | | Source Address Register |
| | 0-31 | Source Address |
| DA | | Destination Address Register |
| | 0-31 | Destination Address |
| LENGTH | | Length Register |
| | 0-31 | Transfer length in bytes |

haviour. The SA and DA registers hold the bus addresses for the source and destination, respectively. The LENGTH register defines the number of bytes to be transfered. Writing a value different from zero to this register starts the transfer.

For example, in order to perform a transfer of $L$ bytes from $S$ into $D$, assuming $S$ corresponds to a memory address on the testbench module and $D$ a memory address external to the module, the steps shown in Fig 3.8 are required. The SINC,DINC and SLOCAL flags are set and DLOCAL flag

---

$DMACR \Leftarrow X$"$e0000000''$ {SINC=1; DINC=1; SLOCAL=1; DLOCAL=0}
$SA \Leftarrow S$ {program source address}
$DA \Leftarrow D$ {program destination address}
$LENGTH \Leftarrow L$ {start transfer by setting transfer length}

---

Figure 3.8: Steps needed to perform a DMA transfer.

is reset (DMACR=x"e0000000"). SA is loaded with $S$ and DA registers is loaded with $D$. The transfer is started by writing $L$ to the LENGTH register.

On the implemented system, the reconfigurable part of the FPGA contains all the peripheral modules that are usually present on a general purpose PC (Dram controller, DMA controller, Interrupt controller, network controller, etc) besides the testbench module. One of the PowerPC processors is then used to run a Montavista Linux system, that provides a powerful

filesystem and network access. It can be used to provide input and output to the target applications.

## 3.6 Linux Device Driver

In order to allow the usage of the testbench module by the Linux system and applications a device driver is required, which provides access to the resources present on the module (namely registers and memories).

The device driver is the part of the Linux kernel that is responsible for controlling the hardware device and for hiding the details of how the device works so it also creates an abstraction layer that simplifies the development of complex systems.

Each driver usually extends the kernel with a set of operations that can be performed on the hardware to which it is associated. The set of operations that can be chosen depends on the class of the driver. There are several driver classes, but the most common are:

**Character Devices** – can mostly be thought of as regular files, and usually define at least the "open", "close", "read" and "write" system calls.

**Block Devices** – can host a filesystem and define, at least, the "mount" system call.

**Network Interfaces** – are in charge of sending and receiving data packets and instead of "read" and "write" implement system calls related to packet transmission.

The testbench module device driver developed belongs to the character device because it is well suited to the reading and writing to the registers and memories.

Linux device drivers can take two forms: monolithically built into the kernel or loadable modules. Loadable modules have the advantage of not requiring a complete kernel recompilation every time a change is made to the driver and also provide the possibility of being loaded and unloaded at run time.

When the device driver is loaded the "init_module()" function is evoked. This function is responsible for allocating all the resources needed by the driver. These resources include the physical addresses for the hardware in question, the character device number that will be associated to the device and memory buffers that the driver needs.

The address resources are allocated by using the request_mem_region() and ioremap() functions. The first one reserves the required physical address range for use with the module, and the second remaps the physical addresses

into the virtual space. The init_module() function is also responsible for registering the set of capabilities that the driver implements. This is accomplished by setting up a file_operations structure which holds function pointers to the capabilities that are defined for this driver. In the implemented driver the defined capabilities are the "read" and the "write" functions. All other operations are kept at the kernel default for the class character device driver class. The register_chardev() function registers the file_operations structure on the kernel and assigns a major device to the driver, so it can be accessed by a char device file. Figure 3.9 shows an overview of the driver operation. Each time an application performs a FILE operation on the device files the kernel evokes the corresponding file function that is listed on the registered file_operations structure.

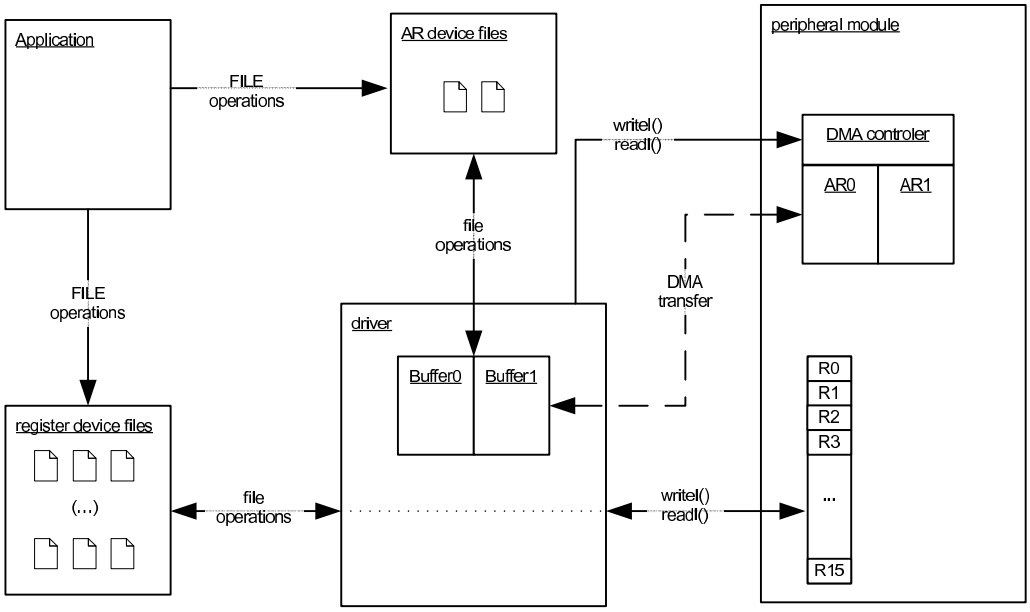


Figure 3.9: Driver Block Diagram

For the read operation, the data must be read from the remapped memory (which represents a hardware resource) into kernel memory space using the readl() [1] and then copied to user space using the copy_to_user() function.

The write function does the reverse, i.e., the data is copied from user space into kernel space by the copy_from_user() function and then written into the remapped memory using the writel() function.

The behaviour of the transfer functions is different according to the type

[1]Because the PowerPC system is big endian, the actual functions used are _raw_readl() and _raw_writel().

of resource being accessed (register or memory space). On transfers that involve the testbench memories, the readl() and writel() functions are replaced by calls to the DMA subsystem which is able to unattendedly transfer a contiguous amount of data between the peripheral address ranges and main memory.

For write operations, the data is transfered from the user application to a DMA capable buffer and then the DMA is started. For read operations, the data is transfered by DMA to the buffer and then passed to the user application. Because of the memory management of the Linux system, a buffer of contiguous virtual addresses (e.g. the result of a malloc() operation ) may actually be mapped into non contiguous physical addresses. Since DMA transfers require contiguous physical addresses, the pci_alloc_consistent() call must be used ([26].

In order to start the DMA transfers, the driver must program the IPIF DMA control registers with the proper values as depicted in section 3.5.

Using the presented platform, in the next chapter we propose the reconfigurable architecture for software/hardware motion estimation and in chapter 5 a complete video coder is implemented in this platform.

# Chapter 4

# Proposed Reconfigurable Architecture for Motion Estimation

The various fast search algorithms that have been proposed for motion estimation on video sequences exploit the statistical characteristics of the movement in order to reduce the searched space. The processing in these search algorithms is data dependent (see section 2.0.2) and the relative performance depends on the images. Therefore it is advantageous to have a programmable or configurable processor that can adapt the search procedure to the image sequence characteristics. In this chapter a new configurable architecture for fast search motion estimation is proposed. This proposal has been published in the international conference on Field Programmable Logic and Applications (FPL) 2007 [27].

## 4.1 Search Algorithm Configuration Mechanism

By analyzing the characteristics of the fast search algorithms for video motion estimation, it can be observed that they are iterative and their evolution through the various iterations depend on the distances measured in run-time.

Most of these algorithms are regular, and are comprised of separate steps. On each step a number of possible positions is analyzed and the one that yields the smallest error is chosen as seed for the next searching step.

The main idea behind the proposed run-time configuration mechanism is to model this behaviour through the use of simple data structures associated

with specific hardware resources

These data structures must contain the information required to implement the mechanism, namely:

- **cartesian displacements** ($\Delta x$ and $\Delta y$) – values accumulated in each step for producing the output motion vector;
- **raster displacement** – This precomputed value provides the address offset within a frame that corresponds to the displacement of the current MV. Although redundant, it provides a significant reduction in the required hardware.
- **next pattern address** - used to select the initial MB every time a new search step begins;
- **step end** - flag signaling the end of the current search step, causing the loading of the address corresponding with the MB with smallest error;
- **search end** - flag signaling the end of the search process.

Fig. 4.1 depicts the mechanism for searching the "best" candidate block for a given MB. Under normal operation, the system starts by fetching a pattern from the pattern memory. The raster displacement information on the pattern is then used to compute the base address for this candidate block. The candidate block is then fetched from the frame memory and the SAD is computed for the current and the candidate MBs. This SAD value is then compared to the stored smallest SAD value computed so far. If the current SAD value is smaller than the stored one, the candidate macroblock becomes the best known match, which causes the SAD output to be stored in the smallest-SAD register and the pattern information to be stored in the smallest-SAD-pattern register. This procedure is successively repeated until the step end flag is set on. When this flag is set, the values stored on the smallest-SAD-pattern register are used to start the next search step. This recenters the search process on the candidate MB that yielded the smallest SAD and selects the pattern search corresponding to the next search step.

It should be noted that from the current step, different sets of patterns will be used for the subsequent step depending on the pattern that is selected. This is exemplified on the Four Step Search Algorithm implementation on A.2.

The step end flag also causes the smallest SAD pattern's cartesian displacements to be accumulated on the motion vector output accumulators. At any given time, the output motion vectors accumulators hold the sum of the cartesian displacements of the points that produced the smallest SAD,

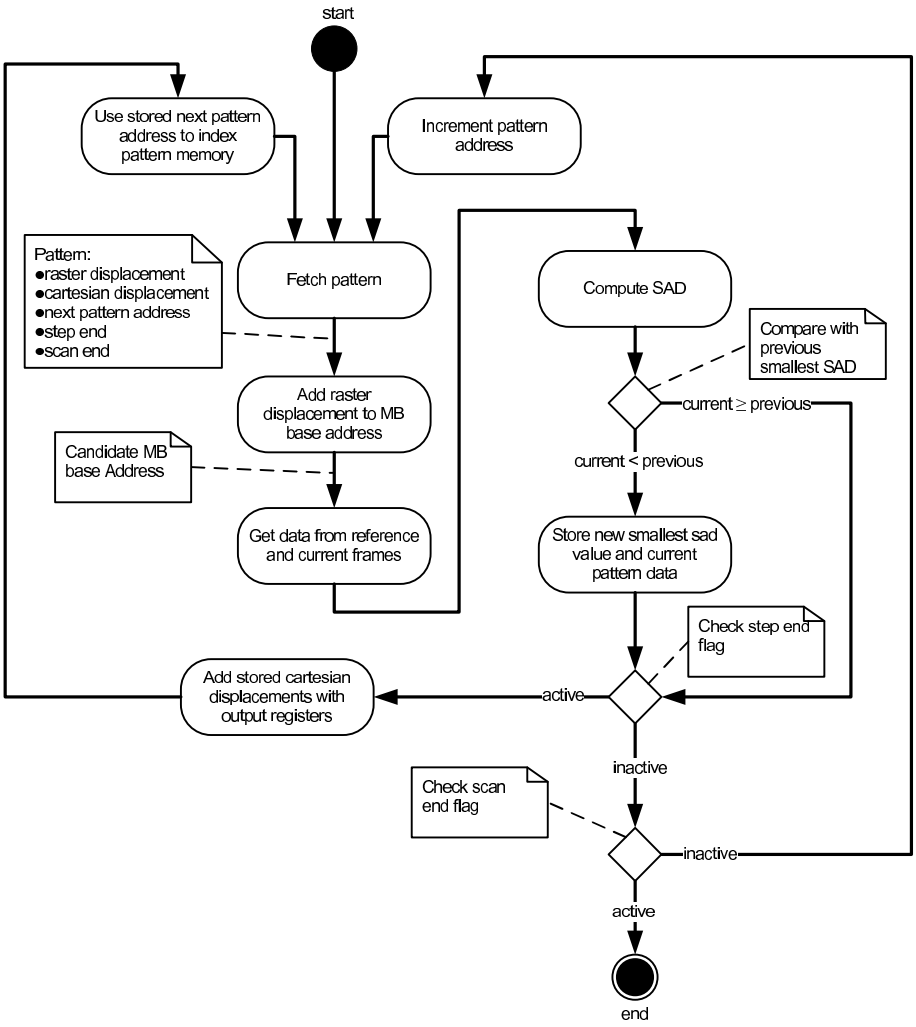


Figure 4.1: flowchart for the search process

for each of the previous steps:

$$MV = \sum_{s=1}^{N} (MV(P_s^I) | SAD(P_s^I) <= SAD(P_s^i))(0 \leq i \leq n_s) \qquad (4.1)$$

where;
$N$ - number of search steps;
$MV(P)$ - the cartesian displacements on the pattern P;
$P_s^i$ - the $i$th pattern of the $s$th search step;
$n_s$ -the number of candidate MBs checked on the $s$th search step.

Fig. 4.2 shows a simple example search that tests 9 candidate blocks in 3 steps, while table 4.1 shows the input data structure needed in order to implement the standard 3SS and Diamond standard algorithms, which test 25 and a number candidate MBs that depends on the actual data on the search area, respectively. The conditional jumps programmed in the data structure allow for a section to be reused, thus keeping the overall size requirements for the data structure small, while implementing fairly complex algorithms.

a)

| PA | ScE | StE | NPA | Δx | Δy |
|---|---|---|---|---|---|
| 0 | | | 4 | 0 | 0 |
| 1 | | | 4 | -9 | -9 |
| 2 | | | 4 | 9 | -9 |
| 3 | | 1 | 4 | 0 | 9 |
| 4 | | | 7 | 0 | -2 |
| 5 | | | 7 | -2 | 2 |
| 6 | | 1 | 7 | 2 | 2 |
| 7 | | | | -1 | -1 |
| 8 | 1 | | | 1 | 1 |

b)

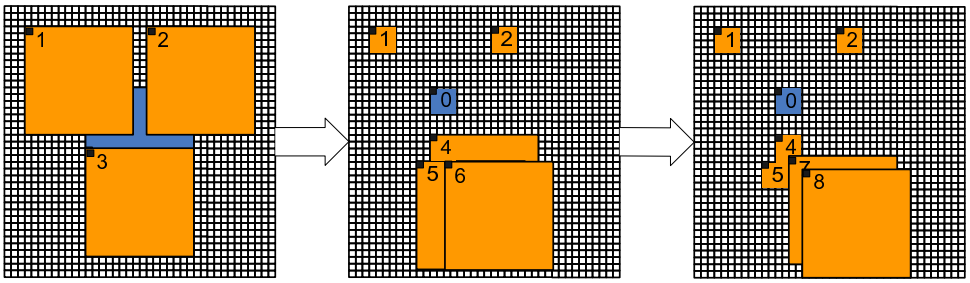


Figure 4.2: An example of a search process, assuming that pattern addresses 3 and 6 yield the smallest error for the corresponding search step: a)data structure; b)visual representation of the search process.

The processor is prepared to stop the search process when either one of the conditions is true:

- ss_changed=0 - at the end of a search step, if none of the step patterns produced a smaller SAD result than the one stored;
- sad_threshold - for every pattern, if the computed SAD result is smaller than the input threshold (not represented on the flowchart);
- step_limit - at the end of a search step, if the preset maximum number of steps is reached.

This behaviour can lead to some amount of speed gain, but will produce motion vectors that yield larger residual error. The ss_changed=0 condition can be disabled by inserting a pattern with cartesian displacements $[\Delta x, \Delta y] = [0, 0]$ on the beginning of each step. This extra pattern will introduce very little overhead because the previously computed SAD will be used.

The sad_threshold condition can be disabled by setting the input threshold to zero. The step_limit condition can be disabled by setting the number of steps to a large value. However this value should always be coherent with the current search algorithm.

Table 4.1: Data structure for the 3SS and diamond search algorithm: PA-Pattern address; SeE-Search End; StE-Step End; NPA-Next Pattern Address

| 3SS algorithm | | | | | | Diamond search algorithm | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| PA | **SeE** | **StE** | **NPA** | $\Delta x$ | $\Delta y$ | **SeE** | **StE** | **NPA** | $\Delta x$ | $\Delta y$ |
| 0 | | | 9 | 0 | 0 | | | 49 | 0 | 0 |
| 1 | | | 9 | -4 | -4 | | | 9 | 0 | -2 |
| 2 | | | 9 | 0 | -4 | | | 15 | 1 | -1 |
| 3 | | | 9 | 4 | -4 | | | 19 | 2 | 0 |
| 4 | | | 9 | -4 | 0 | | | 25 | 1 | 1 |
| 5 | | | 9 | 4 | 0 | | | 29 | 0 | 2 |
| 6 | | | 9 | -4 | 4 | | | 35 | -1 | 1 |
| 7 | | | 9 | 0 | 4 | | | 39 | -2 | 0 |
| 8 | | 1 | 9 | 4 | 4 | | 1 | 9 | -1 | -1 |
| 9 | | | 18 | 0 | 0 | | | 49 | 0 | 0 |
| 10 | | | 18 | -2 | -2 | | | 39 | -2 | 0 |
| 11 | | | 18 | 0 | -2 | | | 45 | -1 | -1 |
| 12 | | | 18 | 2 | -2 | | | 9 | 0 | -2 |
| 13 | | | 18 | -2 | 0 | | | 15 | 1 | -1 |
| 14 | | | 18 | 2 | 0 | | 1 | 19 | 2 | 0 |
| 15 | | | 18 | -2 | 2 | | | 49 | 0 | 0 |
| 16 | | | 18 | 0 | 2 | | | 9 | 0 | -2 |
| 17 | | 1 | 18 | 2 | 2 | | | 15 | 1 | -1 |
| 18 | | | | -1 | -1 | | 1 | 19 | 2 | 0 |
| 19 | | | | 0 | -1 | | | 49 | 0 | 0 |
| 20 | | | | 1 | -1 | | | 9 | 0 | -2 |
| 21 | | | | -1 | 0 | | | 15 | 1 | -1 |
| 22 | | | | 1 | 0 | | | 19 | 2 | 0 |
| 23 | | | | -1 | 1 | | | 25 | 1 | 1 |
| 24 | | | | 0 | 1 | | 1 | 29 | 0 | 2 |
| 25 | 1 | 1 | | 1 | 1 | | | 49 | 0 | 0 |
| ... | ... | | | | | ... | ... | ... | ... | ... |
| ... | | | | | | ... | ... | ... | ... | ... |
| 49 | | | | | | | | | 0 | -1 |
| 50 | | | | | | | | | 1 | 0 |
| 51 | | | | | | | | | 0 | 1 |
| 52 | | | | | | 1 | | | -1 | 0 |

## 4.2 Architecture for Motion Estimation

The motion estimator architecture can be represented by three main blocks (see fig. 4.3) connected to a frame memory.

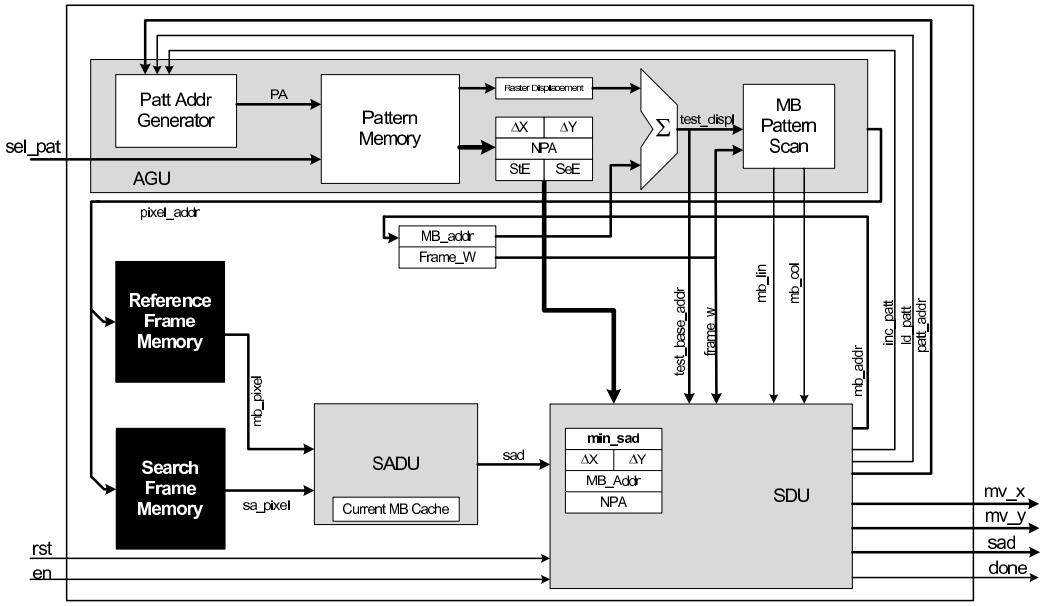


Figure 4.3: Top level block diagram of the motion estimation processor

The address generation unit (AGU) is responsible for generating the addresses required to access the frame memories. Because the pattern sets are stored on this unit, the AGU also outputs the active pattern information which is needed by the search decision unit in order to control the flow of the search process (see Figure 4.3).

The search decision unit is in charge of choosing the best candidate block for each search step. It contains the registers and the accumulators required to hold key values throughout the search process and an arithmetic comparator that is used to determine whether the current pattern produces a smaller SAD value than the preceding ones. It is also capable of determining whether the current search coordinates are valid for the current MB.

The SAD unit computes distance between pairs of MBs. It also contains a cache whose function is, besides caching the current MB, gathering and aligning the data from the candidate macroblock in a way that the SAD unit can process.

These units are described in further detail in the next subsections, considering MBs of $16 \times 16$ pixels.

### 4.2.1 Address Generation Unit (AGU)

The pattern entries described earlier are stored on a small dedicated memory located on this unit. Using the raster displacement information from each pattern, this unit is capable of determining all the pixel addresses for the respective MB.

The block diagram of the AGU is presented in fig. 4.4 where all the addresses are in raster format and the address to the memory is available in the addr_out port.

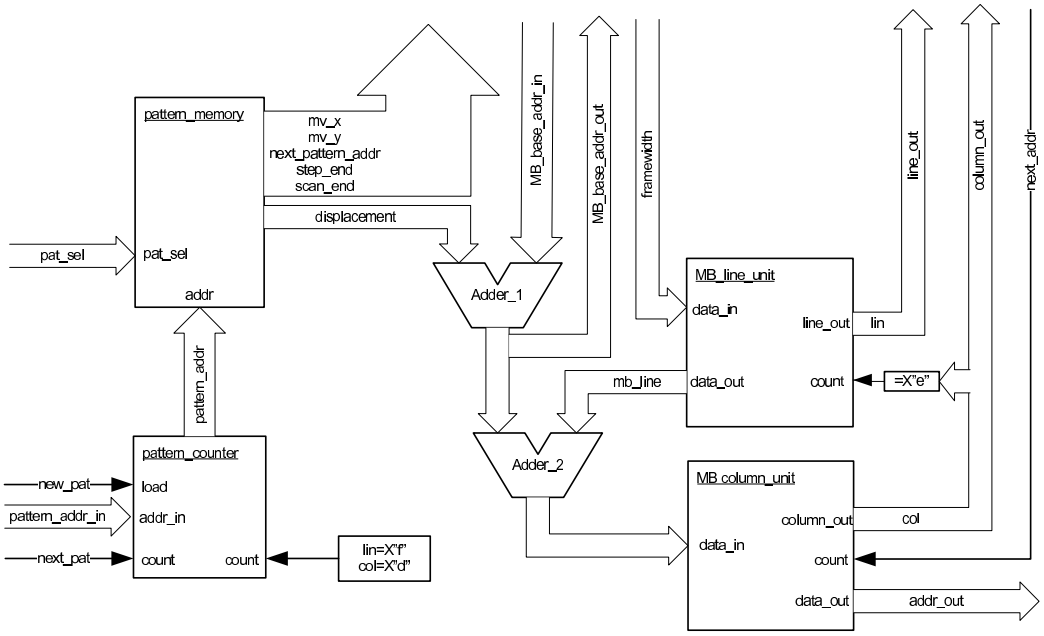


Figure 4.4: Block diagram of the AGU

For each match operation, the raster displacement information is added to the MB_base_addr input. The result is the address of the top left corner of the candidate block that will be currently used for estimation.

In order to generate all the addresses for the macroblock pixels, this value is then added to mb_line, which is the current output of the MB_line_unit, and then loaded and processed by MB_column_unit.

The MB_column_unit in fig. 4.4 is composed by two 4-bit counters in parallel, one with parallel load (column register), and the other a simple counter (column counter) that provides the MB column number, as seen in fig. 4.5. Initially the former counter is loaded with the output of adder_2 and the latter is reset. On each clock rising edge, both counters are incremented. Each time the column counter reaches the maximum value the parallel data

is loaded. Thus the horizontal scan of the macroblocks' lines is achieved according to table 4.2.

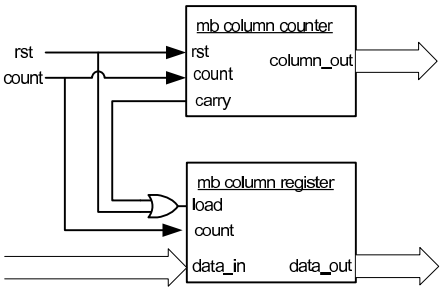


Figure 4.5: Block diagram of MB_column_unit

Table 4.2: MB_column_unit operation

| **rst** | **clk** | **count** | **load** | **column_out** | **column_out** | **data_out** |
|---|---|---|---|---|---|---|
| 1 | | | - | | 0 | data_in |
| 0 | ↑ | | 1 | | 0 | data_in |
| 0 | ↑ | 1 | | 15 | 0 | data_in |
| 0 | ↑ | 1 | | | +=1 | +=1 |
| 0 | | | | | hold | hold |

The MB_line_unit is similar to MB_column_unit, but the parallel load counter is replaced by a register and an adder (see fig. 4.6). Initially, the 4 bit counter (line counter) and the register are reset. The counter is incremented on each enabled rising edge clock and the data_in value (which is the framewidth) is added to the register. Thus, the output is the raster format address displacement that corresponds to a vertical raster displacement of line_out rows for a frame with a width of line_in.

Table 4.3 summarizes this unit's behaviour.

Table 4.3: MB_line_unit behaviour

| **rst** | **clk** | **count** | **load** | **line_out** | **line_out** | **data_out** |
|---|---|---|---|---|---|---|
| 1 | | | | | 0 | data_in |
| 0 | ↑ | | 1 | | 0 | 0 |
| 0 | ↑ | 1 | | 15 | 0 | 0 |
| 0 | ↑ | 1 | | | +=1 | +=framewidth |
| 0 | | | | | hold | hold |

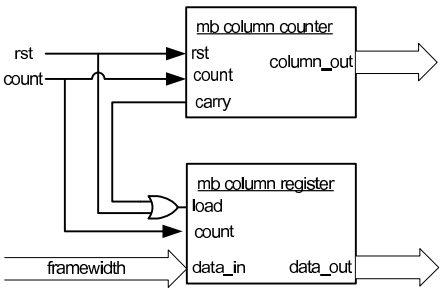


Figure 4.6: Block diagram of MB_line_unit

The pattern_counter is responsible for generating the addresses to index the pattern_memory. The pattern address is incremented each time line_out=X"f" and column_out=X"d", which is two clocks before the end of address generation for the current MB. Adopting this procedure the AGU is able continue outputting one new address per clock cycle for candidate blocks within a search step. The pattern address is also incremented when next_pat='1', which happens when an invalid pattern is detected. However, in this case the AGU will output invalid addresses for the next two clock cycles but there is logic on the search_decision_unit to handle this situation. Parallel loading is controlled by the new_pat signal that is activate when a search step ends and the next_pattern_address must be loaded. The pat_sel input signals are used to select the data structure corresponding to the selected search algorithm. Reserving a fix number of memory positions to each search algorithms and in particular one that is a power of two, the pat_sel input signals correspond to extra msb addressing bits.

### 4.2.2 Search Decision Unit

The search decision unit includes three distinct functionalities. The first one is to control and update the motion vector on each step, as can be seen in fig. 4.7. For each pattern, the accumulated motion vectors from the previous steps are added to the motion vectors of the current candidate MB. These values along with the base address and next pattern address are then stored on the "c_" registers until the SAD unit completes the computation.

When the SAD output is available it is compared to the smallest SAD value stored on the smallest_sad register (which is set to X"ffff" at the initial stage). If the current SAD output is smaller than the stored one, the smallest_sad register loads the new value and all the "ss_" registers load the values on the "c_" registers. When the system reaches the LOAD state the values on the "ss_mv_" registers are loaded into the "mv_" registers and a

Table 4.4: AGU I/O Signals

| Name | bits | Dir | Description |
|---|---|---|---|
| clk | 1 | in | clock input |
| rst | 1 | in | global reset, active high |
| next_addr | 1 | in | enable output of the next address (active high) |
| new_pat | 1 | in | load of new pattern (active high) |
| next_pat | 1 | in | load of next pattern (active high) |
| mb_x_max | 5 | in | maximum MB x coordinate |
| mb_x_max | 5 | in | maximum MB y coordinate |
| framewidth | 8 | in | framewidth in pixels |
| pattern_in | 32 | in | pattern input bus |
| pattern_addr_in | 7 | in | pattern input address |
| pattern_write | 1 | in | pattern write enable, active high |
| MB_base_addr_in | 20 | in | address of the top left pixel of current MB |
| motion_vector_x_out | 5 | out | motion vector output, x coordinate |
| motion_vector_x_out | 5 | out | motion vector output, y coordinate |
| step_end_out | 1 | out | end of a scanning step |
| scan_end_out | 1 | out | end of the motion estimation |
| next_pattern_addr_out | 7 | out | address of new pattern to load |
| MB_base_addr_out | 20 | out | |
| addr_out | 20 | out | |
| column_out | 4 | out | |
| line_out | 4 | out | |
| first_mb | 1 | out | |

new search step begins.

The second functionality is the detection of patterns that hold invalid motion vectors for the current macroblock due to the restrictions imposed by the image borders or if the resulting motion vector is larger than the allowed maximum. This is done by comparing the "base_mb_coordinate_" signals with the corresponding "mb_max_coordinate_" signal and with 0: For each of the MBs, if the one of it's coordinates is zero, the corresponding displacement can not be negative, which means that the most significant bit must be '0'. If the coordinate is equal to the maximum for the axis the coordinate can not be positive, so either the most significant bit must be '1' or the motion vector must be zero (see fig. 4.8). The motion vector size detection is performed by testing for overflow: the motion vector accumulator has an extra msb (totaling 6 bits) and the two msb are XORed to detect an error.

The third and last functionality is the control of the motion estimation operation based on a Finite State Machine (FSM). This FSM is composed

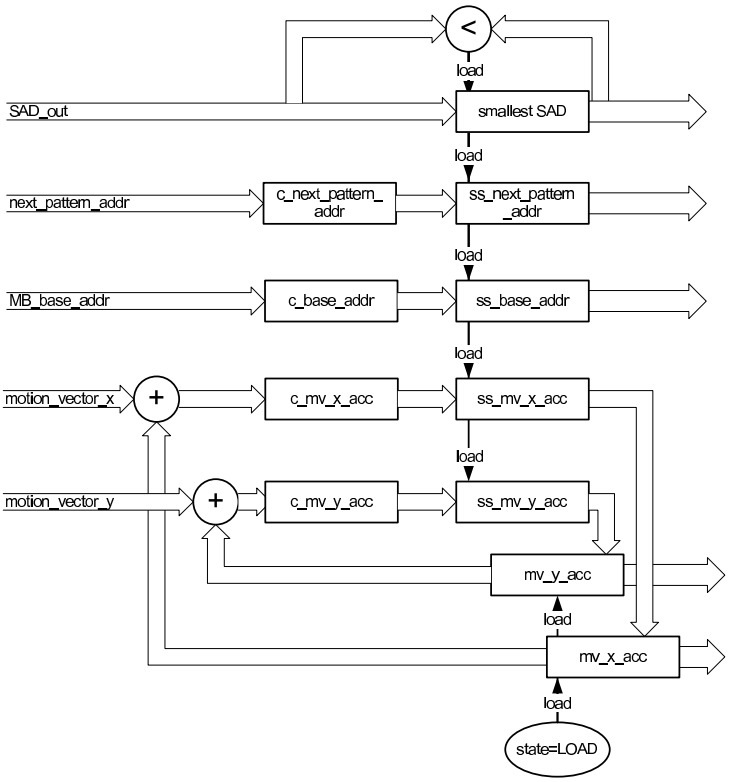


Figure 4.7: Search Decision Unit diagram

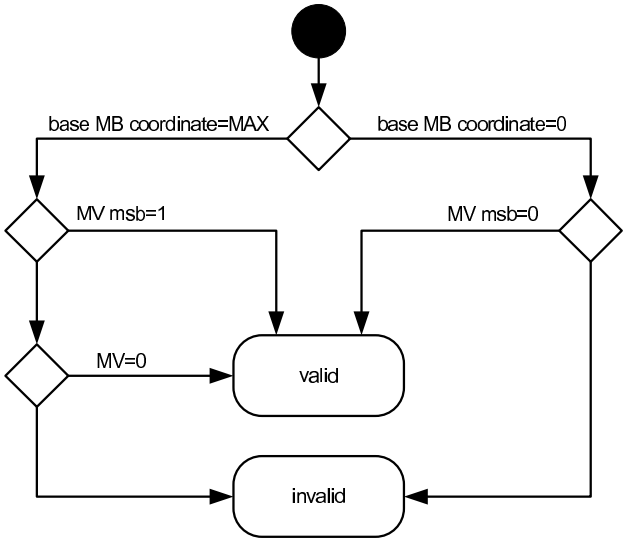


Figure 4.8: Invalid MV decision flowchart

by seven different states and was directly implemented in hardware, shown in Figure 4.9.

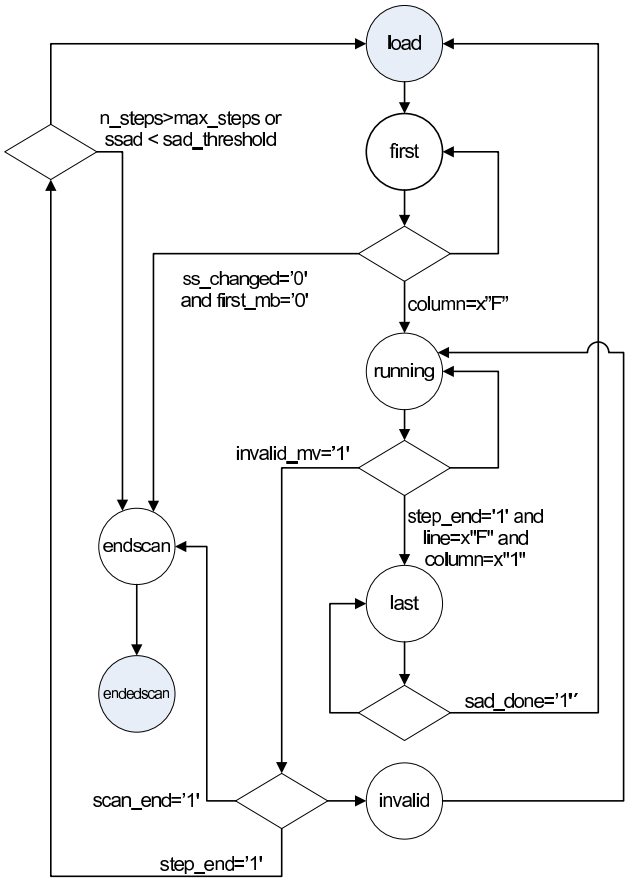


Figure 4.9: State Machine

The “load” state activates the new_pattern line on the AGU, which causes the AGU to load a new pattern and clear all the counters. It also causes the motion vector registers to load the currently accumulated motion vectors (Fig:4.7). Whenever the reset signal is activated, the system changes to this state (this transition is not represented on the figure). On the next clock cycle the state machine changes state to the “first“ state.

The ”first” state signals the first MB line of the first MB of the current search step. Due to the latency associated with loading a new pattern and getting the corresponding data from memory, during this state the data on the reference_mb_line and current_mb_unit is not yet valid to be used on the SAD block, as signaled by the ’0’ value on the mb_data_ready signal. After the first MB line the state is “running”.

During the “running“ state the system tries different patterns for the one yielding the smallest SAD. The state is maintained until the current pattern has an active step_end flag and the last MB line is reached, which causes the transition to the ”last“ state.

The system is in the ”last“ state until the SAD computation is completed, which is signaled by sad_done='1'. This happens a number of clocks after the last address of each MB is output by the AGU, because of the latency involved in accessing the frame memory, loading the output into the reference_mb_line and current_mb blocks and doing the actual SAD calculation. The ensuing state is ”load“.

The system changes to the ”endsearch“ state instead of the LAST state when the current pattern has the end_search flag raised. The (“ss_changed”='0' and “first_step”='0') transition to the ENDSCAN state implements the early finish mechanism in the case when none of patterns on the previous search steps produced a decrease on in the error measure. After the SAD calculation is finished the system changes to the ”endedsearch” state.

The “endedsearch“ state raises the motest_done signal, which signals the end of the motion estimation for the current MB.

The the invalid_mv signal causes the system to change to the ”invalid” state. This signal is generated as depicted on 4.8. The state activates the next_pat signal, which in turn causes the AGU to stop processing the current pattern, increment the pattern_counter and start using the new pattern.

### 4.2.3 SAD Unit

The arithmetic part of the SAD unit performs eight absolute differences in parallel and accumulates the result in a single clock cycle. Therefore, it needs 32 clock cycles to compute the SAD for 16x16 MBs.

So the SAD expression:

$$SAD(u,v) = \sum_{l=0}^{15}\sum_{c=0}^{15} |C_{(l,c)} - R_{(u+l,c+v)}| \ , \tag{4.2}$$

for a Current ($C$) and a Reference ($R$) MBs is computed as:

$$SAD(u,v) = \sum_{l=0}^{15}\sum_{c=0}^{1} SAD\_8(l,c,u,v) \tag{4.3}$$

$$SAD\_8(l,c,u,v) = \sum_{i=0}^{7} |C_{(l,8*c+i)} - R_{(u+l,v+8*c+i)}| \tag{4.4}$$

where $SAD\_8(l, c, u, v)$ is computed in a single clock cycle.

In order to provide both 8 byte signals to the arithmetic part, the SAD Unit also contains a buffer the current and the reference MBs. Because the current MB is used throughout of the search process it is completely cached on a buffer. The candidate macroblock data must be held only until it is needed by the arithmetic part, thus the candidate MB buffer is only able to hold a single macroblock line. Both buffers are dual-port in order to allow the simultaneous reading and writing: while the SAD is operating on the first 8 pixels of each macroblock line, the system is generating the addresses and fetching the data for the remaining 8 pixels.

### 4.2.4 Operation

Table 4.5 shows a summary of the input and output signals of the motion estimation processor.

Table 4.5: Motion estimator I/O Signals

| **Name** | **bits** | **Dir** | **Description** |
|---|---|---|---|
| clk | 1 | in | clock input |
| rst | 1 | in | global reset, active high |
| mb_x_max | 5 | in | maximum MB x coordinate |
| mb_x_max | 5 | in | maximum MB y coordinate |
| framewidth | 8 | in | framewidth in pixels |
| pattern_in | 32 | in | pattern input bus |
| pattern_addr_in | 7 | in | pattern input address |
| pattern_write | 1 | in | pattern write enable, active high |
| tb_ar0_addr | 20 | out | pixel address for current frame |
| tb_ar0_data | 8 | in | pixel data for current frame |
| tb_ar1_addr | 20 | out | pixel address for reference frame |
| tb_ar1_data | 8 | in | pixel data for reference frame |
| base_mb_x | 5 | in | x coordinate of current MB |
| base_mb_y | 5 | in | y coordinate of current MB |
| MB_base_addr_in | 20 | in | address of the top left pixel of current MB |
| sad_threshold | 16 | in | target maximum value for the SAD value |
| max_steps | 3 | in | maximum number of search steps |
| motest_done | 1 | out | motion estimation complete flag, active high |
| motion_vector_x_out | 5 | out | motion vector output, x coordinate |
| motion_vector_x_out | 5 | out | motion vector output, y coordinate |
| d_sad | 16 | out | SAD obtained for the output motion vectors |

The motion estimator operation is divided into two phases. The first one

is the set up phase, where the frame characteristics are defined and the pattern sets are transfered into the pattern memory. This is accomplished by means of the "framewidth", the "MB_max_coordinate_" pair and the "pattern_" set of signals. The "framewidth" signal is the number of pixels in each line of the frame and is used on the "MB_line_unit" which provides address offsets that produce the vertical displacement within the macroblocks (see subsection 4.2.1). The "MB_max_coordinate_" signals contain the coordinates of the bottom line and leftmost column of macroblocks and are used by the Search Decision Unit in order to detect motion vectors that exceed the frame dimensions. In order to load the pattern sets into the motion estimator, the "pattern_write" signal must be activated after setting valid values on the "pattern_addr_in" and "pattern_in" signals for each of the patterns. This is a synchronous transaction so the signals must be held active on a clock rising edge, considering the set up

The second phase is the motion estimation itself. It is started by setting the "MB_base_coordinate_" and "Base_MB_addr" signals and then activating the "motest_en" signal. This will instruct the processor to perform the motion estimation process on the referenced MB. In order to perform motion estimation for a whole frame this process must be performed once for each MB on the frame, setting the appropriate MB coordinates and MB address for each MB. The "pattern_sel" signal controls the search algorithm by selecting the pattern set to be used for the motion estimation. The "Base_MB_coordinate_" signals are used by the Search Decision Unit (SDU) to detect invalid MVs. The "Base_MB_addr" signal is the memory address of the top left pixel of the current MB. The AGU generates the address offsets that are needed by the search process and requires this signal in order to apply the offsets to a specific MB address and produce valid addresses.

# Chapter 5

# Prototype System for Video Coding and Experimental Results

The prototype system was implemented by integrating the motion estimator described on the previous chapter on the testbench module described on section 3.5: the R0 to R4 registers are used to configure and control the motion estimator, while the frame memories AR0 and AR1 hold the current and reference frame, respectively.

A software video encoder capable of offloading the motion estimation task to the hardware motion estimation processor was executed on the Linux system. This video encoder was based on the Telenor h263 reference code [28] but was greatly simplified: mainly the sequence of frame types was set to IPIP, the sub pixel accuracy motion estimation mode was disabled and no rate control is performed.

The motion estimator control and output signals are mapped into the control registers as shown in Table 5.1. The motion estimator control is performed through the registers R0 through R4. R0 is used as the command entry point and also provides status output. The commands and feedback values are presented on Tab. 5.2.

The GO command starts the motion estimation, R0 changes to BUSY on next clock. The RST command activates the "motest_rst" signal, R0 changes to IDLE on next clock. The PATTERN_WRITE command activates the "pattern_write" signal. This command is used in order to program the pattern sets. The IDLE keyword means that the processor is idle and ready to receive commands. The MOTEST_DONE keyword signals a completed motion estimation process. The BUSY keyword means that a motion estimation is in process, and that the "motest_en" signal is activated.

Table 5.1: Testbench Register configuration for motion estimator control

| **Register** | **Bit(s)** | **Signal name** | **Description** |
|---|---|---|---|
| R0 | | Control and feedback | |
| R1 | | Macroblock address | |
| | 16-23 | base_mb_x | Macroblock x coordinate |
| | 24-31 | base_mb_y | Macroblock x coordinate |
| | 44-63 | MB_base_addr | Macroblock base address |
| R2 | | Limits | |
| | 16-23 | mb_x_max | Max Macroblock x coordinate |
| | 24-31 | mb_y_max | Max Macroblock x coordinate |
| | 56-63 | framewidth | Framewidth |
| R3 | | Pattern In | |
| | 23 | pattern_sel | pattern series select |
| | 25-31 | pattern_addr_in | pattern address |
| | 32-63 | pattern_in | pattern data |
| R4 | | Output | |
| | 19-23 | motion_vector_x_out | Motion Vector Y |
| | 27-31 | motion_vector_y_out | Motion Vector Y |
| | 48-63 | d_sad | SAD |

Table 5.2: Commands and keywords recognized by the module

| **Command** | **Value** | **Description** |
|---|---|---|
| GO | X"0000000000000001" | Starts the motion estimation. R0 changes to BUSY on next clock. |
| RST | X"0000000000000002" | Activates the "motest_rst" signal. R0 changes to IDLE on next clock. |
| PATTERN_WRITE | X"0000000000000008" | Activates the "pattern_write" signal. This command is used in order to program the pattern sets |
| IDLE | X"0000000000000000" | The processor is idle and ready to receive commands. |
| MOTEST_DONE | X"1000000000000001" | Signals a completed motion estimation process. |
| BUSY | X"0000000000000004" | A motion estimation is in process. The "motest_en" signal is activated. |

Before starting the operation of the motion estimator, it must first be configured (see section 4.2.4). The frame limits are written directly onto

R2 and the pattern sets are configured by entering the PATTERN_WRITE keyword into R0 and by writing the different pattern entries into R3.

After the motion estimator is configured, the video encoder starts its operation by loading the frame data from the input files. Since the first frame is always coded in intra mode, no motion estimation is performed on it and the required data is only the raw data for the first frame.

The next frame is encoded in inter mode so the motion estimator requires that the current and reference frames are transfered into the frame memories. Usually the frames that are used in motion estimation and compensation are the current raw and the reconstructed reference. However, in order to keep the video encoder simple, in this case the frames used for motion estimation were the raw current and raw reference (the reconstructed frames were still used for motion compensation). This can lead to motion vectors that are slightly worse, but since the aim of the video encoder is only to demonstrate the motion estimator operation, this effect is not very serious.

In order to transfer the frame it is just necessary to write the frame data into the /dev/ar0 and /dev/ar1 device files. The encoder then instructs the motion estimator to perform motion estimation by using /dev/r0 and /dev/r1 (Tab. 5.1) and reads the motion vectors from /dev/r4 for each of the MBs in the frame. Motion compensation is then performed using these MV values and the remainder of the encoding process is then performed as described earlier (Fig. 2.5).

The typical operation of the motion estimator processor by the software video encoder is depicted on Figure 5.1.

The ML310 prototype board provides the option of using personality module (PM), which consist in external hardware connected through high speed buses to dedicated pins on the Virtex FPGA (PM1 and PM2 on Fig 3.3). Using this feature a system was conceived that uses an external camera module as the source of input frames.

In order to acquire the frames, a new keyword was defined for the control register (GETFRAME) that is used by the video encoder to request a frame from the camera, and a multiplexer that provides access to the frame memories from the camera PM (Fig. 5.2).

Each time the GETFRAME keyword is used, the camera PM will transfer a new frame into one of the motion estimator frame memories and then change the value on the control register to IDLE. In this case, each time the video encoder needs a new frame it will issue the GETFRAME command and then begin a DMA transfer from the frame memories.

The motion estimation co-processor was described using Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) behavioral description and its functionality was thoroughly tested. Results

```
/dev/r2 ⇐ limits {set the framewidth, max mb coordinates}
for all patterns do
   /dev/r3 ⇐ patternseries : patternaddress : pattern
   /dev/r0 ⇐ LOAD_PATTERN
end for
/dev/r3 ⇐ patternseriesselect
loop
   load_frame()
   /dev/ar1 ⇐ frame
   code_frame_intra()
   load_frame()
   /dev/ar0 ⇐ frame
   for all MBs in frame do
      /dev/r1 ⇐ MB_coordinates : MB_address
      /dev/r0 ⇐ DO_MOTEST
      /dev/r4 ⇒ MVs
   end for
   inter_coding()
end loop
```

Figure 5.1: Pseudocode depicting the interaction between the software video encoder and the motion estimation processor.

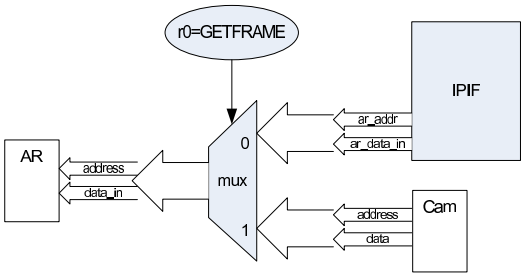


Figure 5.2: Multiplexer that selects the current frame memories controller

obtained with ISE 8.1.03i, with default options, are presented in Table 5.3. This table shows the results for the 132 nm process Virtex-II Pro that is found on the ML310 prototyping board and also for a more recent 90 nm process Virtex 4.

The minimum time required by the motion estimator in order to perform motion estimation on a single frame is given by the Equation 5.1:

Table 5.3: FPGA Occupation and Timing Report

| | **XC2VP30-6FF896** | | **XC4VFX60-11FF672** | |
|---|---|---|---|---|
| **Logic Utilization** | **Used** | **Available** | **Used** | **Available** |
| Total Number Slice Registers | 2517 (9%) | 27392 | 2518 (4%) | 50560 |
| Number of occupied Slices | 2214 (16%) | 13696 | 2212 (3%) | 25280 |
| Number of Block RAMs | 1 (1%) | 136 | 1 (1%) | 232 |
| | **Delay** | $f_{max}$ | **Delay** | $f_{max}$ |
| **Clock** | 15.4 ns | 65 MHz | 3.1 ns | 323 MHz |

$$T = M\frac{2s + 256c + R}{f} \tag{5.1}$$

where:
T - time required to perform motion estimation on a frame.
M - number of macroblocks in a frame;
s - number of steps of the search algorithm;
c - number of candidate macroblocks tested per output motion vector;
R - number of clocks required to read the motion vectors;
f - clock frequency;

This expression clearly shows that the performance of the motion estimator is greatly dependent on the search algorithm being used. As an example, the 3SS algorithm tests 25 candidate blocks in order to find each motion vector. Therefore, assuming R=10, for the 65MHz achieved maximum clock frequency on the Virtex-II Pro, the time required to motion estimate a frame is 9.8 ms which corresponds to framerates of about 102 fps and 25 fps achieved for QCIF and CIF resolutions, respectively. If the clock frequency obtained for the Virtex 4 is used, the CIF framerate will be 127 fps.

However, this expression is only valid for a single frame because the time required to fill the frame memories is not being taken into account. Using a value of 10 for R also assumes that reading the result motion vectors introduces very little overhead. In order to obtain an estimate for the framerate achieved by the Linux based prototype system these factors must be considered. Setting up the DMA transfers for the frame data and reading the result motion vectors are particularly expensive tasks in the implemented system because they involve file operations performed by the application and the driver. This means using Equation 5.2 and using a larger value for R. A reasonable estimate is R=100 and D=600, where D is the number of clock cycles required to set up a DMA transfer.

On the prototype system, the motion estimator is running at a clock

frequency of 50 MHz the PowerPC core is running at a clock frequency of 300 MHz (5 faster than the motion estimator). This means that the values used for D and R correspond to 3000 and 500 clocks on the Linux system, respectively.

$$T = 2D + M\frac{2s + 256c + R}{f} \tag{5.2}$$

where:
T - time required to perform motion estimation on a frame.
D - number of clocks required to set up a DMA transfer;
M - number of macroblocks in a frame;
s - number of steps of the search algorithm;
c - number of candidate macroblocks tested per output motion vector;
R - number of clocks required to read the motion vectors;
f - clock frequency;

Applying this equation to an encoding process of a CIF sequence using the 3SS algorithm and estimating D=600 and R=100, for a clock rate of 50 MHz the time required to motion estimate a frame is 14 ms which corresponds to a framerate of 17 fps.

The previous expressions consider only the motion estimation and the tasks that are inseparable from it. Although motion estimation is the single most demanding task in a video encoder, there are several other tasks that are also time consuming, and consequently affect the framerate, such as I/O, DCT transform and entropy coding.

The software video encoder used on the prototype system is not optimized for running on the PPC405 processor available on the prototype board, which, does not include a floating point unit. This means that the computation of the DCT is very inefficient because the floating point operations are translated into a large number of integer and logic operations.

These factors severely hinder the performance of the complete test encoder. The frame rate achieved while encoding a test Quarter Common Intermediate Format – $176 \times 144$ (QCIF) sequence is only around 0.6 fps.

# Chapter 6

# Conclusions and Future Work

In recent years video features have found a way into an increasing number of applications, introducing new functionalities and improving on existing ones. On the one hand, mobile devices such as cell phones and PDAs integrate increasingly complex video functionalities while requiring hardware architectures that are energy efficient. On the other hand, traditional video applications are becoming more demanding (e.g. HDTV 1080) and require higher performance.

Further developments are thus mainly hindered by problems related to computation performance and energy efficiency (power consumption). Dedicated specialized hardware solutions are able to address both issues as they can be optimized in order to provide very high performance with lower power consumption.

Video coding is a fundamental part of video applications and motion estimation is the most demanding part of video coding that can account for more than 60% of computation time and power consumption. This makes motion estimation a prime candidate for research in order to produce efficient and versatile hardware architectures.

This work proposes an innovative mechanism able to reconfigure in runtime the search algorithm in hardware motion estimation processors. This mechanism allows to develop very efficient motion estimators, with the performance of the hardwired solutions and the flexibility of programmable approaches. This work has been published in an international conference [27].

The proposed reconfigurable motion estimator was synthesized for Xilinx FPGAs Virtex-II and Virtex-4 families. Experimental results show these reconfigurable processors achieve real-time motion estimation on intermediate and high resolution image sequences. Because no FPGA-specific macros were used, the designs can easily be synthesized into static silicon technology yielding yet greater performance.

The currently available hardware development platforms provide a rapid and cost effective way of developing and prototyping new designs. Programmable devices have matured to the point where it is easy to implement a complete system on a single device: a hybrid H.263 video encoder was implemented in the Xilinx Virtex-II Pro ML310 Embedded Development Platform. In this platform, most of the processing is run on the PowerPC while the motion estimation is offloaded to a co-processor implemented in the reconfigurable part of the FPGA. Although this prototype is just a proof of concept, it shows the possibility of integrating the proposed and implemented motion estimator as a reconfigurable co-processor in a complete video coding system.

## 6.1 Future Work

The implemented system is a proof of concept prototype, so there is room for improvements, especially on the the frame memory fetch process, where the effective buswidth is only 8 bits. The frame memories have a data width of 64 bits, and therefore can output 8 pixels of data per clock cycle. The inputs for SAD block are also 64 bits wide so under optimum conditions it can compute the SAD value in 32 clock cycles (1/8 of the clocks currently used). The AGU block produces the address of each pixel to be fetched from the frame memories at the rate of one by clock cycle. This means 256 clock cycles are needed to generate the addresses for each Macroblock. However, this rate can be easily altered by changing the counters on the AGU_Column_Unit in order to perform a +=n operation instead of an increment operation. Setting n to 8, the AGU will output addresses compatible with the 64-bit memory datawidth. However, because the candidate macroblocks are not 8 byte aligned, this approach depends on the SAD block caches being able to perform an extra fetch cycle in order to fetch all the required data and align it in the required way. The usage of larger and more efficient caches (e.g. as described on Appendix C) can also improve the design, by reducing the memory data bus usage.

# Appendix A

# Search Algorithm Tables

Table A.1 shows the standard Three Step Search algorithm, which tests 25 candidate MBs, Table A.2 shows the standard Four Step Search algorithm, which tests 33 candidate MBs and Table A.4 shows the standard Diamond Search algorithm, which tests 25 candidate MBs.

Table A.1: Three Step Search

| Pattern address | **Scan end** | **Step end** | **Next pattern address** | $\Delta x$ | $\Delta y$ |
|---|---|---|---|---|---|
| 0 | | | 9 | 0 | 0 |
| 1 | | | 9 | -4 | -4 |
| 2 | | | 9 | 0 | -4 |
| 3 | | | 9 | 4 | -4 |
| 4 | | | 9 | -4 | 0 |
| 5 | | | 9 | 4 | 0 |
| 6 | | | 9 | -4 | 4 |
| 7 | | | 9 | 0 | 4 |
| 8 | | 1 | 9 | 4 | 4 |
| 9 | | | 18 | 0 | 0 |
| 10 | | | 18 | -2 | -2 |
| 11 | | | 18 | 0 | -2 |
| 12 | | | 18 | 2 | -2 |
| 13 | | | 18 | -2 | 0 |
| 14 | | | 18 | 2 | 0 |
| 15 | | | 18 | -2 | 2 |
| 16 | | | 18 | 0 | 2 |
| 17 | | 1 | 18 | 2 | 2 |
| 18 | | | | -1 | -1 |
| 19 | | | | 0 | -1 |
| 20 | | | | 1 | -1 |
| 21 | | | | -1 | 0 |
| 22 | | | | 1 | 0 |
| 23 | | | | -1 | 1 |
| 24 | | | | 0 | 1 |
| 25 | 1 | 1 | | 1 | 1 |

Table A.2: Four Step Search

| Pattern address | **Scan end** | **Step end** | **Next pattern address** | $\Delta x$ | $\Delta y$ |
|---|---|---|---|---|---|
| 0 | | | 73 | 0 | 0 |
| 1 | | | 9 | -2 | -2 |
| 2 | | | 15 | 0 | -2 |
| 3 | | | 19 | 2 | -2 |
| 4 | | | 25 | -2 | 0 |
| 5 | | | 29 | 2 | 0 |
| 6 | | | 33 | -2 | 2 |
| 7 | | | 39 | 0 | 2 |
| 8 | | 1 | 43 | 2 | 2 |
| 9 | | | 89 | 0 | 0 |
| 10 | | | 49 | -2 | -2 |
| 11 | | | 55 | 0 | -2 |
| 12 | | | 59 | 2 | -2 |
| 13 | | | 65 | -2 | 0 |
| 14 | | 1 | 73 | -2 | 2 |
| 15 | | | 89 | 0 | 0 |
| 16 | | | 49 | -2 | -2 |
| 17 | | | 55 | 0 | -2 |
| 18 | | 1 | 59 | 2 | -2 |
| 19 | | | 89 | 0 | 0 |
| 20 | | | 49 | -2 | -2 |
| 21 | | | 55 | 0 | -2 |
| 22 | | | 59 | 2 | -2 |
| 23 | | | 69 | 2 | 0 |
| 24 | | 1 | 83 | 2 | 2 |
| 25 | | | 89 | 0 | 0 |
| 26 | | | 49 | -2 | -2 |
| 27 | | | 65 | -2 | 0 |
| 28 | | 1 | 73 | -2 | 2 |
| 29 | | | 89 | 0 | 0 |
| 30 | | | 59 | 2 | -2 |
| 31 | | | 69 | 2 | 0 |
| 32 | | 1 | 83 | 2 | 2 |
| 33 | | | 89 | 0 | 0 |
| 34 | | | 49 | -2 | -2 |
| 35 | | | 65 | -2 | 0 |
| 36 | | | 73 | -2 | 2 |
| 37 | | | 79 | 0 | 2 |
| 38 | | 1 | 83 | 2 | 2 |
| 39 | | | 89 | 0 | 0 |
| 40 | | | 73 | -2 | 2 |
| 41 | | | 79 | 0 | 2 |
| 42 | | 1 | 83 | 2 | 2 |
| 43 | | | 89 | 0 | 0 |
| 44 | | | 59 | 2 | -2 |
| 45 | | | 69 | 2 | 0 |
| 46 | | | 73 | -2 | 2 |
| 47 | | | 79 | 0 | 2 |
| 48 | | 1 | 83 | 2 | 2 |

Table A.3: Four Step Search (continuation)

| Pattern address | **Scan end** | **Step end** | **Next pattern address** | $\Delta x$ | $\Delta y$ |
|---|---|---|---|---|---|
| 49 | | | 89 | 0 | 0 |
| 50 | | | 89 | -2 | -2 |
| 51 | | | 89 | 0 | -2 |
| 52 | | | 89 | 2 | -2 |
| 53 | | | 89 | -2 | 0 |
| 54 | | 1 | 89 | -2 | 2 |
| 55 | | | 89 | 0 | 0 |
| 56 | | | 89 | -2 | -2 |
| 57 | | | 89 | 0 | -2 |
| 58 | | 1 | 89 | 2 | -2 |
| 59 | | | 89 | 0 | 0 |
| 60 | | | 89 | -2 | -2 |
| 61 | | | 89 | 0 | -2 |
| 62 | | | 89 | 2 | -2 |
| 63 | | | 89 | 2 | 0 |
| 64 | | 1 | 89 | 2 | 2 |
| 65 | | | 89 | 0 | 0 |
| 66 | | | 89 | -2 | -2 |
| 67 | | | 89 | -2 | 0 |
| 68 | | 1 | 89 | -2 | 2 |
| 69 | | | 89 | 0 | 0 |
| 70 | | | 89 | 2 | -2 |
| 71 | | | 89 | 2 | 0 |
| 72 | | 1 | 89 | 2 | 2 |
| 73 | | | 89 | 0 | 0 |
| 74 | | | 89 | -2 | -2 |
| 75 | | | 89 | -2 | 0 |
| 76 | | | 89 | -2 | 2 |
| 77 | | | 89 | 0 | 2 |
| 78 | | 1 | 89 | 2 | 2 |
| 79 | | | 89 | 0 | 0 |
| 80 | | | 89 | -2 | 2 |
| 81 | | | 89 | 0 | 2 |
| 82 | | 1 | 89 | 2 | 2 |
| 83 | | | 89 | 0 | 0 |
| 84 | | | 89 | 2 | -2 |
| 85 | | | 89 | 2 | 0 |
| 86 | | | 89 | -2 | 2 |
| 87 | | | 89 | 0 | 2 |
| 88 | | 1 | 89 | 2 | 2 |
| 89 | | | | -1 | -1 |
| 90 | | | | 0 | -1 |
| 91 | | | | 1 | -1 |
| 92 | | | | -1 | 0 |
| 93 | | | | 1 | 0 |
| 94 | | | | -1 | 1 |
| 95 | | | | 0 | 1 |
| 96 | 1 | 1 | | 1 | 1 |

Table A.4: Diamond Search

| Pattern address | **Scan end** | **Step end** | **Next pattern address** | $\Delta x$ | $\Delta y$ |
|---|---|---|---|---|---|
| 0 | | | 49 | 0 | 0 |
| 1 | | | 9 | 0 | -2 |
| 2 | | | 15 | 1 | -1 |
| 3 | | | 19 | 2 | 0 |
| 4 | | | 25 | 1 | 1 |
| 5 | | | 29 | 0 | 2 |
| 6 | | | 35 | -1 | 1 |
| 7 | | | 39 | -2 | 0 |
| 8 | | 1 | 9 | -1 | -1 |
| 9 | | | 49 | 0 | 0 |
| 10 | | | 39 | -2 | 0 |
| 11 | | | 45 | -1 | -1 |
| 12 | | | 9 | 0 | 2 |
| 13 | | | 15 | 1 | 1 |
| 14 | | 1 | 19 | 2 | 0 |
| 15 | | | 49 | 0 | 0 |
| 16 | | | 9 | 0 | -2 |
| 17 | | | 15 | 1 | -1 |
| 18 | | 1 | 19 | 2 | 0 |
| 19 | | | 49 | 0 | 0 |
| 20 | | | 9 | 0 | -2 |
| 21 | | | 15 | 1 | -1 |
| 22 | | | 19 | 2 | 0 |
| 23 | | | 25 | 1 | 1 |
| 24 | | 1 | 29 | 0 | 2 |
| 25 | | | 49 | 0 | 0 |
| 26 | | | 19 | 2 | 0 |
| 27 | | | 25 | 1 | 1 |
| 28 | | 1 | 29 | 0 | 2 |
| 29 | | | 49 | 0 | 0 |
| 30 | | | 19 | 2 | 0 |
| 31 | | | 25 | 1 | 1 |
| 32 | | | 29 | 0 | 2 |
| 33 | | | 35 | -1 | 1 |
| 34 | | 1 | 39 | -2 | 0 |
| 35 | | | 49 | 0 | 0 |
| 36 | | | 19 | 0 | 2 |
| 37 | | | 25 | -1 | 1 |
| 38 | | 1 | 29 | -2 | 0 |
| 39 | | | 49 | 0 | 0 |
| 40 | | | 19 | 0 | 2 |
| 41 | | | 25 | -1 | 1 |
| 42 | | | 29 | -2 | 0 |
| 43 | | | 35 | -1 | -1 |
| 44 | | 1 | 39 | 0 | -2 |
| 45 | | | 49 | 0 | 0 |
| 46 | | | 39 | -2 | 0 |
| 47 | | | 45 | -1 | -1 |
| 48 | | 1 | 9 | 0 | -2 |
| 49 | | | | 0 | -1 |
| 50 | | | | 1 | 0 |
| 51 | | | | 0 | 1 |
| 52 | 1 | | | -1 | 0 |

# Appendix B

# Development platform configuration

## B.1 Hardware Configuration

The base hardware configuration was performed using the Xilinx supplied PCI reference design. This design configures the FPGA in order to implement several Xilinx IP cores that are used by the Linux system.

The design can be found on the ML310 documentation CD under:

`tutorials/Linux_Lab/ml310_pci_design.zip`

Because the reference design is intended for use with Xilinx Platform Studio (XPS) 6.1, it is necessary to apply the project update if XPS 6.3 or superior is to be used. The project update can be found at

`http://www.xilinx.com/products/boards/ml310/current/`
`/reference_designs/pci/ml310_pci_design_edk63_update.zip`

The updated project is created by uncompressing the base archive to an empty `<xps_prj_dir>` folder and then uncompressing the update onto the same folder. The project can then be opened with XPS.

In order to generate the bitstream that describes the hardware configuration on the FPGA it is only required to use the option "Generate Bitstream" under the "Project" menu. The "Update Bitstream" menu item adds the processors' BRAM initialization data to the hardware configuration bitstream. In this project this is a simple loop instruction, which prevents the processors from executing instructions from uninitialized memory.

The newly created binary file is located at `<xps_prj_dir>\implementation\download.bit`. Bit files are Xilinx FPGA configuration files generated by the Xilinx FPGA

design software. They are proprietary format binary files containing configuration information.

Section B.3 describes the process of downloading this hardware description file into the FPGA.

## B.2 Operating System Configuration

The ML310 is supplied with a working Linux system by means of an ACE file and a partition on the flashcard. The ACE file configures the FPGA hardware and programs the processor's BRAMs with the kernel image.

In order to allow for custom hardware definition, the default ACE file was not used in this project. Instead, the hardware definition file was created as described on the previous section and the software file used was a customized Linux kernel. The partition on the flash card was also replaced by a hard drive so that the Linux system more disk space available [23].

### B.2.1 Cross Compiler

Because the software development was not performed natively on the ML310, a cross compiler was required. This tool is able to generate binary code for a different platform thatn the one it is running on. The Montavista Linux Preview Kit for Professional Edition 3.1 was used as the cross compiler system on this work. This free version has limited functionality and integration and is only available for Linux platforms, so the process of configuring the test platform was slightly different from the one documented on [23].

In order to download the preview kit for the ml310 it is necessary to fill out the registration form at (http://www.mvista.com/previewkit/index.html), choosing "Xilinx Virtex-II Pro ML300" as the platform. This will produce an e-mail sent to the the address entered into the "E-mail address" field which contains an FTP link and a password to be used during the installation process. After downloading the file from the FTP link, the preview kit can be installed by entering the commands below on a Linux shell.

```
    $ mkdir <mount_dir>
    $ su
    # mount previewkit-mvl310_xilinx-ml300-encrypt.img \
-t iso9660 -o loop <mount_dir>
    # cd <mount_dir>
    # ./install_previewkit
```

This creates the <mount_dir> directory, changes to root context mounts the previewkit-mvl310_xilinx-ml300-encrypt.img image file on <mount_dir> as an iso9660 filesystem and starts the installation script.

The install script prompts for the password that was included in the email from Montavista from the registration process. Once it is entered a path to be the top installation directory is requested (<previewkit_dir>, default is /opt). Next, the script prompts for choices of architectures to install binaries for. The selection screen should look something like:

```
      Architecture   Linux Support Package
      ------------   ---------------------
1.         ppc_405   xilinx-ml300-previewkit
                     Xilinx® ML300
```

The actual installation starts after selecting the '1' option. The install script completes after a couple of minutes. The mount point for the image file can then be released and root privileges dropped:

```
# cd ..
# umount <mount_dir>
# exit
```

In order to make use of the PowerPC crosscompiler toolchain, it is necessary to add the binary files location to the PATH. From a bash shell:

```
$ PATH=$PATH:<previewkit_dir>/montavista/preview/ppc/405/bin
```

### B.2.2 Custom Kernel Configuration

The Montavista Linux Preview Kit for Professional Edition 3.1 provides a kernel source tree that has been patched for PowerPC processors and is appropriate for the ML310. In order to create the binary boot file for the hardware platform, this source tree should first be copied to a work directory (<kernel_src_dir>).

```
$ cp -R <previewkit_dir>/montavista/previewkit/lsp\
/xilinx-ml300-previewkit-ppc_405/linux-2.4.20_mvl31 \
<kernel_src_dir>
$
```

Next, Board Support Packages should be merged into the kernel source directory. The Board Support Packages (BSPs) are definitions and drivers generated by XPS [29].

In order to generate the BSPs the design described on B.1 must be opened in XPS and the menu option "Generate Libraries and BSPs" on thew "Tools" menu must be used. The location where the BSP files are placed can be chosen by setting the "TARGET_DIR" parameter on the "Library/OS Parameters" pane of the "Software Platform Settings" dialog which is accessible on the "Project" menu. This parameter should be set to the base of the kernel source tree (`<kernel_source_dir>`) so there is no need to manually copy the BSPs into the kernel directory.

Once the BSPs are built the kernel sources are final and the kernel can be configured. The ML310 documentation provides a kernel configuration file that is appropriate for the Montavista patched kernel (.config on the `tutorials/Linux_Lab/ml310_pci_linux_kernel_config.zip` archive). This file must be uncompressed to the base of the kernel source tree (`<kernel_src_dir>`) and then the command below must be issued to set the defaults for the kernel configuration.

```
$ cd <kernel_src_dir>
$ make oldconfig
```

Because the system is supposed to use the hard drive as the root device, the boot parameters must be changed. To this end the following command must be issued.

```
$ make menuconfig
$
```

This starts the kernel configuration program. The "Initial kernel command string", under "General setup" should be set to "console=ttyS0,9600 ip=off root=/dev/hda1 rw". After exiting and saving the changes to the configuration (select `<Exit>` and answer `<Yes>` when prompted whether to save changes), the kernel can be compiled by issuing the commands below.

```
$ make dep
$ make bzImage
```

Once the compile finishes, the kernel file zImage.elf can be found in `<kernel_src_dir>/arch/ppc/boot/`

## B.3 Platform Initialization

After both the hardware and the software files have been compiled and the Linux filesystem has been transfered into the hard drive the Linux system

can be started. To this end it is necessary to program the FPGA with the hardware file and then use the software file to initialize the memories with data that will be executed by the processor.

The FPGA hardware configuration is performed by the iMPACT tool. This application is able to download the hardware configuration file into the FPGA using JTAG Boundary-Scan through the parallel cable supplied with the ML310.

The command `impact -batch _program` runs iMPACT in batch mode, reading commands from the`_program` shown file below:

```
setMode -bs
setCable -port lpt1
addDevice -position 1 -file ‘‘download.bit’’
program -p 1
quit
```

The commands first select Boundary-Scan and the lpt1 parallel port for communication, then assign the “download.bit” file to the first device in the chain and finally download the hardware file to the device.

After the process completes, the hardware is ready to use and the the software file can be transferred to the FPGA. The Xilinx Microprocessor Debugger (XMD) was used to download the software application into the FPGA.

The command used to run XMD is `xmd -tcl _download.tcl` which runs XMD in batch mode, reading commands from the `_download.tcl` file shown below:

```
connect ppc hw
stop
rst
dow zImage.elf
con
```

These commands cause XMD to connect to the PowerPC on the FPGA, halt it, reset the program counter, download the zImage.elf file to the its BRAMS and then start it. This will boot up the Linux system on the ML310. The basic operating system setup is described on [23].

# Appendix C

# Cache

A given region in the reference frame will be used during the motion estimation of several macroblocks. Thus, the use of a caching mechanism will reduce the required memory bandwidth.

The search algorithms use a rectangular area (of the reference frame) centered on the current macroblock. Since the frames are kept in memory in raster format, the memory access pattern has strides of size equal to the framewidth. This has an ill impact on the cache's performance because of aliasing effects.

In order to evaluate these effects a couple of simple programs were developed.

The first one was a program that simulates a motion estimation pass over a line of macroblocks and prints out the accesses memory addresses. Different search algorithms were implemented, but a worst case scenario in terms of cache performance was always assumed: the search algorithm is steered towards the upper left corner of the search area, in order to reduce data reusage.

To get accurate and reliable measurements of the cache's performance a cache simulator was used (http://www.cs.wisc.edu/ markhill/DineroIV/).

To get more real results, the telenor h263 encoder was slightly altered so it outputs the memory addresses accessed during the motion estimation phase. The results obtained using these sets of addresses were consistent with the ones obtained with the synthetic addresses.

The access pattern is based on macroblocks. This means that, for a 16x16 MB there will be 16 consecutive addresses interleaved by framewidth

In order to minimize the effect of the stride, the

If the cache is configured as a plain (one way) cache, the least significant bits of a main memory address is used as the cache address and the remaining bits are used as the tag.

The tag is stored in the cache along with the data for each address. Each time a main memory address is requested, the corresponding cache address is accessed, and the tag is compared to the corresponding bits on the requested address.

This configuration is not adequate for situations where large strides are present. If, for example, a 1k cache is used, each macroblock line will overwrite the previous one, because the cache addresses will be the same. This effect is called aliasing.

One way of solving aliasing problems is to use associative caches. If an associative cache is used it will be possible to store more than one value for the same cache address. A number of separate memory banks share the same addresses and for each read a multiplexer selects the proper output bus (if any) based on the tag. For the write sequence it's necessary to choose which memory bank will be replaced. There are a number of different replace strategies more or less sophisticated or complex, and with different hardware requirements. Due to the rather large number of different and consecutive strided accesses during motion estimation, the associative order must be the same as the number of lines of the search are, or at the very least the same as the number of lines on a macroblock. This requires a large logic overhead.

In the particular situation of motion estimation a better way is to identify the address bits that are relevant for the search area and base the cache addresses around those address lines.

Setting the framewidth to a power of 2, 512 for example, will make evident the behaviour.

assuming the base address of the frame is 0x00000000, the top right macroblock's addresses will be

0x00000000 to 0x0000000f (the first line) 0x00000100 to 0x0000020f (the second line) 0x00000200 to 0x0000040f (the third line) ... 0x00000f00 to 0x00001e0f (the sixteenth line)

The address can be expressed as b .... .... .... .... ...l lllw wwww cccc

where c: Macroblock column l: Macroblock line w: frame width

The four rightmost bits (c) determine the column of the macroblock. The concatenation of these bits to the five following (w) produces a 9 bit quantity that determines the column on the complete frame. The following four bits (l) determine the macroblock line.

The search area is usually about three times taller and three times wider than a single macroblock, which yields b .... .... .... .... .yyl lll. ..xx cccc where: x: search width y: search height

Although this is an optimum case example where the macroblock is word aligned and the framewidth is a power of 2, the results are valid for the general case.

If the framewidth is not a power of 2 the boundary between the framewidth (w) bits and the macroblock line bits (l) will be less defined.

If the macroblock is not word aligned an iteration through all the columns of the macroblock will change more than just the four least significant bits of the address(c), but this extra change will only happen once, and there will still be a point to point relation between the four lsb and the macroblocks columns.

Since the aim here is to reduce the aliasing effects, and not to completely isolate each variable present, the effects of the non-optimum macroblock placement can thus be overlooked.

Taking this into consideration, the cache was built using the address lines that are not constant over the search area, macroblock line (l) macroblock column (c) search area height (y) and search area width (x).

using the Macroblock column, mb line, search area x and search area y address bits as the cache address, and the remainder bits as tag.

This was done by defining a mangling binary mask. This mask is used to change the bit significance of the address passed to the cache. The address bits that have a logical '1' in the mask are passed to the LSB. The address bits that have a '0' in the mask are passed to the MSB. (e.g. a mangling mask of 0x00f0f changes the address 0x01234 into 0x01324)

The optimum mask depends on the value of framewidth because, as seen above, the useful address bits' position changes with this value.

# Bibliography


[1] I. Richardson, H.264 and MPEG-4 Video Compression, Video Coding for Next-generation Multimedia. John Wiley & Sons, 2003.

[2] V. Bhaskaran and K. Konstantinides, Image and Video Compression Standards: Algorithms and Architectures, 2nd ed. Kluwer Academic Publishers, June 1997.

[3] R. Li, B. Zeng, and M. L. Liou, "A new three-step search algorithm for block motion estimation," IEEE Transactions on Circuits and Systems for Video Technology, vol. 4, no. 4, pp. 438–442, Aug. 1994.

[4] S. Zhu and K.-K. Ma, "A new diamond search algorithm for fast block-matching motion estimation," IEEE Transactions on Image Processing, vol. 9, no. 2, pp. 287–290, Feb. 2000.

[5] A. Tourapis, O. Au, and M. Liou, "Predictive motion vector field adaptive search technique (pmvfast) - enhancing block based motion estimation," in Proceedings of SPIE - Visual Communications and Image Processing (VCIP), San Jose, CA, Jan. 2001, pp. 883–892.

[6] H. Yeo and Y. Hu, "A modular high-throughput architecture for logarithmic search block-matching motion estimation," IEEE Transactions on Circuits ans Systems for Video technology, vol. 8, no. 3, pp. 299–315, June 1998.

[7] T. Dias, S. Momcilovic, N. Roma, and L. Sousa, "Adaptive motion estimation processor for autonomous video devices," EURASIP Journal on Embedded Systems - Special Issue on Embedded Systems for Portable and Mobile Video Platforms, no. 57234, pp. 1–10, May 2007.

[8] H. Peters, R. Sethuraman, A. Beric, P. Meuwissen, S. Balakrishnan, C. Pinto, W. kruijtzer, F. Ernst, G. Alkadi, J. van Meerbergen, and G. Haan, "Application specific instruction-set processor template for

motion estimation in video applications," IEEE Transactions on Circuits ans Systems for Video technology, vol. 15, no. 4, pp. 508–527, Apr. 2005.

[9] Y. Ooi, "Motion estimation system design," in Digital Signal Processing for Multimedia Systems, K. K. Parhi and T. Nishitani, Eds. Marcel Dekker, Inc, 1999, chapter 12, pp. 299–327.

[10] E. Dubois and J. Konrad, "Estimation of 2-D motion fields from image sequences with application to motion-compensated processing," in Motion Analysis and Image Sequence Processing, M. Sezan, , and R. Lagendijk, Eds. Kluwer Academic Pulishers, 1993, vol. 218, pp. 53–87.

[11] C. Stiller and J. Konrad, "Estimating motion in image sequences: A tutorial on modeling and computation of 2-D motion," IEEE Signal Processing Magazine, vol. 16, pp. 70–91, July 1999.

[12] R. Srinivasan and K. R. Rao, "Predictive coding based on efficient motion estimation," IEEE Transactions on Circuits and Systems on Video Technology, vol. 33, no. 8, pp. 888–896, Aug. 1985.

[13] M. Ghanbari, "The cross-search algorithm for motion estimation," IEEE Transactions on Communications, vol. 38, no. 7, pp. 950–953, July 1990.

[14] T. Zahariadis and D. Kalivas, "A spiral search algorithm for fast estimation of block motion vectors," in Signal Processing VIII - Theories and Applications, 1996, pp. 1079–1082.

[15] L.-M. Po and W.-C. Ma, "A novel four-step search algorithm for fast blcok motion estimation," IEEE Transactions on Circuit and Systems for Video Technology, vol. 6, no. 3, pp. 313–317, June 1996.

[16] C. Zhu, X. Lin, L.-P. Chau, K.-P. Lim, H.-A. Ang, and C.-Y. Ong, "A novel hexagon-based search algorithm for fast block motion estimation," in Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing – ICASSP-2001, Salt Lake City - USA, May 2001.

[17] P. L.-M. C. Chun-Ho, "A novel cross-diamond search algorithm for fast block motion estimation," IEEE Transactions on Circuits and Systems for Video Technology, vol. 12, no. 12, pp. 1168–1177, Dec. 2002.

[18] T. Koga, K. Iinuma, A. Hirano, Y. Iijima, and T. Ishiguro, "Motion compensated interframe coding for video conferencing," in National Telecommunications Conference, New Orleans - LA, Nov. 1981, pp. 5.3.1–5.3.3.

[19] J. R. Jain and A. K. Jain, "Displacement measurement and its application in interframe image coding," IEEE Transactions on Communications, vol. 29, no. 12, pp. 1799–1808, Dec. 1981.

[20] B. Liu and A. Zaccarin, "New fast algorithms for the estimation of block matching vectors," IEEE Transactions on Circuits and Systems for Video Technology, vol. 3, no. 2, pp. 148–157, Apr. 1993.

[21] E. Ogura, Y. Ikenaga, Y. Iida, Y. Hosoya, M. Takashima, and K. Yamash, "A cost effective motion estimation processor LSI using a simple and efficient algorithm," in Proceedings of International Conference on Consumer Electronics - ICCE, 1995, pp. 248–249.

[22] S. Lee, J.-M. Kim, , and S.-I. Chae, "New motion estimation algorithm using adaptively-quantized low bit resolution image and its VLSI architecture for MPEG2 video coding," IEEE Transactions on Circuits and Systems for Video Technology, vol. 8, no. 6, pp. 734–744, Oct. 1998.

[23] ML310 Linux Tutorial, Xilinx, tutorials/Linux_Lab/ml310_pci_linux_lab.pdf on the ml310 documentation.

[24] 64-Bit Processor Local Bus Architecture Specifications, 1st ed., IBM, May 2001.

[25] PLB IPIF Product Specification, Xilinx, May 2004.

[26] D. S. Miller, R. Henderson, and J. Jelinek, "Dynamic dma mapping," Linux kernel source, (linux/Doxumentation/DMA-mapping.txt).

[27] M. Ribeiro and L. Sousa, "A run-time reconfigurable processor for video motion estimation," in 17th International Conference on Field Programmable Logic and Applications (FPL). IEEE, May 2007.

[28] Telenor, TMN (Test Model Near Term) - (H.263) encoder/decoder (source code), v2.0 ed., Telenor Research and Development, June 1996.

[29] Automatic Generation of Linux Support Packages Through Xilinx Platform Sturio (XPS), Xilinx, June 2005, http://www.xilinx.com/ise/embedded/edk82i_docs/linux_mvl31_ug.pdf.