1-1-2008

Implementation of BMA based motion estimation hardware accelerator in HDL

Nachiket Jugade
University of Nevada, Las Vegas

IMPLEMENTATION OF BMA BASED MOTION ESTIMATION HARDWARE ACCELERATOR IN HDL

by

Nachiket Jugade

Bachelor of Science in Electrical Engineering
University of Pune, India
2006

A thesis submitted in partial fulfillment
of the requirements for the

Masters of Science Degree in Electrical Engineering
Department of Electrical and Computer Engineering
Howard R. Hughes College of Engineering

Graduate College
University of Nevada Las Vegas
August 2008

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

Thesis Approval
The Graduate College
University of Nevada, Las Vegas

JULY 15, 2008

The Thesis prepared by

NACHIKET JUGADE

Entitled

IMPLEMENTATION OF BMA BASED MOTION
ESTIMATION HARDWARE ACCELERATOR USING HDL

is approved in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

[illegible]
Examination Committee Chair

[illegible]
Dean of the Graduate College

[illegible] 07/15/08
Examination Committee Member

[illegible]
Examination Committee Member

[illegible]
Graduate College Faculty Representative

1017-53

ABSTRACT

Implementation of BMA Based Motion Estimation Hardware Accelerator In HDL

by

Nachiket Jugade

Dr. Henry Selvaraj, Examination Committee Chair
Professor of Electrical Engineering
University of Nevada, Las Vegas

Motion Estimation in MPEG (Motion Pictures Experts Group) video is a temporal prediction technique. The basic principle of motion estimation is that in most cases, consecutive video frames will be similar except for changes induced by objects moving within the frames. Motion Estimation performs a comprehensive 2-dimensional spatial search for each luminance macroblock (16x16 pixel block). MPEG does not define how this search should be performed. This is a detail that the system designer can choose to implement in one of many possible ways. It is well known that a full, exhaustive search over a wide 2-dimensional area yields the best matching results in most cases, but this performance comes at an extreme computational cost to the encoder. Some lower cost encoders might choose to limit the pixel search range, or use other techniques usually at some cost to the video quality which gives rise to a trade-off.

Such algorithms used in image processing are generally computationally expensive. FPGAs are capable of running graphics algorithms at the speed comparable to dedicated

graphics chips. At the same time they are configurable through high-level programming languages, e.g. Verilog, VHDL. The work presented entirely focuses upon a Hardware Accelerator capable of performing Motion Estimation, based upon Block Matching Algorithm. The SAD based Full Search Motion Estimation coded using Verilog HDL relies upon a 32x32 pixel search area to find the best match for single 16x16 macroblock.
Keywords: Motion Estimation, MPEG, macroblock, FPGA, SAD, Verilog, VHDL

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I take this opportunity to acknowledge the people who have helped me during the course of my thesis work. My special thanks to my advisor Dr. Henry Selvaraj who has helped me and guided me in the right direction throughout the course of this thesis. His invaluable support and encouragement has made this possible. I would also like to thank my parents and my sister who are my greatest support system and encouraged me to work hard during the period of 2 years. Without them, it would have been not only difficult, but also impossible. I would also like to thank my committee members, Dr. Venki, Dr. Regentova and Dr Gewali for their timely support and encouragement. Last but not the least, I would like to thank my B348 lab mates, Ram, Arun, Vikram and Ashwini with whom I enjoyed the lighter moments during my time in this lab. I cannot imagine these 2 years without such wonderful fellow lab mates. Thank you everyone!

CHAPTER 1

INTRODUCTION

1.1 Introduction to motion estimation theory

Image compression, whether for still pictures or motion pictures (e.g., video), plays an important role in Internet and multimedia applications, digital appliances such as HDTV, and handheld devices such as digital cameras and mobile phones. Compression allows one to represent images and video with a much smaller amount of data and negligible quality loss. The reduction in data decreases storage requirements (important for embedded devices) and provides higher effective transmission rates (important for Internet enabled devices). Unfortunately, implementing a compression scheme can be especially difficult. For performance reasons, implementations are typically not portable as they are tuned to specific architectures. And while image and video compression is needed on embedded systems, desktop PCs, and high end servers; implementing all probable architectures separately is not cost effective. Furthermore, compression standards are also continuously evolving, and thus compression programs must be easy to modify and update.

In the last few years there has been a growing trend to design very complex and efficient processing systems by integrating already developed and dedicated cores which implement, in a particularly efficient way, certain specific and critical parts of the main system. Such design approach can either be conducted in order to obtain very complex and autonomous processing architectures, or to implement specific and dedicated processing structures that will be integrated with other larger scale processing modules, in the form of co-processors, to alleviate the computational burden. As a consequence, a significant amount of quite different processing modules have been proposed and made available, providing an easy integration with the target processing systems and a substantial reduction of the design effort. To attain such objective, these processing cores have to follow strict design methodologies, in order to provide an easy and efficient implementation in a broad range of target implementation technologies (e.g.: FPGA, ASIC, etc.) [2].

Recently, we are witnessing a new trend in embedded processor design that is again quickly reshaping the embedded processor design. Instead of implementing the time critical tasks in ASICs, these tasks are to be implemented in field-programmable gate arrays (FPGA) structures or comparative technologies [6, 7]. Since FPGAs have the advantages such as

- Increased flexibility: The functionality of the embedded processor can be quickly changed without requiring another roll-out of the embedded processor itself and design faults can be quickly rectified. It also allows for quick adaptation of new (possibly unforeseen) developments.
- Sufficient performance: The performance of FPGAs has increased tremendously and is quickly approaching that of ASICs [2]. This seems to be mainly due to the faster adaptation of new technological advancements by FPGAs than by ASICs.
- Faster design times: Faster design times are achieved by re-using intellectual property (IP) cores or by slightly modifying them. More importantly, high-level

design languages (such as Verilog, VHDL etc) can be used in the design process and thereby speeding it up significantly.

Field-programmable gate array is a semiconductor device containing programmable logic components called "logic blocks", and programmable interconnects. Logic blocks can be programmed to perform the function of basic logic gates such as AND, and XOR, or more complex combinational functions such as decoders or mathematical functions. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory. A hierarchy of programmable interconnects allows logic blocks to be interconnected as needed by the system designer, somewhat like a one-chip programmable breadboard. Logic blocks and interconnects can be programmed by the customer or designer, after the FPGA is manufactured, to implement any logical function—hence the name "field-programmable".

FPGAs are usually slower than their application-specific integrated circuit (ASIC) counterparts, cannot handle as complex a design, and draw more power (for any given semiconductor process). But their advantages include a shorter time to market, ability to re-program in the field to fix bugs, and lower non-recurring engineering costs. Vendors can sell cheaper, less flexible versions of their FPGAs which cannot be modified after the design is committed. The designs are developed on regular FPGAs and then migrated into a fixed version that more resembles an ASIC. To configure ("program") an FPGA we specify how we want the chip to work with a logic circuit diagram or a source code using a hardware description language (HDL). The HDL form might be easier to work with when handling large structures because it's possible to just specify them numerically rather than having to draw every piece by hand. On the other hand, schematic entry might allow for a more tight specification of what you want. For this purpose VHDL and Verilog HDL are popular. SystemC is also in the race and popular with embedded systems designers. Going from schematic/HDL source files to actual configuration, the source files are fed to a software suite from the FPGA vendor that through different steps will produce a file. This file is then transferred to the FPGA via a serial interface or USB (JTAG) interface or to external memory device like an EEPROM. . The literature survey revealed that many companies have done extensive research on this part of video encoding. The HDL code for motion estimation core is not easily available on the World Wide Web. So the code was written from scratch for this project after comprehensive literature survey. For simulation purposes a smaller test image is used and due to the generic nature of the explained architecture it can be extended to test larger images but with more complex debugging equipments and techniques.

Digital video compression entails the utilization of many coding techniques with the ultimate goal to reduce the size of the digital representation of a video sequence. The same techniques used to compress digital pictures, e.g., in the JPEG picture standard, can be applied to single video frames. Such techniques exploit the fact that colors in photographic images tend to only gradually change when traversed from one side to another. In the video coding case, the fact that subsequent video frames do not differ much can be similarly exploited in order to increase compression efficiency. All coding techniques can be categorized into two main categories, namely lossy and lossless techniques. Lossy coding techniques remove pel information that the human eye is unable to perceive using coding techniques such as the discrete cosine transform and quantization. The information that has been removed in most cases cannot be exactly regained, but it usually can only be approximated. On the other hand, lossless coding techniques do not remove any information. Instead, it exploits redundancies, i.e., similarities, between pels found in and between video frames which results in the representation of pel information using fewer bits. A lossless coding technique is predictive coding which predicts *current* pel(s) using *reference* pel(s) and then store the difference(s) between the prediction and the current pel(s). Assuming redundancy between pels, the differences are usually small and can be coded using less bits than the coding of the original pels. Predictive coding can use pels from the same video frame as reference pels (intra-coding) or pels from other video frames (intercoding). Inter-frame predictive coding can contribute to the overall compression efficiency, because consecutive video frames are usually similar, i.e., they do not differ much. In this sense, the reference pels can be found in a reference frame located at the same position as the current pels in the current to be coded frame. This approach can also be used to capture scene changes by choosing the reference frames in the near future of the current (to be encoded) frame instead from its past. However, such a straightforward approach has one major drawback. Objects in a video scene tend to move around resulting in poor compression performance of the straightforward inter-frame predictive coding method, because pels located at the same location in consecutive frames are now quite different.

In video coding, similarities between video frames can be exploited to achieve higher compression ratios. However, moving objects within a video scene diminish the compression efficiency of the straightforward approach that only considers pels located at the same position in the video frames. In order to achieve higher compression efficiency, motion estimation was introduced in an attempt to accurately capture such movements. It is performed for every macroblock, i.e., an array of 16x16 pels, in the to be encoded frame by finding its 'best' match in a reference frame. The most commonly used metric is the "Mean Absolute Differences" (MAD), which adds up the absolute differences between corresponding elements in the macroblocks. The MAD operation is very time-consuming due to the complex nature of the absolute operation and the subsequent multitude of additions. In [3], a parallel hardware implementation was proposed to speed up the MAD computation process.

Motion Estimator is one such module deserving a particular attention in the scope of digital video coding. This block enjoys its own independence as it is not constrained by any video coding protocols and its functionality is solely based upon the designers' creativity, need and application. In fact, although this block is often regarded as one of the most important operations in video coding to exploit temporal redundancies in sequences of images, it often represents most of the computation cost of these systems [1]. As a consequence, real-time Motion Estimation is usually only achievable by adopting specialized VLSI structures to implement this processing task. Motion Estimation in MPEG video is a temporal prediction technique. The basic principle of motion estimation is that in most cases, consecutive video frames will be similar except for changes induced by objects moving within the frames. In the trivial case of zero motion between frames (and no other differences caused by noise, etc.), it is easy for the encoder to efficiently predict the current frame as a duplicate of the prediction frame. In such as case, the only information necessary to transmit to the decoder becomes the syntactic overhead necessary to reconstruct the picture from the original reference frame. When there is motion in the images, the situation is not as simple.

Motion estimation techniques form the core of video compression and video processing applications. Motion estimation extracts motion information from the video sequence. The motion is typically represented using a motion vector (x, y). The motion vector indicates the displacement of a pixel or a pixel block from the current location due to motion. Motion information is used in video compression to find best matching block in reference frame to calculate low energy residue, used in scan rate conversion to generate temporally interpolated frames. It is also used in applications such motion compensated de-interlacing, video stabilization, motion tracking etc. Varieties of motion estimation techniques are available. There are pel-recursive techniques, which derive motion vector for each pixel. There is the phase plane correlation technique, which generates motion vectors via correlation between current frame and reference frame. The computational complexity of a motion estimation technique can then be determined by three factors: Search algorithm, Cost function/evaluate function and Search range parameter.

Actually, we can reduce the complexity of the motion estimation algorithms by reducing the complexity of the applied search algorithm and/or the complexity of the selected cost function. A full search algorithm evaluates all the weights in the search window, and a more efficient, less complex search algorithm will decrease the search space. Intuitively, one might expect that the ideal processor for reducing temporal redundancy is one that tracks every pixel from frame to frame. This is computationally intensive, and such methods do not provide reliable tracking due to presence of noise in frames. Instead of tracking individual pixels from frame to frame, video coding standards only allow tracking information for 16x16 pixel regions, commonly referred to as macroblocks [1]. The macroblock dimension of 16x16 is chosen because it provides a good compromise between providing efficient temporal redundancy reduction and requiring moderate computational requirements.

Let the two consecutive frames in Fig.1.1 be denoted as frame (t - 1) and frame (t). In the first stage, we segment frame (t) into non-overlapping 16x16 pixel regions (macroblocks), and for each 16x16 block we determine a corresponding 16x16 pixel region in frame (t-1).

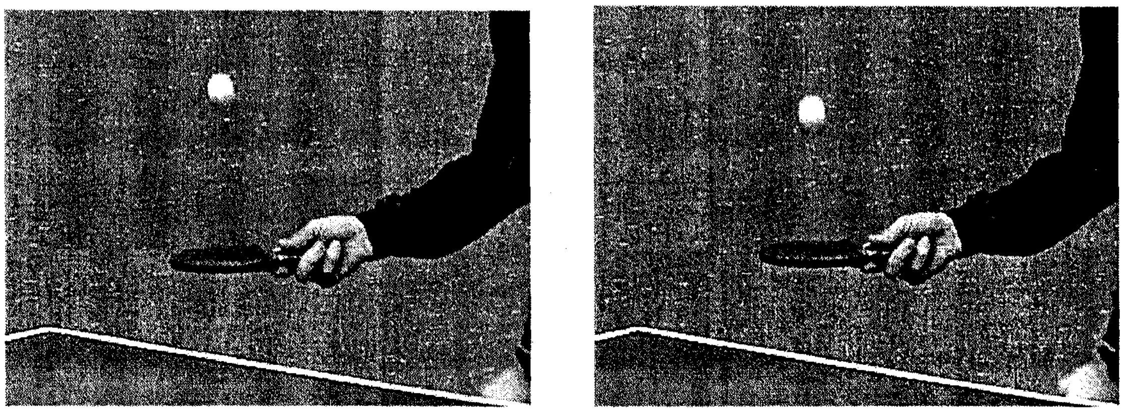

Fig. 1.1 Illustration of two consecutive frames

Using corresponding 16x16 pixel region from frame (t-1), the temporal redundancy reduction processor generates a representation for frame (t) that contains only the changes between the two frames. If the two frames have a high degree of temporal redundancy, then the difference frame would have a large number of pixels that have values near zero [1]. For example, in Fig.1.1, there is a high degree of temporal redundancy, as evidenced by the similarity of features in both frames. On the other hand, if frame (t) were completely different than frame (t-1), then the temporal redundant reduction processor may fail to corresponding regions between two frames. The other techniques will be discussed in detail in the later part of this chapter. The most popular technique is Block Matching Algorithm. The implementation described here uses Block Matching Algorithm. The implementation is based upon a proposed motion estimation accelerator module in [3, 5]. The extension is implemented with a special hardware for alignment of reference frames and the required control circuitry. A 32x32 pixel search area of the reference frame is used as standard for each current frame. The implementation differs from [3, 5] due to the pipelining approach which considerably reduces the total computation time for finding the best match.

1.2 Block matching algorithm theory

1.2.1 Introduction

Block Matching Algorithm (BMA) is the most popular motion estimation algorithm. Block Matching Algorithm calculates motion vector for an entire block of pixels instead of individual pixels. The same motion vector is applicable to all the pixels in the block. This reduces computational requirement and also results in a more accurate motion vector since the objects are typically a cluster of pixels. Block Matching Algorithm is illustrated in Fig. 1.2 [14]. The current frame is divided into pixel blocks and motion estimation is performed independently for each pixel block. Motion estimation is done by identifying a pixel block from the reference frame that best matches the current block, whose motion is being estimated. The reference pixel block is generated by displacement from the current block's location in the reference frame. The displacement is provided by the Motion Vector (MV). MV consists of is a pair (x, y) of horizontal and vertical displacement values.

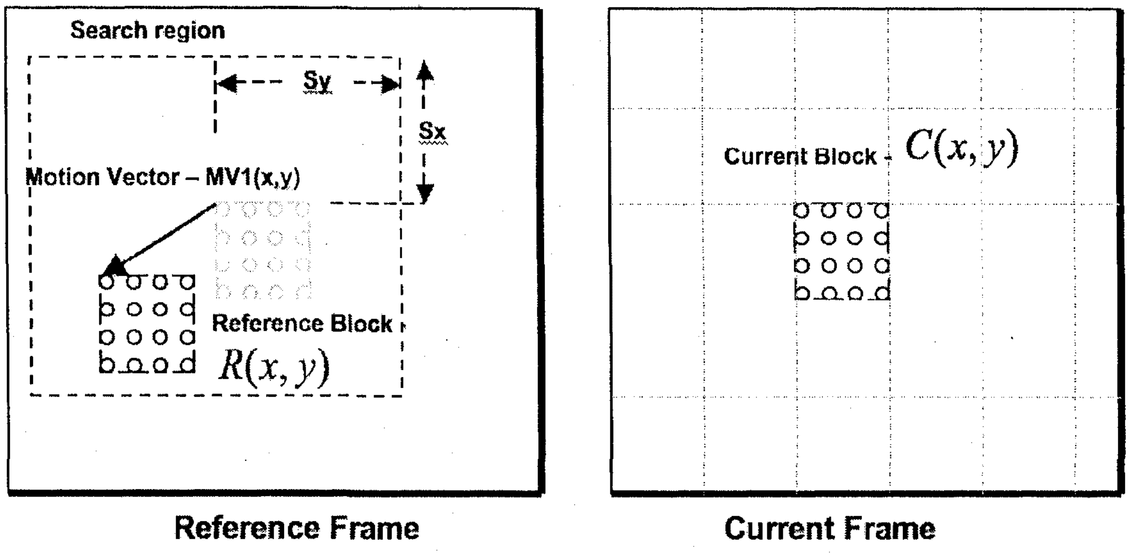

Fig. 1.2 Illustration of Block Matching Algorithm (BMA)

In video coding terminology, the match is being performed between rectangular regions; this is referred to as a block matching criterion, and search techniques to find the motion vectors, that yield the smallest Mean Absolute Difference (MAD), are referred to as Block Matching Algorithms. There are various criteria available for calculating block matching. We focus ourselves to MAD. Let the pixels of the macroblock in the current frame be denoted as C (x+k, y+l) and the pixels in the reference frame be denoted as R(x+i+k, y+j+l). The cost function becomes

Mean Absolute Difference (MAD) =

$$\frac{1}{MN}\sum_{k=0}^{M-1}\sum_{l=0}^{N-1}\left| (C(x+k,y+l)-R(x+i+k,y+j+l))\right| \qquad \text{(Eq 1.1)}$$

Sum of Absolute Differences (SAD) =

$$\sum_{k=0}^{M-1} \sum_{l=0}^{N-1} (C(x+k, y+l) - R(x+i+k, y+j+l)) \Big| \qquad \text{(Eq 1.2)}$$

In video coding standards, N = M = 16. The best matching block is the block R(x+i, y+j) for which MAD (i, j) is minimized. Thus, the coordinates (i, j) for which MAD is minimized define the motion vector. Basically, MAD is obtained by dividing SAD by the product of MN i.e. 256. In hardware it indicates a shift of value to the right by 8 positions, since $2^8 = 256$. MAD provides fairly good match at lower computational requirement. Hence it is widely used for block matching. There are various other criteria also available such as cross correlation, maximum matching pixel count etc. The reference pixel blocks are generated only from a region known as the search area. Search region defines the boundary for the motion vectors and limits the number of blocks to evaluate. The height and width of the search region is dependant on the motion in video sequence. The available computing power also determines the search range. Bigger search region requires more computation due to increase in number of evaluated candidates. Typically the search region is kept wider (i.e. width is more than height) since many video sequences often exhibit panning motion. The search region can also be changed adaptively depending upon the detected motion.

1.2.2 Full search block matching algorithm

Among all the BMAs, Full-Search Block Matching Algorithm (FSBMA) is the most popular. FSBMA evaluates every possible pixel block in the search region. Hence, it can generate the best block matching motion vector. This type of BMA can give least possible residue for video compression. But, the required computations are prohibitively high due to the large amount of candidates to evaluate. For typical values for broadcast TV (I = 720, J = 480 and F = 30), motion estimation based on full-search algorithm requires 29.89 GOPS (Giga operations per second) for a search area of 32x32 pixels [1].

The FSBMA is usually used in the hardware implementation of Motion Estimation, because of its simplicity, regularity, and optimum result. The most commonly used metric to determine the best match for FSBMA in hardware is the Mean Absolute Differences. Main goal is to compute the minimum MAD from among all the candidate blocks. To do this, search iteration is performed for each candidate block. The MAD adds up the absolute differences between corresponding elements in the candidate and reference block. The MAD cost function is described in Equation (1).

Field Programmable Gate Arrays supports a high number of processor elements (PE) in parallel mode. This property can be used to process, at the same time, all SAD operations from a MPEG macroblock in a search area. With this real time video encoder for Motion Estimation can be reached [15].

1.2.3 2D logarithmic search

2D Logarithmic search is very similar to binary search and it tests limited candidates. In the first step, the [-p, p] search rectangle is divided into two areas: one inside a $\left[\frac{-p}{2}, \frac{p}{2}\right]$ (at integer pixel location) rectangle and one outside it. Furthermore, instead of searching the whole area, the Block Matching Criteria is computed for nine locations: at (0, 0) and at the eight major points in the perimeter of the area. That is, if the distance between these points is d_1, we compute the minimum at (0, 0), (0, d_1), (0, $-d_1$), ($-d_1$, 0), (d_1, 0), (d_1, d_1), (d_1, $-d_1$) and ($-d_1$, $-d_1$). The distance d_1 is given by $d_1 = 2^{k-1}$, where k =

[$\log_2 p$]. For example for $p = 7$, $k = 3$, $d_1 = 4$ pixels. Using the best match location as the starting point, we then look for the best match in the eight perimeter points at distance d_2 which is $d_1/2$. We continue this process until the kth search, where the eight perimeter search locations are spaced by one point. After these eight locations have been examined, we determine the location that yields the smallest criteria.

As shown in Fig. 1.3, during the first iteration, a total of five candidates are tested. The candidates are centered around the current block location in a diamond shape. The step size for first iteration is set equal to half the search range. For the second iteration, the centre of the diamond is shifted to the best matching candidate. The step size is reduced by half only if the best candidate happens to be the centre of the diamond. If the best candidate is not the diamond centre, same step size is used even for second iteration. In this case, some of the diamond candidates are already evaluated during first iteration. Hence, there is no need for block matching calculation for these candidates during the second iteration. The results from the first iteration can be used for these candidates. The process continues till the step size becomes equal to one pixel. For this iteration all eight surrounding candidates are evaluated. The best matching candidate from this iteration is selected for the current block. The number of evaluated candidate is variable for the 2D logarithmic search. However, the worst case and best case candidates can be calculated. For $I = 720$, $J = 480$, $F = 30$ and search area of 32x32, logarithmic search requires one GOP. The complexity of logarithmic search is only 3.3 percent of the complexity of full search [1]

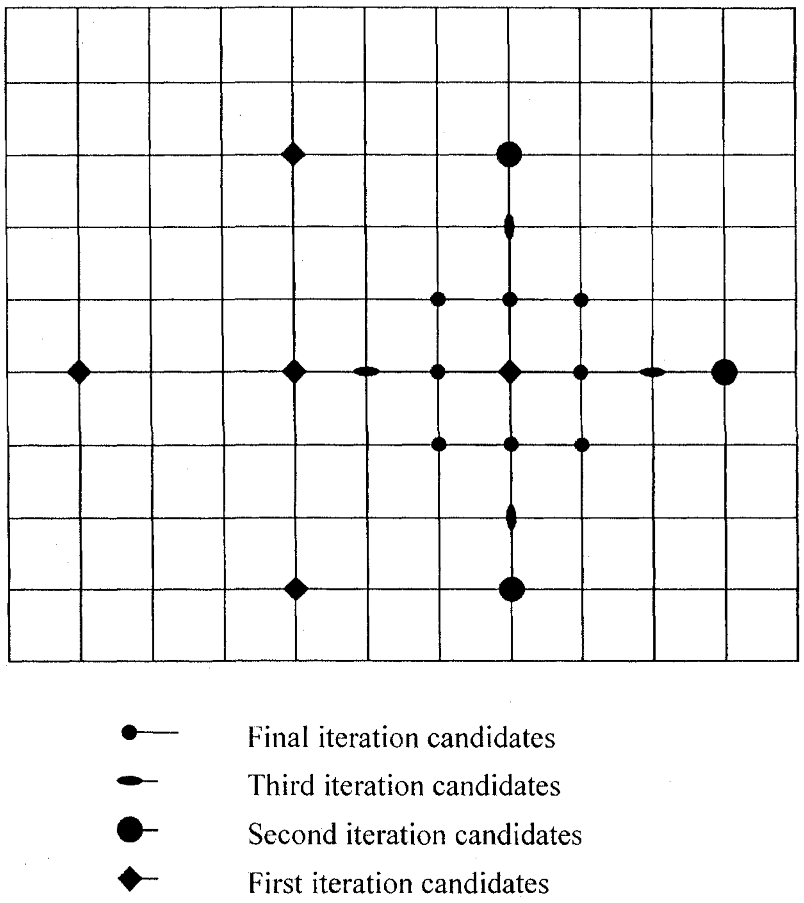
Final iteration candidates
Third iteration candidates
Second iteration candidates
First iteration candidates

Fig. 1.3 2D Logarithmic search

1.2.4 Three step search

In a three-step search (TSS) algorithm, the first iteration evaluates nine candidates as shown in Fig.1.4. The candidates are centered around the current block's position. The step size for the fist iteration is typically set to half the search range. During the next iteration, the search centre is shifted to the best matching candidate from the first iteration. Also, the step size is reduced by half. The same process continues till the step size becomes equal to one pixel. This is the last iteration of the three-step search algorithm. The best matching candidate from this iteration is selected as the final candidate. The motion vector corresponding to this candidate is selected for the current block. The number of candidates evaluated during three-step search is very less compared to the full search algorithm. The number of evaluated candidate is fixed depending upon the step size set during the first iteration. For example, the computational complexity associated with 25 search locations is 777.6 MOPS (Million operations per second) [1].

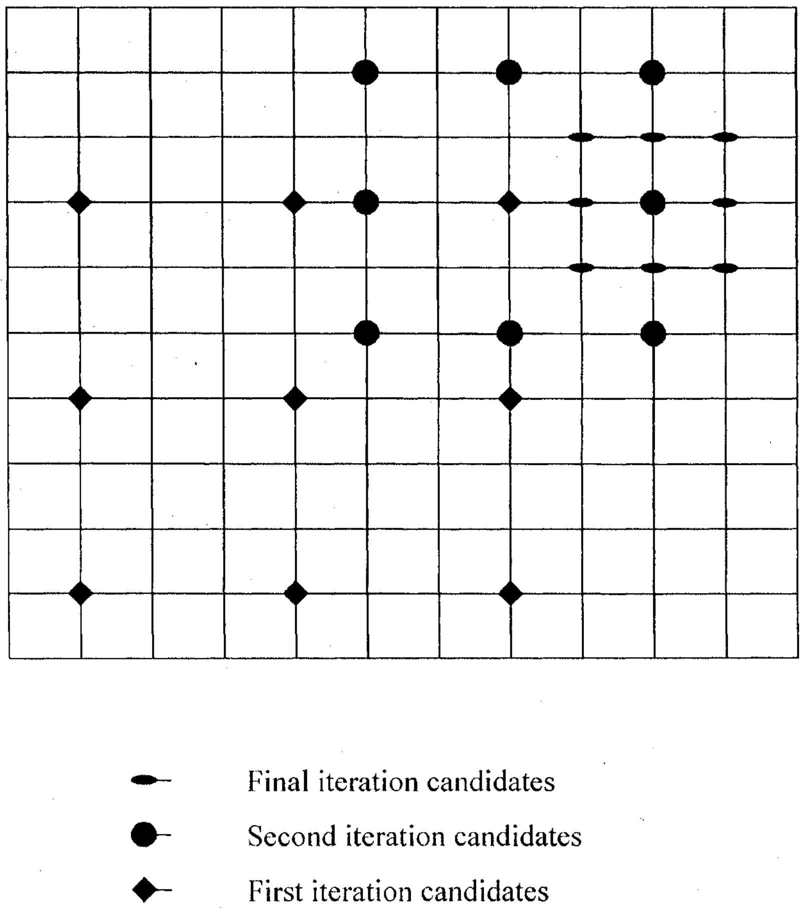
Final iteration candidates
Second iteration candidates
First iteration candidates

Fig. 1.4 Three step search

1.2.5 Parallel Hierarchical One-Dimensional Search (PHODS)

Unlike the logarithmic search, the search in this search strategy is done independently along the two dimensions. The search algorithm is as follows:

1. For a [-p, p] search region let $S = 2^{[\log 2p]}$ and set the origin of the search space at search location (0,0). Denote the origin as (di, dj).
2. In parallel, compute the
    a. i-axis local minimum: Among the three locations (di–S,dj), (di,dj), (di+S,dj), find the location that yields the smallest MAD. Set dj to the j coordinate of this location.
    b. j-axis local minimum: Among the three locations (di,dj-S), (di,dj), (di,dj+S), find the location that yields the smallest MAD. Set dj to the j coordinate of this location.

Set $S = \frac{S}{2}$.

Repeat step 2, until S= 0. The final (di,dj) is the motion vector that yields the best match for the macroblock in the current picture. For the case of p = 7, we need to examine 13 search locations, which for frames at 720 x 480 resolution and 30 frames/s corresponds to 404.35 MOPS. Parallel Hierarchical One-Dimensional Search has two distinct advantages over TSS: (1) the MAD calculations are parallelizable, and (2) it has regular data flow, since the search locations are always along the i-axis and the j-axis.

Both logarithmic and the PHODS methods belong to the class of fast algorithms that reduce motion estimation complexity by reducing the number of search locations that are used in determining the minimum MAD. For p = 7, compared to full search method, the complexity is reduced from 6.99 GOPS to 404.35 MOPS [1]. Fast algorithms that work in reduced search space assume that MAD (i,j) increases monotonically as the search area moves away from the best matched location. Such algorithms perform as well as the full-search method if this assumption holds; however, in practice the assumption often fails, since not all the search locations are visited and the search for a global minimum may get trapped into a local minimum. Moreover, it is easy to parallelize Full-Search architectures whereas logarithmic algorithms require complex control mechanisms.

CHAPTER 2

BACKGROUND OF MPEG

2.1 Background and overview

Video pictures in today's digital era pose a problem of compression. Uncompressed digital video pictures take up enormous amounts of information. If you were to record digital video to a CD without compression, it could only hold about five minutes, and that's without any sound. MPEG standards reduce the amount of data needed to represent video, at the same time manages to retain very high picture quality. The Moving Picture Experts Group, commonly referred to as simply MPEG, is a working group of International Organization for Standardization (ISO)/ International Electrotechnical Commission (IEC) charged with the development of video and audio encoding standards. Its first meeting was in May of 1988 in Ottawa, Canada. As of late, MPEG has grown to include approximately 350 members per meeting from various industries, universities, and research institutions. MPEG's official designation is ISO/IEC JTC1/SC29 WG11. ISO/IEC JTC 1 is Joint Technical Committee 1 of the ISO and the IEC. It deals with all matters of information technology. MPEG has standardized the following compression formats and ancillary standards:

- MPEG-1: Initial video and audio compression standard. Later used as the standard for Video CD, and includes the popular Layer 3 (MP3) audio compression format.
- MPEG-2: Transport, video and audio standards for broadcast-quality television. Used for over-the-air digital television ATSC, DVB and ISDB, digital satellite TV services like Dish Network, digital cable television signals, SVCD, and with slight modifications, as the .VOB (Video OBject) files that carry the images on DVDs.
- MPEG-3: Originally designed for HDTV, but abandoned when it was realized that MPEG-2 (with extensions) was sufficient for HDTV.
- MPEG-4: Expands MPEG-1 to support video/audio "objects", 3D content, low bitrate encoding and support for Digital Rights Management.

In addition, the following standards, while not sequential advances to the video encoding standard as with MPEG-1 through MPEG-4, are referred to by similar notation:

- MPEG-7: A multimedia content description standard.
- MPEG-21: MPEG describes this standard as a multimedia framework.

MPEG compresses high data imagery and slightly affects the picture quality, which is not notable to the human eye. The illusion of movement in TV and cinema pictures is actually created by showing a sequence of still pictures in quick succession, each picture changing a small amount from the one before. We cannot detect the individual pictures - our brain 'smoothes' the action out. A dumb analogue TV picture sends every part of every picture, but digital MPEG video is much smarter. It looks at two pictures and works out how much of the picture is the same in both. Because pictures don't change much from one to the next, there is quite a lot of repetition. The parts that are repeated don't need to be saved or sent, because they already exist in a previous picture. These parts can be thrown out. Digital video also contains components our eyes can't see, so these can be thrown out as well. MPEG-2 is a popular coding and decoding standard for digital video data. MPEG-2 encoding uses both lossy compression and lossless compression. Lossy compression permanently eliminates information from a video based on a human perception model. Humans are much better at discerning changes in color intensity (luminance information) than changes in color (chrominance information). Humans are also much more sensitive to low frequency image components, such as a blue sky, than to high frequency image components, such as a plaid shirt. Details which humans are likely to miss can be thrown away without affecting the perceived video quality. Lossless compression eliminates redundant information while allowing for its later reconstruction. Similarities between adjacent video pictures are encoded using motion prediction, and all data is Huffman compressed. The amount of lossy and lossless compression depends on the video data. Common compression ratios range from 10:1 to 100:1. Certain sections of video are more complicated than other sections. When there is lots of action and fine detail it's much more difficult to encode properly than a slow moving scene with large areas of the same color or texture in the picture. MPEG deals with this by concentrating its efforts and data use on the complicated parts. This means that the video is encoded in the best possible way. In MPEG, video is represented as a sequence of pictures, and each picture is treated as a two-dimensional array of pixels (pels). The color of each pel is consists of three components: Y (luminance), Cb and Cr (two chrominance components).

The process can be explained in short as following: The encoder operates on a sequence of pictures. Each picture is made up of pixels arranged in a 16x16 array known as a macroblock. Macroblocks consist of a 2x2 array of blocks (each of which contains an 8x8 array of pixels). There is a separate series of macroblocks for each color channel, and the macroblocks for a given channel are sometimes downsampled to a 2x1 or 1x1 block matrix. The compression in MPEG is achieved largely via motion estimation, which detects and eliminates similarities between macroblocks across pictures. Specifically, the motion estimator calculates a motion vector that represents the horizontal and vertical displacement of a given macroblock (i.e., the one being encoded) from a matching macroblock-sized area in a reference picture. The matching macroblock is removed (subtracted) from the current picture on a pixel by pixel basis, and a motion vector is associated with the macroblock describing its displacement relative to the reference picture. The result is a residual predictive-code (P) picture. It represents the difference between the current picture and the reference picture. Reference pictures encoded without the use of motion prediction are intra-coded (I) pictures. In addition to forward motion prediction, it is possible to encode new pictures using motion estimation from both previous and subsequent pictures. Such pictures are bidirectionally predictive-coded (B) pictures, and they exploit a greater amount of temporal locality. Each of I, P, and B pictures then undergoes a 2-dimensional discrete cosine transform (DCT) which separates the picture into parts with varying visual importance. The input to the DCT is one block. The output of the DCT is an 8x8 matrix of frequency coefficients. The upper left corner of the matrix represents low frequencies, whereas the lower right corner represents higher frequencies. The latter are often small and can be neglected without sacrificing human visual perception. The DCT coefficients are quantized to reduce the number of bits needed to represent them. Following quantization, many coefficients are effectively reduced to zero. The DCT matrix is then run-length encoded by emitting each non-zero coefficient, followed by the number of zeros that precede it, along with the number of bits needed to represent the coefficient, and its value. The run-length encoder scans the DCT matrix in a zig-zag order to consolidate the zeros in the matrix. Finally, the output of the run-length encoder, motion vector data, and other information (e.g., type of picture), are Huffman coded to further reduce the average number of bits per data item. The compressed stream is sent to the output device.

In order to achieve high compression ratio, we must use hybrid coding techniques to reduce both spatial redundancy and temporal redundancy. In the MPEG coding, there are two kinds of blocks: the 16x16 (pels) macro-block and the 8x8 (pels) basic block. The basic block is used when the DCT is performed and the macro-block is used for motion estimation. The encoding of a video stream is done in several steps. Each of the steps separately depicted in Fig. 2.1 are explained below. In the Fig. 2.1, DCT stands for Discrete Cosine Transform, Q for Quantization, IDCT for Inverse Discrete Cosine Transform, IQ for Inverse Quantization. The FRAMES block stores two frames at a time i.e. currently encoded frame and a previously encoded frame. These two frames are used to estimate the motion occurred between two consecutive frames of the video sequence.

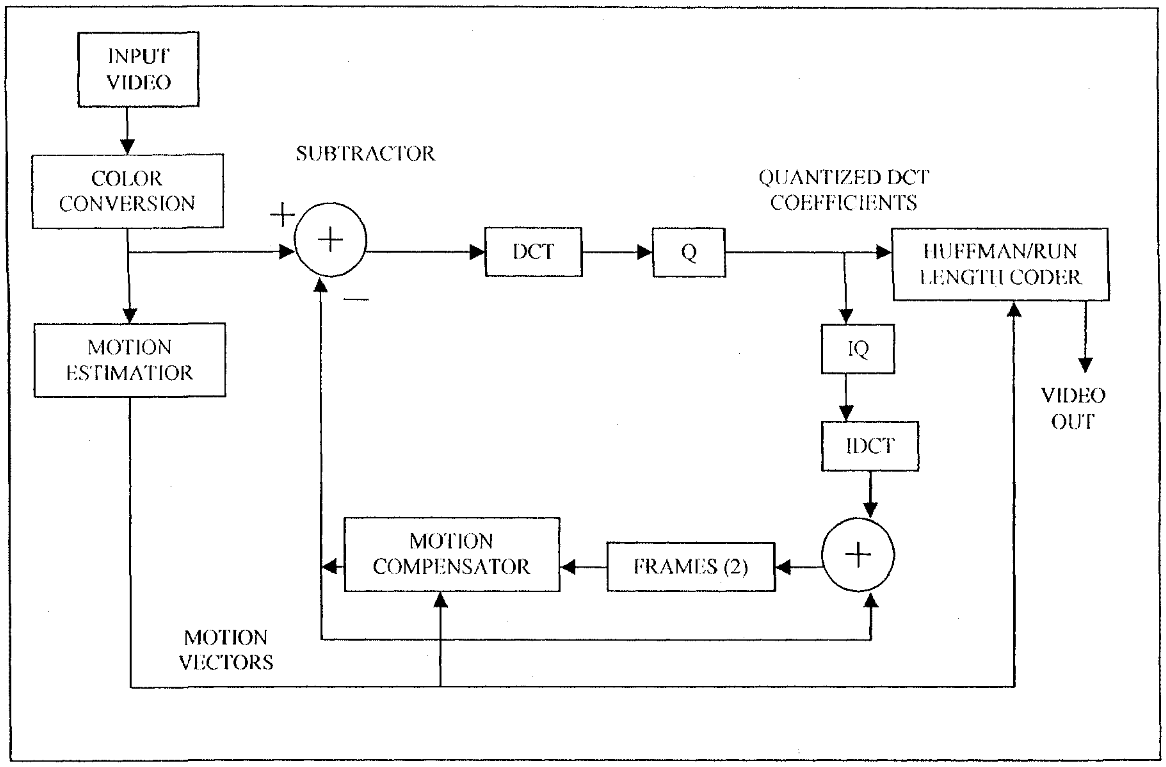

Fig. 2.1 Video encoder block diagram

2.1.1 Color conversion

In this step the input color-space is transformed into the YCbCr color-space. Furthermore, the chrominances are subsampled by a factor of two in both the horizontal and vertical direction. Thus, a 16x16 block from the video signal results in four 8x8 luminance blocks, one 8x8 Cb block, and one 8x8 Cr block. These 8x8 blocks are used by the DCT. The 16x16 luminance block is used by the motion estimation.

2.1.2 Motion estimator

Motion estimation is the process of determining motion vectors that describe the transformation from one 2D image to another; usually from adjacent frames in a video sequence. It is an ill-posed problem as the motion is in three dimensions but the images are a projection of the 3D scene onto a 2D plane. The motion vectors may relate to the whole image (global motion estimation) or specific parts, such as rectangular blocks, arbitrary shaped patches or even per pixel. The motion vectors may be represented by a translational model or many other models that can approximate the motion of a real video camera, such as rotation and translation in all three dimensions and zoom. In motion estimation an exact 1:1 correspondence of pixel positions is not a requirement. Applying the motion vectors to an image to synthesize the transformation to the next image is called Motion compensation. The combination of motion estimation and motion compensation is a key part of video compression as used by MPEG 1, 2 and 4 as well as many other video codecs.

2.1.3 Motion compensator

One method used by various video formats to reduce file size is motion compensation. For many frames of a movie, the only difference between one frame and another is the result of either the camera moving or an object in the frame moving. In reference to a video file, this means much of the information that represents one frame will be the same as the information used in the next frame. Motion compensation takes advantage of this to provide a way to create frames of a movie from a reference frame. For example, in principle, if a movie is shot at 24 frames per second, motion compensation would allow the movie file to store the full information for every fourth frame. The only information stored for the frames in between would be the information needed to transform the previous frame into the next frame. If a frame of information is one MB in size, then uncompressed, one second of this film would be 24 MB in size. Using motion compensation, the file size for one second of the film could be reduced to a little over 6 MB. More formally, in video compression, motion compensation is a technique for describing a picture in terms of the transformation of a reference picture to the current picture. The reference picture may be previous in time or even from the future. When images can be accurately synthesized from previously transmitted/stored images then the compression efficiency can be improved. In MPEG, images are predicted from previous frames (P frames) or bidirectionally from previous and future frames (B frames). B frames are not so popular because the image sequence must be transmitted/stored out of order so that the future frame is available to generate the B frames. After predicting frames using motion compensation, the coder finds the error (residual) which is then compressed using the DCT and transmitted.

In block motion compensation (BMC), the frames are partitioned in blocks of pixels (e.g. macroblocks of 16×16 pixels in MPEG). Each block is predicted from a block of equal size in the reference frame. The blocks are not transformed in any way apart from being shifted to the position of the predicted block. This shift is represented by a motion vector. To exploit the redundancy between neighboring block vectors, (e.g. for a single moving object covered by multiple blocks) it is common to encode only the difference between the current and previous motion vector in the bit-stream. The result of this differencing process is mathematically equivalent to global motion compensation capable of panning. Further down the encoding pipeline, an entropy coder will take advantage of the resulting statistical distribution of the motion vectors around the zero vector to reduce the output size. It is possible to shift a block by a non-integer number of pixels, which is called sub-pixel precision. The in-between pixels are generated by interpolating neighboring pixels. Commonly, half-pixel or quarter pixel precision is used. The computational expense of sub-pixel precision is much higher due to the extra processing required for interpolation and on the encoder side, a much greater number of potential source blocks to be evaluated.

The main disadvantage of block motion compensation is that it introduces discontinuities at the block borders (blocking artifacts). These artifacts appear in the form of sharp horizontal and vertical edges which are easily spotted by the human eye and produce ringing effects (large coefficients in high frequency sub-bands) in the Fourier-related transform used for transform coding of the residual frames.

Block motion compensation divides up the current frame into non-overlapping blocks, and the motion compensation vector tells where those blocks come from (a common misconception is that the previous frame is divided up into non-overlapping blocks, and the motion compensation vectors tell where those blocks move to). The source blocks typically overlap in the source frame. Some video compression algorithms assemble the current frame out of pieces of several different previously-transmitted frames. Frames can also be predicted from future frames. The future frames then need to be encoded before the predicted frames and thus, the encoding order does not necessarily match the real frame order. Such frames are usually predicted from two directions, i.e. from the I- or P-frames that immediately precede or follow the predicted frame. These bidirectionally predicted frames are called B-frames. A coding scheme could, for instance, be IBBPBBPBBPBB.

2.1.4 Discrete Cosine Transform (DCT) and Inverse Discrete Cosine Transform (IDCT)

DCT is a lossy compression scheme where an N x N image block is transformed from the spatial domain to the DCT domain. DCT decomposes the signal into spatial frequency components called DCT coefficients [23]. The lower frequency DCT coefficients appear toward the upper left-hand corner of the DCT matrix, and the higher frequency coefficients are in the lower right-hand corner of the DCT matrix. The Human Visual System (HVS) is less sensitive to errors in high frequency coefficients than it is to lower frequency coefficients. Because of this, the higher frequency components can be more finely quantized, as done by the quantization matrix. Each value in the quantization matrix is pre-scaled by multiplying by a single value, known as the quantizer scale code. This value can range in value from one to 112 and is modifiable on a macroblock basis. Dividing each DCT coefficient by an integer scale factor and rounding the results accomplishes quantization. This sets the higher frequency coefficients (in the lower right corner), that are less significant to the compressed picture, to zero by quantizing in larger steps. The low frequency coefficients (in the upper left corner), are more significant to the compressed picture, and are quantized in smaller steps. The goal of quantization is to force as many of the DCT coefficients to zero, or near zero, as possible within the boundaries of the prescribed bit-rate and video quality parameters. Thus, since quantization throws away some information, it is a lossy compression scheme.

The data compressed at the transmitter needs to be decompressed at the receiver. IDCT is used to decompress DCT compressed data in the decoder. DCT and IDCT are two of the most computation intensive funtions in compression. Therefore, a fast and optimized DCT/IDCT implementation is essential in improving the performance of the video coder and decoder.

2.1.5 Quantization and Inverse Quantization

Quantization is done to achieve better compression. Quantization reduces the number of bits needed to store information by reducing the size of the integers representing the information in the scene. These are details that the human visual system ignores. This step represents one key segment in the multi- compression process. A reduction in the number of bits reduces storage capacity needed, improves bandwidth, and lowers implementation costs. Quantization is the process of selectively discarding visual information without a significant loss in the visual effect. Quantization reduces the number of bits needed to store an integer value by reducing the precision of the integer. Each discrete cosine transform (DCT) component is divided by a separate quantization coefficient, and rounded to the nearest integer. The larger the quantization coefficient (i.e., coefficient weighting), the smaller the resulting answer and associated bits needed to express the DCT component. In the reverse process, the fractional bits are "rounded" and are recovered as zeros, constituting a precision loss from the original number. Quantization could be considered as input data binning where the number of bins is less than the number of possible input values. The number of bins is decided by the quantization factor Q. If the input data range is from one to 60, and if Q is 5, then 60/5 is 12 bins (0 to 5, 6 to 10, and so on). A different input data range of 60 is now reduced to 12 possible bins [24]. The quantized Discrete Cosine transform coded coefficients are fed into the quantizer. The quantized coefficients are taken through an inverse quantizer to get back the original DCT coefficients. Since quantizing is a lossy process where certain DCT coefficients are thrown away, the inverse quantization will not given back all of the original 64 DCT coefficients. The non-recovered coefficients have the least visual effect on the picture.

2.1.6 Huffman Coding

Frequently occurring symbols are assigned short code words whereas rarely occurring symbols are assigned long code words. The resulting code string can be uniquely decoded to get the original output of the run length encoder. The code assignment procedure developed by Huffman is used to get the optimum code word assignment for a set of input symbols. The procedure for Huffman coding involves the pairing of symbols. The input symbols are written out in the order of decreasing probability. The symbol with the highest probability is written at the top, the least probability is written down last. The least two probabilities are then paired and added. A new probability list is then formed with one entry as the previously added pair. The least symbols in the new list are then paired. This process is continued till the list consists of only one probability value. The values "0" and "1" are arbitrarily assigned to each element in each of the lists. Fig. 2.2 shows the following symbols listed with a probability of occurrence where: A is 30%, B is 25%, C is 20%, D is 15%, and E = 10% [25].

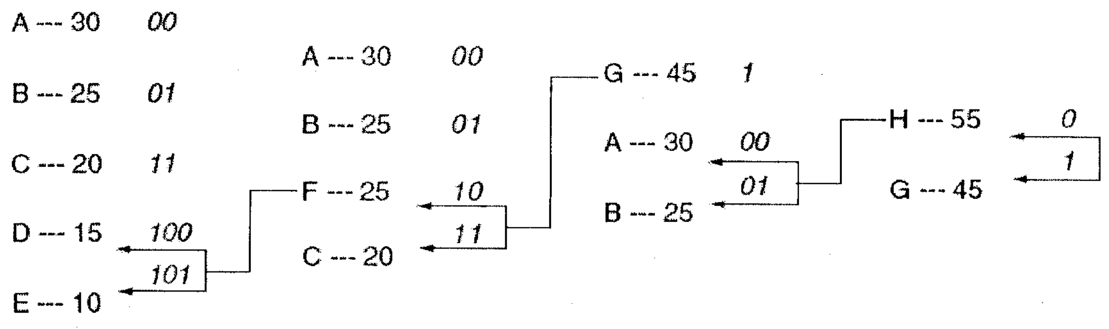

Fig. 2.2 Illustration of huffman coding

Steps in Huffman coding

1. Adding the two least probable symbols gives 25%. The new symbol is F
2. Adding the two least probable symbols gives 45%. The new symbol is G
3. Adding the two least probable symbols gives 55%. The new symbol is H
4. Write "0" and "1" on each branch of the summation arrows. These binary values are called branch binaries.
5. For each letter in each column, copy the binary numbers from the column on the right, starting from the right most column (i.e., in column three, G gets the value "1" from the G in column four.) For summation branches, append the binary from the right-hand side column to the left of each branch binary. For A and C in column three append "0" from H in column four to the left of the branch binaries. This makes A "00" and B "01".

Completing step 5 gives the binary values for each letter: A is "00", B is "01", C is "11", D is "100", and E is "101". The input with the highest probability is represented by a code word of length two, whereas the lowest probability is represented by a code word of length three.

CHAPTER 3

IMPLEMENTATION OF MOTION ESTIMATION HARDWARE ACCELERATOR

3.1 Introduction

The main purpose of this project was to build a lab prototype of a motion estimation hardware accelerator which can be easily mounted to a general purpose RISC processor. There are varieties of implementations described by research teams all around the world, but none provide any modules or codes in implementation and testing of a motion estimator module in hardware. This was the very reason which prompted me to write a Verilog HDL code for the motion estimator hardware accelerator. The HDL approach facilitates reconfigurability and modifiability. The entire code has been written in Verilog HDL. During the course of the project number of problems were faced and tackled. Control circuitry had to be redone a couple of times with respect to optimized control and output. Various algorithms were studied and the performance descriptions provided in [1] gave a clear idea about advantages and disadvantages of each algorithm. The literature survey gave a fair idea of the recent research being carried out in this field [16, 17, 18, 19, 20, and 21]. Though the logarithmic and much superior estimation algorithms assist in achieving faster computations, the complex control associated with them and the probability of error left with the option to choose Full Search Block Matching Algorithm as the candidate for implementation. The advantages like parallelizable structures and ease of implementation of Full Search Block Matching Algorithm described in [1, 2, 15, and 18] makes it an ideal algorithm to be implemented when it comes to FPGA based systems. The very features have been fully utilized in the implementation which is an extension to the proposed SAD motion estimation architecture explained in [4, 5].

The hardware accelerator consists of the SAD module, Current frame control, Reference frame control and the frame storages. The output is the motion vectors which describe where in the 2-dimensional area is the best match found. The SAD module forms the core of the whole system. The following block diagram in Fig. 3.1 gives an idea of different components and the next section explains the working of each module in detail.

3.2 Block diagram description

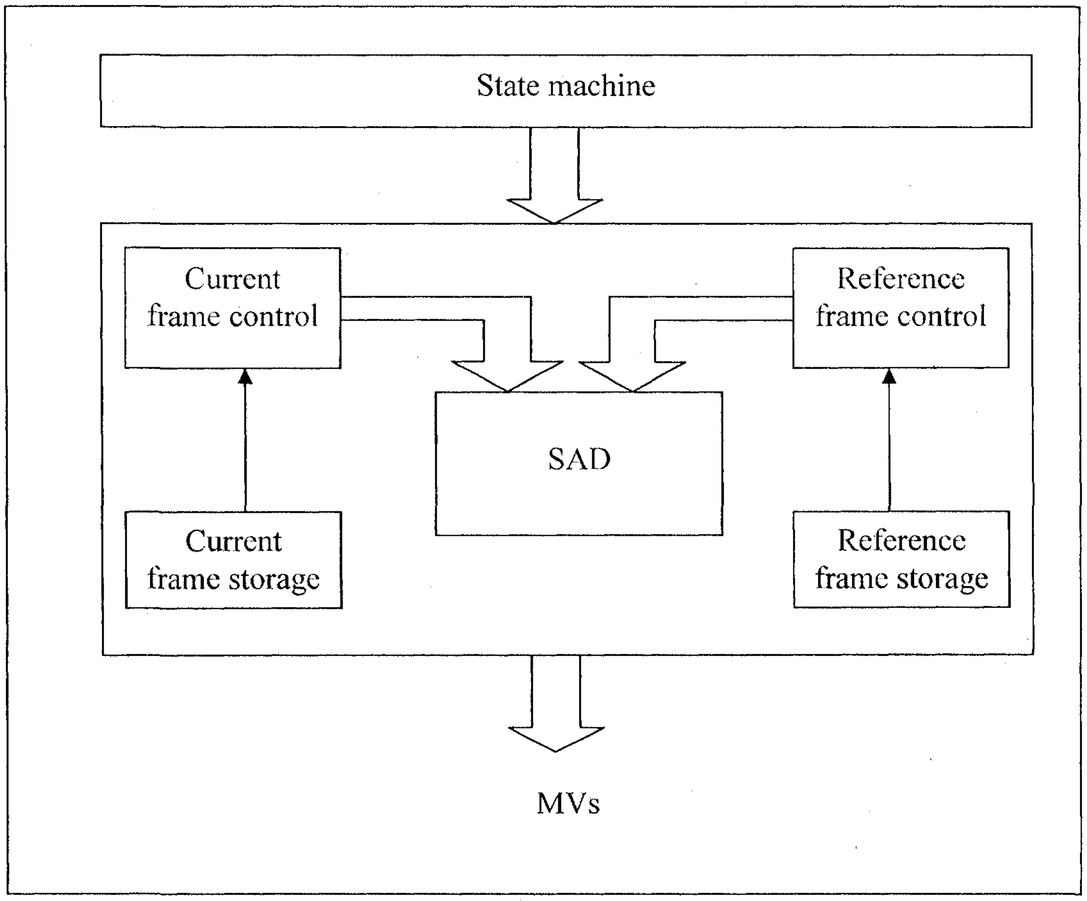

Fig. 3.1 Motion estimation block diagram

3.2.1 Reference frame storage

The current macroblocks are 16x16 blocks which contain the current frame information and have to be compared with the reference macroblocks which are already stored. For this project, MATLAB is used to segregate the macroblocks. But, I will explain it here theoretically to give an idea. The Frame Grabber board [22] which we have has a Video RAM installed on it which stores frames of moving pictures and has a huge FIFO structure to store each pixel one by one in it as explained in [22]. The FIFO read pointer initially points to the 0th location where the luminance (Y) information of the first pixel of the current frame is stored. The Chrominance components (Cb and Cr) of the 1st pixel are stored in 1st and 2nd location. The Frame grabber can be programmed to make the read pointer go to the user desired location which facilitates the segregation of 32x32 search area of the reference frame as well as the 16x16 macroblock of the current frame. Moreover, for motion estimation module only luminance information is used to compute the motion vectors. So the Frame grabber can be programmed to just read and send Luminance pixels of the required 256 bytes. The increment pointer makes it even easier to hop from pixel 16 to pixel 257 to achieve the sliding macroblock effect. For my implementation I have made use of IrfanView ©software to first segregate the individual frames from the movie [8]. Further I have written a code in MATLAB™© to segregate individual current frame into 16x16 macroblocks which can be directly fed into Block RAMs in the FPGA and also to segregate the 32x32 search area of the reference frame. This is just for the functional simulation purposes. Once IrfanView grabs into individual frames from the movie, the MATLAB code segregates the individual macroblocks and search areas into specific text files which are read by the HDL code testbench. The testbench operates in a sequential fashion. So this takes up time during simulation. In real-time the Video RAM buffer has to be used and data has to be read from the same. For this an on-chip or off-chip SDRAM can be used. Normally, all the FPGA vendors sell the SDRAM controller HDL codes as it is very complex and is out of the scope of this project.

3.2.2 Current frame storage

For the current frame storage is done in a similar way as the reference frame storage. The only difference is segregation has to be done for 16x16 macroblocks. It requires less memory and is faster as each time only 256 bytes of luminance pixels have to be read. As explained earlier the testbench method helps in testing the functional simulation but is not helpful in speed up due to its sequential operation. This can be avoided by concurrent operation which can be achieved in case of an on-chip or off-chip memory. If two different RAM modules are used with the read cycles properly synchronized by a FSM then while one port is being utilized to work a macroblock the other port can be used to read the second 16x16 macroblock and store it in the second Current Frame FIFO explained in the next section. Similar can be applied to the Reference Frame Sliding Window Controller explained in the further sections. This will help in nullifying wait times for the SAD module in reading the required data.

To simulate this on-chip memory is used, which in case of FPGAs is the Block RAMs (BRAMs). The BRAMs can be instantiated with the help of the CORE Generator feature in-built in the Xilinx ISE. CORE Generator is a graphical interactive design tool that enables us to create high-level modules such as memory elements, math functions and communications and IO interface cores. We can customize and pre-optimize the modules to take advantage of the inherent architectural features of the Xilinx FPGA architectures, such as Fast Carry Logic, SRL16s, and distributed and block RAM [9].

The instantiation can be done by referring to the text file which is segregated using MATLAB as a Coefficients file (.coe) file [10]. This file loads up to initialize the BRAMs as per the values in the .coe file. For the coefficients file some syntactical rules have to be followed otherwise the CORE generator outputs an error message. After the BRAM is instantiated a my_ram.mif (Memory Initialization Format) file is generated which contains the values we fed as the Coefficients file. Only COE files may be used as inputs to cores for the purpose of specifying initialization values for memory cores and for specifying coefficient values. MIF files can only be generated as output files for use in HDL behavioral simulations. They cannot be used to specify initial values when a core is generated. MIF files will always be written out for memory (in binary format only), based on values specified in any input COE files (or default values, as the case may be).

3.2.3 Reference frame control

This module is a sliding window controller which sweeps across the 32x32 search area i.e. 1024 pixels. The Fig. 3.2 gives a clear picture of the sliding window movement. The Macroblock at each specific point as shown in the figure is latched and fed to the SAD module for further computation. The sliding window gives an accurate account of overall search area. The probability of finding the best match increases in this case.

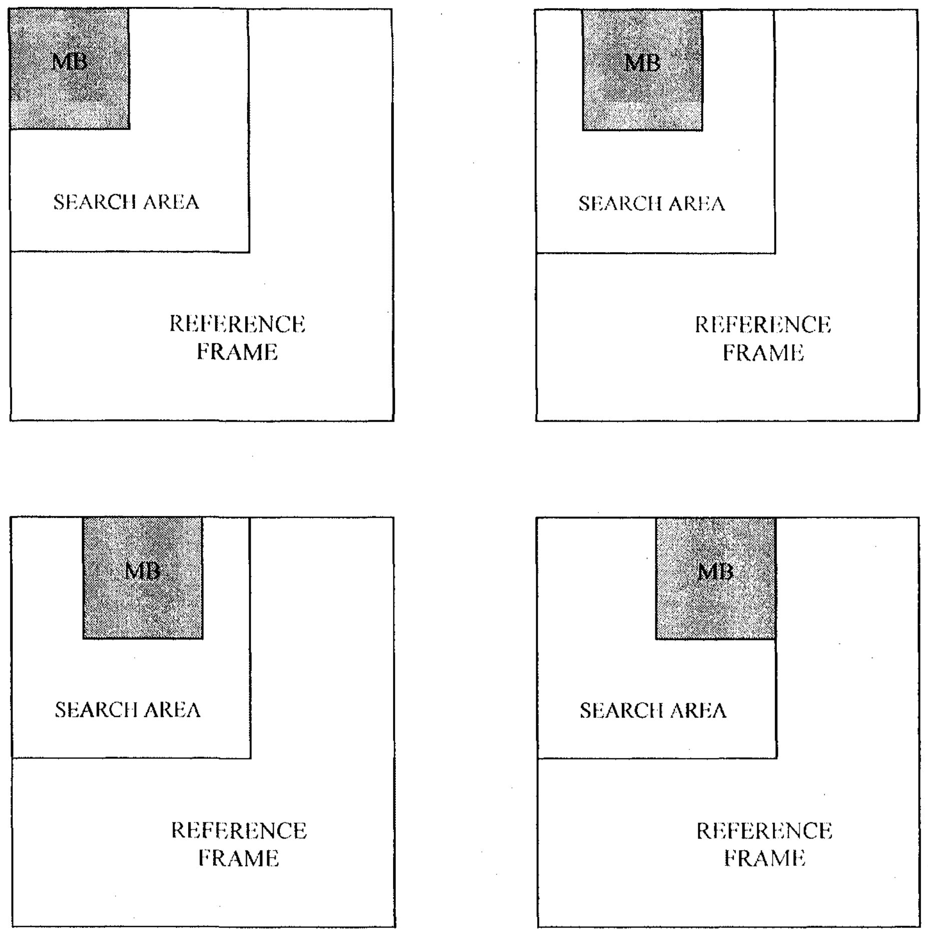

Fig. 3.2 Sliding motion of the reference macroblock within the search area

The hardware for this is as shown in the Fig. 3.3. The FIFOs and D Flip Flops are connected in such a way that at every clock cycle a new set of 16x16 i.e. 256 values are available. These values are fed into the SAD module at once. Each value in the FIFO and the D Flip Flop are a byte long accounting only for the luminance information of the pixels in the image. The feeding in of this structure takes up most of the critical time, which can be properly synchronized to achieve the desired speed up by using two memories and two such structures. While one is active the other structure can start acquiring new set of macroblock values and a control circuit can be set up easily.

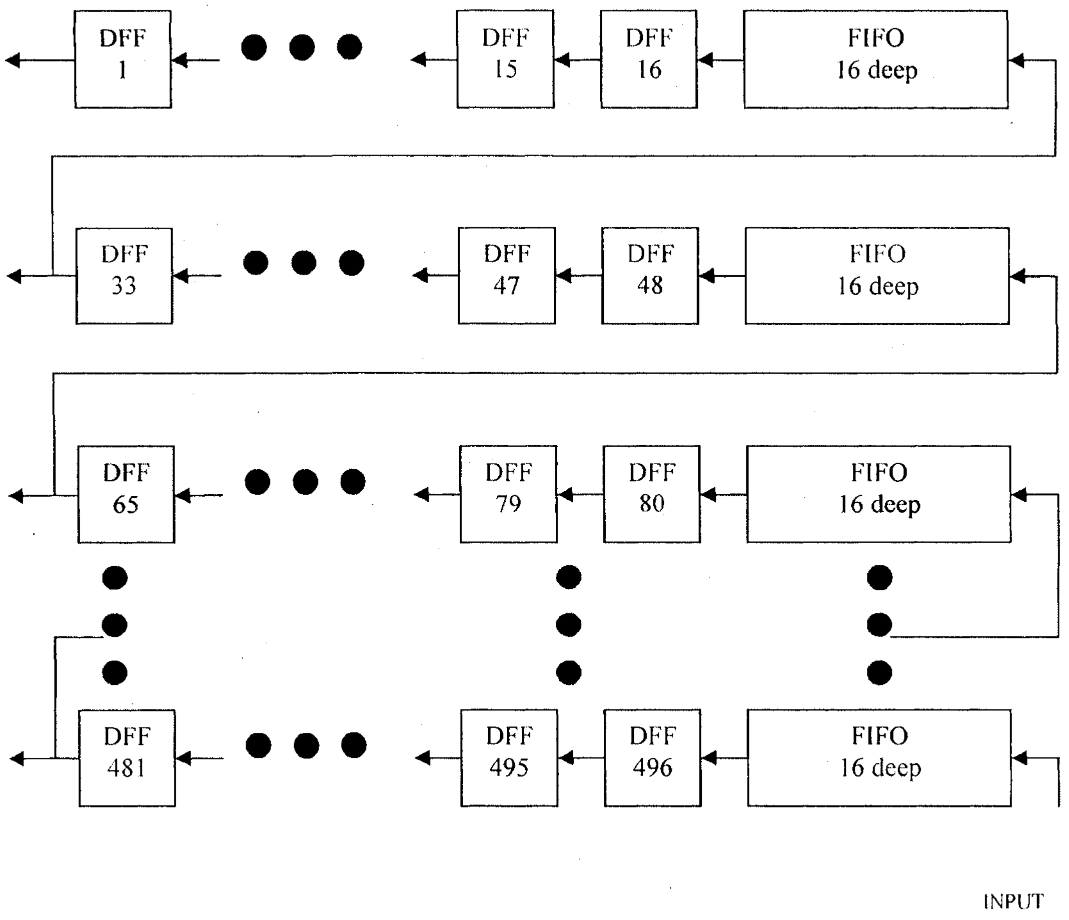

Fig. 3.3 Sliding window architecture

There is an irregularity involved in the architecture shown in Fig. 3.3. There are some values in the search area which are redundant i.e. when the window reaches the rightmost part of the search area. During this time interval, the window has values which exactly do not define any particular macroblock in the search area. The values define nothing but an irregular macroblock which consists of values from rightmost part of the first row and left most part of the second row of the search area. This is controlled by a control signal during which the SAD module does not capture any values. So only those valid values which define the search area properly are captured and used for finding the best match.

This avoids any incorrect results. This structure captures only 289 valid candidate macroblocks for computation. This module assists in selecting all the macroblocks in the search area without having any complex control circuitry or coding techniques. The SAD module takes up almost 38 clock cycles as will be discussed in the next section. That forms the frequency of the reference control sliding window operation which we call as reference clk. This frequency of operation can be increased by achieving pipelining. Pipelining approach has also been adopted here, which will be explained in later sections. By breaking down the combinational logic and inserting registers, the critical path can be comprehensively reduced to 3 times increase in the operating frequency of the Reference Frame controller sliding window module. With minor extra overheads a faster operation can be achieved.

3.2.4 Current Frame Control

This module is a 8-byte Shift Register (SR) which shifts the 256 different luminance values of the current macroblock. As soon as all the 256 values have been clocked in the shift register, all the values are latched into the 256in-256out structure, which then feeds concurrently into the input of SAD block. The synchronization is a bit complex but not as complex as in the case of logarithmic algorithms. In this case, the coefficients values stored in the current macroblock text files are called in the testbench. The text file emulates a RAM which outputs consecutive RAM location values one by one. So, the current macroblock text file writes each luminance value in the Shift register location and all the 256 values are latched at one time. The latched contents are maintained till the whole search area is swept and a final motion vectors are obtained. For this, if another such structure is used with the same memory feeding in SR#2 through a Multiplexer, while SR#1 is maintained for SAD operation, the wait times can be avoided for the SR#2 to fill up. So till the time SR#1 is busy finding the best match, memory can fill in SR#2 which can be ready with the next macroblock to go for second SAD operation. For now, only one structure operation is simulated.

3.2.5 SAD module

This section describes the SAD operation and the possible parallel implementations as proposed and implemented in [4, 5]. Though the theoretical documentation was available, no Verilog codes were available. The Sum of Absolute Differences considers all data units A_i and B_i to be unsigned 8 bits numbers. The general algorithm computing the Sum Absolute Difference of two blocks is depicted in Equation (1). This section first describes the 16x1 SAD operation and then goes further to explain the extension to 16x16 SAD. A direct approach in computation the SAD consists of the following steps:

- Compute $(A_i - B_i)$ for all 16x16 pixels in the two blocks A and B
- Determine which $(A_i - B_i)$ are negative and produce $(B_i - A_i)$ in that case as the absolute value, else produce $(A_i - B_i)$
- Perform the accumulate operation to all 16x16 absolute values.

By determining the smallest of both operands and subtracting it from a constant, it becomes possible to eliminate the absolute operations. This subtraction is a trivial operation, if the constant is chosen correctly. The smaller of two operands is determined by inverting one of the operands, and computing the carry-out which would arise from the addition of both operands. The smaller operand is inverted, which means that its value changes to $(2^8 - 1 - X) = (255 - X)$. Both inverted smallest and the largest values are passed to the adder-tree, which corrects for this constant. The above two steps can be carried out in parallel for 16 pels. The result is 32 8-bit values, on which the following steps are applied. The correction term is added to account for the $(2^n - 1)$'s introduced by the inverting of the smallest value. If the number of pels on which the unit is operating is a power of 2, the correction term is equal to that number, as the sum of the 2^n adds up to one "simple eliminatable bit". If the number of pels the unit operates on is not a power of two, we also have to account for the additional per pel. The resulting rows passed to the adder tree and the correction-term is 33 rows, are reduced to 2 rows by using a Wallace tree carry save adder scheme as proposed in [11, 12, and 13]. In this final step, a full summation of the two remaining rows is performed. The total sum of all constants, which has to be discarded, is the carry out of this addition.

To summarize, the first step is performed by computing [A' + B], where A' stands for inverted A. In case no carry was generated, this means that B is not greater than A and thus B should inverted. Otherwise, A should be inverted. Next to passing the operands to an adder tree, an additional correction term must be added to counter the effects of using inverted values. The adder tree reduces the adder terms two terms which are then passed to an adder. For precise mathematical details of the approach, we refer to [3, 4].

In the previous section, the significance of motion estimation in video coding is mentioned. An important metric used in motion estimation is the sum of absolute differences (SAD). The absolute difference operation can be implemented in several ways: serial, per column in parallel, per row in parallel, and fully parallel. The implementation described in [5] focuses on the SAD16 operation that performs the SAD on one row of a macroblock (16x1). All the input values are 8-bit unsigned binary numbers. By iteration or parallel execution of the SAD16 operation, the complete SAD operation for the 16x16 macroblock can be performed. First, the steps necessary to perform the 16x1 SAD operation in more detail:

- Determine the smaller of the two operands: As suggested in [3, 4], it is only necessary to determine whether (A' + B) produces a carry or not.
- Invert the smallest operand: If no carry was produced then B must be inverted; otherwise, A must be inverted. This is done by utilizing an EXOR operation.
- Pass both operands to an adder tree: After inverting either A or B, the operands must be passed to an adder tree. Thus, the values (A', B) or (A, B') are passed further.
- Add a correction term to the adder tree: Also an additional correction term must be added to the adder tree which is 16 in this case i.e. adding 1 to each of the 16 blocks.
- Reduce the 33 addition terms to 2: All 33 addition terms must be reduced to 2 terms before the final addition can be applied. This can be done using an 8-stage carry save adder tree using 243 carry save adders.
- Add the remaining two terms using an adder: The final two addition terms are added using a 8-bit carry lookahead adder for the most significant bits. The result is a 13-bit unsigned binary number. However, as stated in [4, 5], the most significant bit of this result can be disregarded resulting in a final 12-bit unsigned binary number.

In Fig. 3.4, the first three steps are depicted. The determination whether the addition (A'+ B) generates a carry is performed without actually calculating the addition. Instead, this is achieved by only utilizing certain parts within a carry-lookahead adder that calculate the carry. The resulting carry and inverted carry are fed to two EXORs that will invert the correct term.

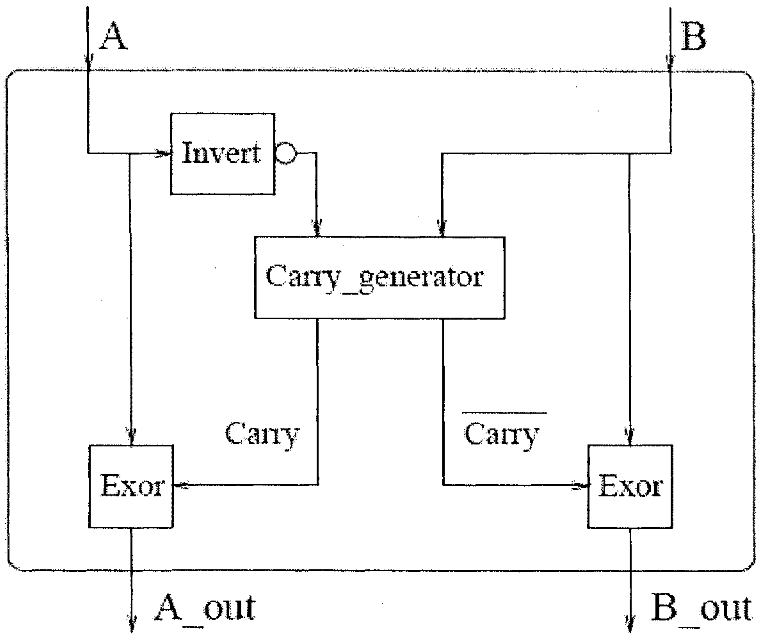

Fig. 3.4 Architecture to find the lower among A and B

The inversion of either As or Bs for all 16 absolute operations can be carried out in parallel and can be fed to an adder tree at the same time [4]. Fig. 3.5 depicts the complete SAD16 operation that has been implemented in [5]. Next to the parallel execution of the first three steps, the figure also depicts the addition of a correction term of 16, the 33 to 2 reduction tree, and the final 2 to 1 reduction. The implementation is synchronous and fully pipeline-able.

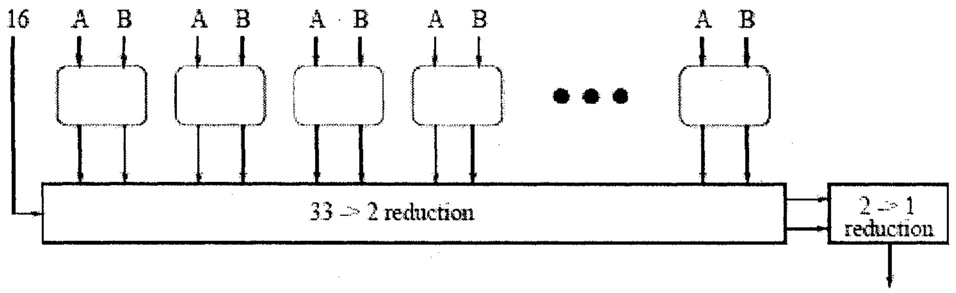

Fig. 3.5 16x1 SAD architecture

The 16 x 16 SAD operations shown in Fig. 3.6, is the implementation carried out in Verilog for this project. The results have been compared with the implementation of [5]. As in [5], for parallel operation of 16x16 SAD, there is only one additional 32 to 2 reduction tree (see Fig. 3.6) when compared to the SAD 16 x 1 unit depicted in Fig. 3.5. This reduction tree is of similar complexity as the 33 to 2 one. For the SAD module of [16] the output is obtained in 27 clock cycles. The first 33 to 2 module requires 8 clock cycles, second 32 to 2 takes another 8 clock cycles and final 2 to 1 reduction tree takes another 9 clock cycles.

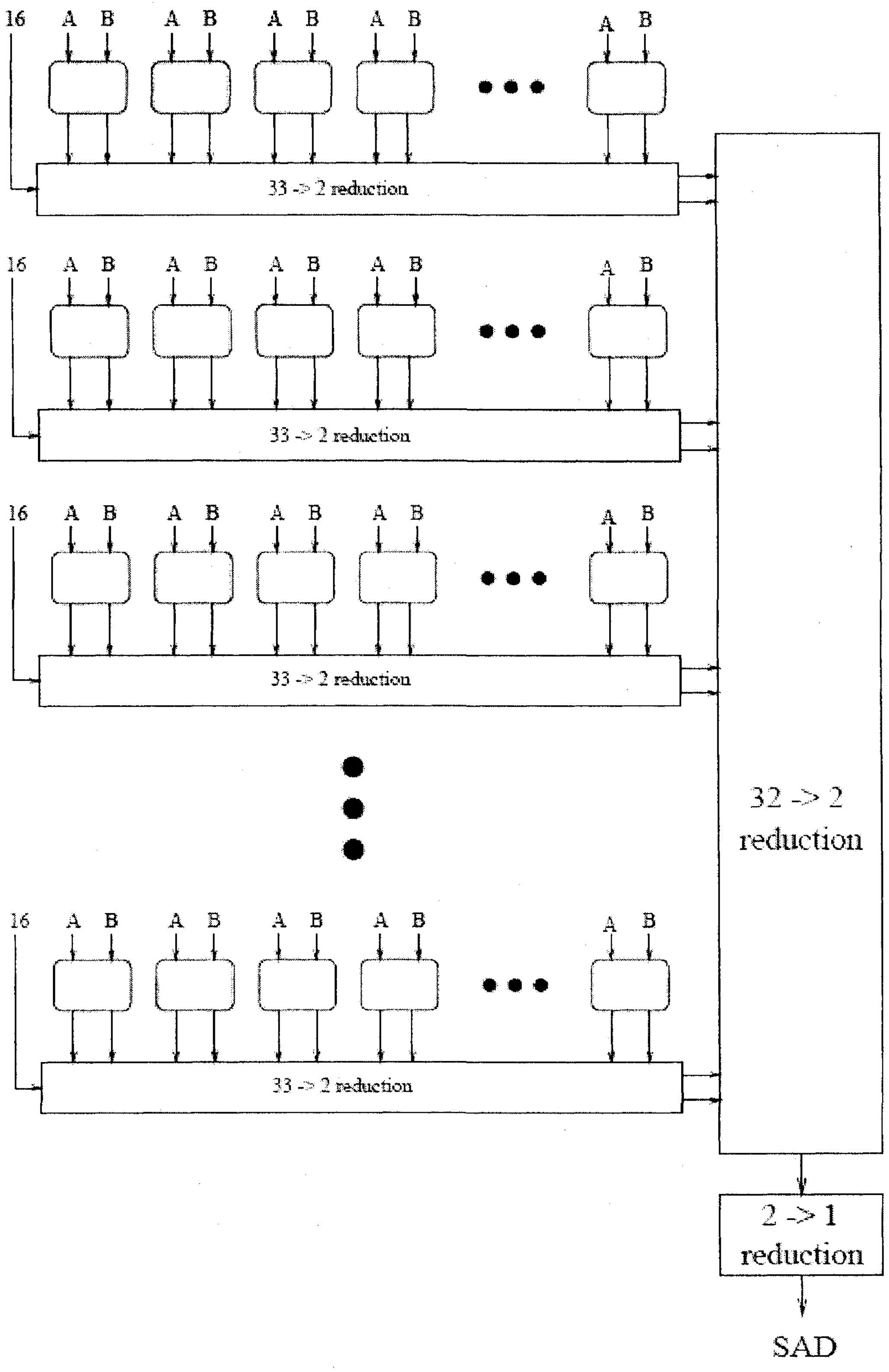

Fig. 3.6 16x16 SAD architecture

The 16x16 SAD operation implemented here forms the critical path of the whole design which controls the reference frames feeding pels to the SAD module. To extend above module and get the motion vectors as final output, a comparator module, a motion vector decoder module and some safety margin is considered. So the final output in this case is gotten after 38 clock cycles. The reference control sliding window circuitry is clocked at f/38 cycles for proper operation. But with the pipelining approach this very frequency is increased by approximately 3 times. So with the pipelining the combination logic is broken down into modules operating at faster frequency. So the final operation frequency of operation at which the Reference sliding window controller operates is f/14.

3.2.6 State machine

The State machine controls the address provided to the BRAMs and the data input to the motion estimation module. It is a fairly simple state machine which utilizes the one-hot state encoding approach. The following figure shows the state machine which controls the current frame control.

There are two state machines running concurrently. One controls the reference frame and other controls the current frame. The states are as shown,

State 1 = "START": This state initializes the state machine

State 2 = ADDR_INIT: The addresses of BRAMs are initialized to all 0s

State 3 = EN_RAM: The BRAMs are enabled and data is read

State 4 = STAY: In this state, internally another straightforward state machine (explained in section explaining 'reference frame control') is activated which controls the flow of valid MBs with help of 'SAD control' signal

State 5 = NEW_ADDR: In this state, after SAD operation over whole reference search area is completed, a new start address is fed and operation begins from EN_RAM state

Same state machine is used for current frame except the BRAM with current MBs will be enabled for 256 addresses only and BRAM for reference frame search MBs for 1008 addresses to compute for all valid macroblocks.

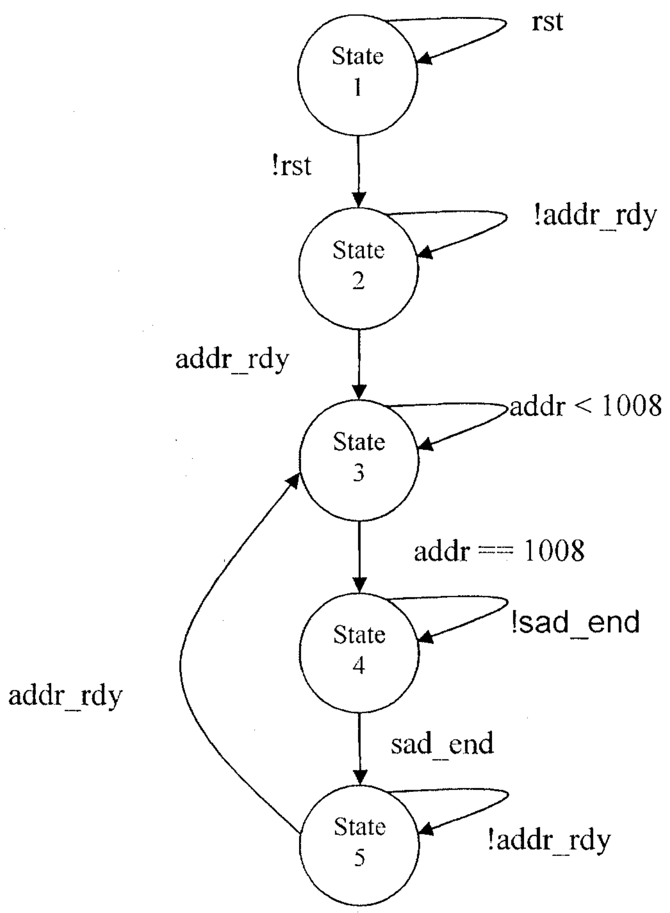

Fig. 3.7 State Machine

3.2.7 Pipelining approach to increase the frequency of operation

The pipelining approach is explained as follows with an example.

- Consider a combinational logic between two registers as shown in Fig. 3.8 below.

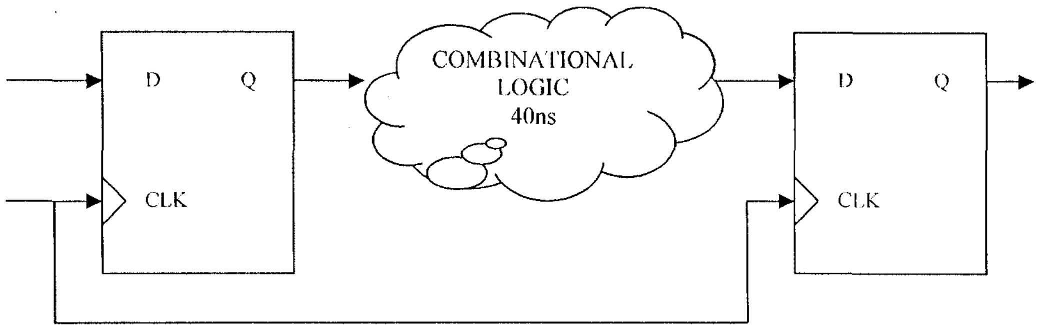

Fig. 3.8 Logic with combinational logic delay

The frequency of operation will depend upon the combination logic path delay, setup time and the clock to output delay of the flip-flops. Let us just consider the combinational logic path delay for the time being. If the delay is 40ns, then our clock frequency becomes 25MHz. Now we see the pipelining approach.

- As shown in fig. 3.9, the combinational logic can be broken down into blocks with smaller delays.

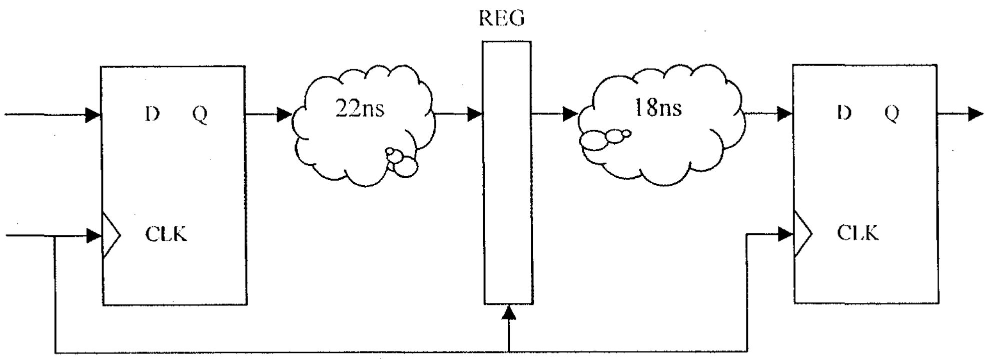

Fig. 3.9 Logic with reduced combinational logic delay

Now, we can insert an intermediate register after each smaller delay block and increase the frequency of operation. With pipelining the above 40ns combinational logic delay is divided into two combinational logic blocks having delay as 22ns and 18ns as shown in Fig. 3.9. Consider the maximum of the two and so 45 MHz becomes our maximum operating frequency. Thus, the frequency increases 1.8 times, which gives a considerable speedup in the whole operation.

CHAPTER 4

RESULTS

The architecture uses a sliding window controller which sweeps the reference image to find the best macroblock match. The valid candidate blocks used to compare with the current macroblock are 289. A control signal controls which candidate blocks are valid to be computed and for which ones the SAD value should be registered. For test purposes an image 80x32 pixels is used. But the module is compatible to any size of image as the architecture is generic provided the macroblocks and search area block are segregated and stored in the memory. The reason for selecting an 80x32 test image was due to ease in debugging and it required lesser simulator memory. The outputs involved lot of values at each instant of time and the generic nature of the module makes it compatible for any size of image. The time taken to output may vary as the image grows larger in size. This module is specifically best utilized for smaller pixel size of images. For example, Fig. 4.1a shows a 16x16 macroblock and the Fig. 4.1b shows the 32x32 search area where the current macroblock will be searched (Images enlarged from normal).

(a) (b)

Fig. 4.1 (a) and (b) Illustrations of current macroblock and reference search area respectively

Fig. 4.2 shows the simulation results for the non-pipelined version of the operation. If the main clock is 50 MHz, our reference frame circuit derived clock will be divide-by-38, which is 1.3 MHz. According to the simulation, the total time taken to compute the best match for one macroblock is 778.64 us. The final best match motion vectors for a full search of one current macroblock is available at the rate of 1.3 KHz. So, the computation of best match of one macroblock takes almost 1000 reference clock cycles. This time considerably reduces for the pipelined version. FPGAs which can operate at very high frequencies like 130MHz, 200MHz etc also assist in speeding-up the operation.

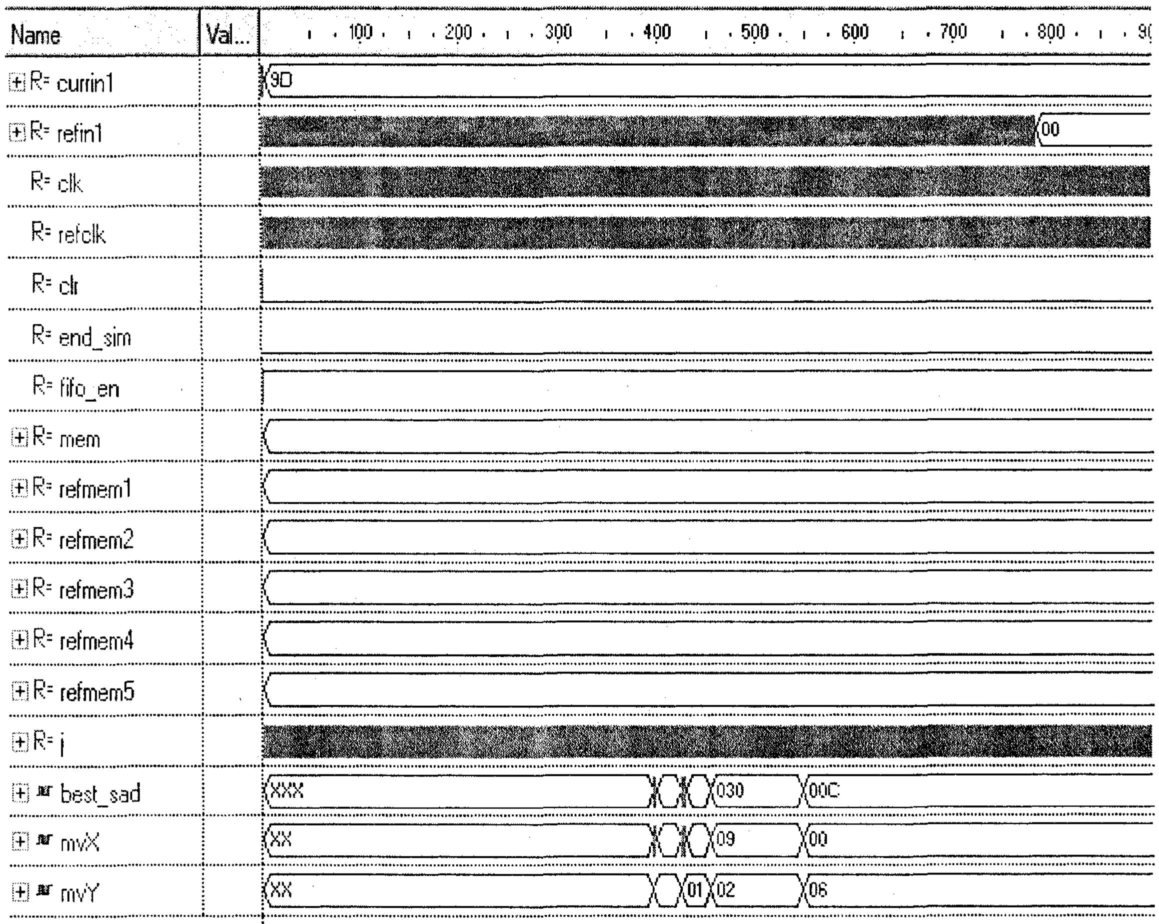

Fig. 4.2 Simulation results for motion estimation without pipelining

As shown in Fig. 4.3, for the same test image, we use the pipelined approach. Here for a main clock of 50 MHz, the derived clock is divide-by-14 i.e. reference clock is 3.5 MHz. Thus the output is available after 284.48 us i.e. 3.5 KHz. Thus, for the 1000 reference clock cycles/MB, the amount of time taken to get the final motion vectors for one macroblock reduces considerably. The time reduces by 491.52 us per current macroblock best match computation.

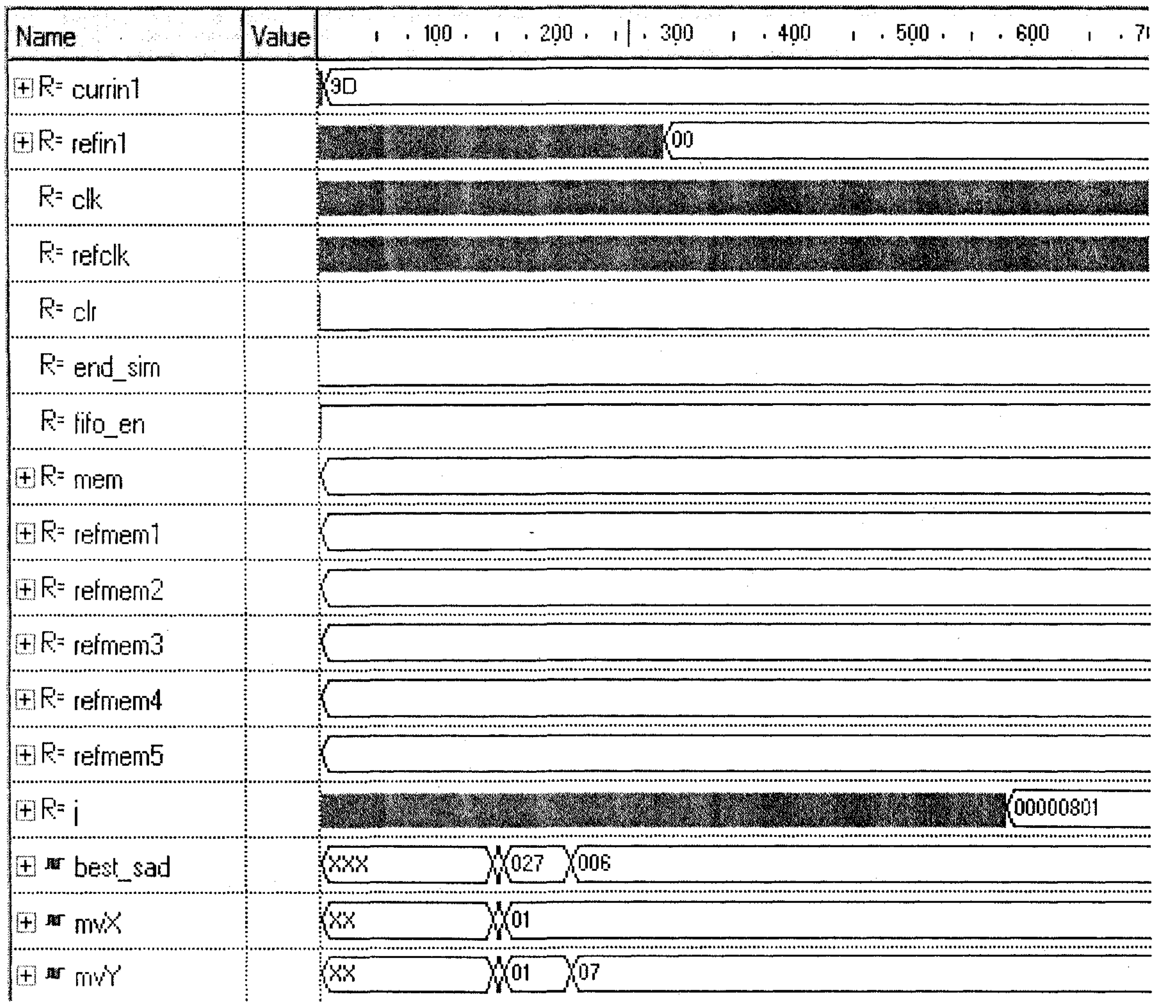

Fig. 4.3 Simulation results for motion estimation with pipelining

Thus from above results, we come up with the following equation, which tells us how much frames per second is supported by this architecture –

$$X = \frac{\text{fmax}}{(\text{ref clock cycles/MB}) \times N \times \text{Total MBs}} \qquad \text{(Eq 4.1)}$$

Where,

X = frames per second which can be supported for the given fmax

fmax = maximum clock frequency

N = fmax divided by N gives the reference clock frequency at which reference sliding window protocol operates.

Based upon the above equation some of the projected results are tabulated as follows. These results target Common Intermediate Format (CIF). CIF is a format used to standardize the horizontal and vertical resolutions in pixels of YCbCr sequences in video signals, commonly used in video teleconferencing systems. It was first proposed in the H.261 standard. CIF was designed to be easy to convert to PAL or NTSC standards. QCIF means "Quarter CIF". To have one fourth of the area as "quarter" implies, height and width of the frame are halved. Terms also used are SQCIF (Sub Quarter CIF), 4CIF (4× CIF) and 16CIF (16× CIF). SIF (Source Input Format) is practically identical to CIF, but taken from MPEG-1 rather than ITU standards. SIF based systems is 352 x 240. Projected results for some of them are tabulated in Table 4.1.

Table 4.1 Results describing the frames per second supported by the architecture

Format	Video Resolution	No. of MBs	FPGA family/Clock (fmax)	Supports fps	
				Non-Pipelined	Pipelined
SQCIF	128x96	48	Spartan-3/50MHz	25 fps	60 fps
			Spartan-3 DSP/130MHz	60 fps	60 fps
			Virtex-4/225MHz	60 fps	60 fps
QCIF	176x144	99	Spartan-3/50MHz	15 fps	30 fps
			Spartan-3 DSP/130MHz	30 fps	60 fps
			Virtex-4/225MHz	60 fps	60 fps
CIF	352x288	396	Spartan-3/50MHz	3 fps	9 fps
			Spartan-3 DSP/130MHz	8 fps	25 fps
			Virtex-4/225MHz	15 fps	30 fps
SIF	352x240	330	Spartan-3/50MHz	4 fps	10 fps
			Spartan-3 DSP/130MHz	10 fps	25 fps
			Virtex-4/225MHz	15 fps	50 fps

CHAPTER 5

CONCLUSION AND FUTURE RECOMMENDATIONS

Motion Estimation in MPEG video is a temporal prediction technique. The basic principle of motion estimation is that in most cases, consecutive video frames will be similar except for changes induced by objects moving within the frames. Motion Estimation performs a comprehensive 2-dimensional spatial search for each luminance macroblock. MPEG does not define how this search should be performed. This is a detail that the system designer can choose to implement in one of many possible ways. The motion estimation hardware accelerator based on a Full Search Block Matching Algorithm is implemented in Verilog HDL. State Machines for reference SAD control and reference BRAM control can be merged together for simplicity in coding. In the case of described implementation, the codes were done in a hierarchical order due to which the state machines are split and are presented in that fashion. The reconfigurable nature of FPGAs will make it easier to implement and make the core and re-test it. This core if tested with a general purpose RISC processor like the Xilinx's Microblaze will make it easier to segregate the macroblocks and aid for the achieving the projected timelines. The core can be instantiated in the Microblaze and pixels can be fetched using Fast Simplex Link interface at a faster rate. The generic nature of the module defines a 32x32 search area for each current macroblock and hence any size of image can be used for testing purpose.

BIBLIOGRAPHY

1. V. Bhaskaran and K. Konstantinides. Image and Video Compression Standards: Algorithms and Architectures. Kluwer Acad. Publish., 2nd edition, June 1997.
2. Tiago Dias, Nuno Carlos André Sebastião, Nuno Roma, Paulo Flores, Leonel Sousa, Programmable IP core for motion estimation: comparison of FPGA and ASIC based implementations, In IV Jornadas sobre Sistemas Reconfiguráveis - REC2008, pages 109 116, February 2008.
3. S. Vassiliadis, E.A. Hakkennes, J.S.S.M. Wong, G.G. Pechanek, "The Sum-Absolute-Difference Motion Estimation Accelerator," euromicro, p. 20559, 24 th. EUROMICRO Conference Volume 2 (EUROMICRO'98), 1998
4. S. Vassiliadis, E. Hakkennes, S. Wong, and G. Pechanek. The Sum-Absolute-Difference Motion Estimation Accelerator. In Proceedings of the 24th Euromicro Conference, 2000.
5. Wong, S.; Vassiliadis, S.; Cotofana, S., "A sum of absolute differences implementation in FPGA hardware," Euromicro Conference, 2002. Proceedings. 28th , vol., no., pp. 183-188, 2002
6. J. Hauser and J.Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In Proceedings of the IEEE Symposium of Field-Programmable Custom Computing Machines, pages 24–33, April 1997.

7. D. Cronquist, P. Franklin, C. Fisher, M. Figueroa, and C. Ebeling. Architecture Design of Reconfigurable Pipelined Datapaths. In Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI, pages 23–40, March 1999.

8. http://www.irfanview.net/

9. http://download.xilinx.com/direct/ise9_tutorials/ise9tut.pdf

10. http://www.xilinx.com/support/documentation/application_notes/xapp463.pdf

11. C. Wallace. A suggestion for parallel multipliers. IEEE Trans. Electron. Comput., EC 13:14–17, 1964.

12. L. Dadda. Some schemes for parallel multipliers. Alta Frequenza, 34:349–356, May 1965.

13. Vassiliadis, S.; Hoekstra, J.; Chiu, H.-T., "Array multiplication scheme using (p, 2) counters and pre-addition ," Electronics Letters , vol.31, no.8, pp.619-620, 13 Apr 1995

14. "Motion estimation techniques in video processing", Electronic Engineering Times India, August 2007

15. Joaquin Olivares; Ignacio Benavides; Javier Hormigo; Julio Villalba; Emilio Zapata, "Fast Full-Search Block Matching Algorithm Motion Estimation Alternatives in FPGA," Field Programmable Logic and Applications, 2006.

16. R. Srinivasan and K.R. Rao, "Predictive Coding based on Efficient Motion Estimation," IEEE Trans. Commun., vol. COM-33, no. 8, 1985, pp. 888–896.

17. S. Kappagantula and K.R. Rao, "Motion Compensated Interframe Image Prediction," IEEE Trans. Commun., vol. COM-33, no. 9, 1985, pp. 1011–1015.

18. L.G. Chen, W.T. Chen, Y.S. Jehng, and T.D. Chiueh, "An Efficient Parallel Motion Estimation Algorithm for Digital Image Processing," IEEE Trans. Circuits System Video Technology., vol. 1, no. 4, 1991, pp. 378–385.

19. M.J. Chen, L.G. Chen, and T.D. Chiueh, "One-dimensional full Search Motion Estimation Algorithm for Video Coding," IEEE Trans. Circuits Syst. Video Technol., vol. 4, no. 5, 1994, pp. 504–509.

20. M. Brunig and W. Niehsen, "Fast Full-search Block Matching," IEEE Trans. Circuits Syst. Video Technol., vol. 11, no. 2, 2001, pp. 241–247.

21. R. Li, B. Zeng, and M.L. Liou, "A New Three-step Search Algorithm for Block Motion Estimation," IEEE Trans. Circuits Syst. Video Technology, vol. 4, no. 4, pp. 438/442, Aug. 1994.

22. http://www.digitalcreationlabs.com/support.htm

23. http://www.xilinx.com/support/documentation/application_notes/xapp610.pdf

24. http://www.xilinx.com/support/documentation/application_notes/xapp615.pdf

25. http://www.xilinx.com/support/documentation/application_notes/xapp616.pdf

VITA

Graduate College
University of Nevada Las Vegas

Nachiket Jugade

Address:
1555 E Rochelle Ave Apt 147
Las Vegas, NV 89119

Degree:

- Bachelor of Engineering, Electronics and Telecom Engineering, 2006
  University of Pune, India

Professional Experience:

- Teaching Assistant, University of Nevada Las Vegas, Aug 2006-May 2008
- Engineering Intern, Aldec Inc. Las Vegas, Jun 2007-Aug 2007

Special Honors and Awards:

- Magna Cum Laude – Member of National Scholars' Honor Society
- Bally Technologies' Graduate Scholarship Recipient, 2007-2008
- Member of Tau Beta Pi – The Engineering Honor Society
- Vice President, Indian Student Association (ISA) at UNLV, 2007-2008

Thesis Title:
Implementation of BMA Based Motion Estimation Hardware Accelerator in HDL

Thesis Examination Committee:
Chairperson, Dr. Henry Selvaraj, Ph.D.
Committee Member, Dr. Emma Regentova, Ph.D.
Committee Member, Dr. Muthukumar Venkatesan, Ph.D.
Graduate College Representative, Dr. Laxmi Gewali, Ph.D.