H.264 MOTION ESTIMATOR DESIGN

by

SİNAN YALÇIN

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of
Master of Science

Sabancı University
Spring 2005

H.264 MOTION ESTIMATOR DESIGN

APPROVED BY:

Assist. Prof. İlker Hamzaoğlu …………………………
(Thesis Supervisor)

Assist. Prof. Ayhan Bozkurt …………………………

Assist. Prof. Hasan Ateş …………………………

DATE OF APPROVAL: …………………………

ABSTRACT

Recently, a new international standard for video compression named H.264 / MPEG-4 Part 10 is developed. This new standard offers significantly better video compression efficiency than previous international standards. The variable block size motion estimation is the most compute-intensive part of an H.264 video encoder. The full search method is impractical for real-time implementations since it requires a high computational complexity. Therefore, many fast motion estimation algorithms have been developed for real-time implementations. In this thesis, we used an SAD reuse based hierarchical motion estimation algorithm for real-time H.264 / MPEG-4 Part 10 video coding. This algorithm uses the Lagrangian cost parameter (SAD+λR) for selecting the best motion vector. We designed a high performance and low cost hardware architecture for real-time implementation of this algorithm. We have considered several alternative designs and decided on this architecture based on a cost/performance analysis. This architecture uses a novel data flow resulting in a low cost and high performance hardware. This hardware is designed to be used as part of a complete H.264 video coding system for portable applications. The proposed architecture is implemented in Verilog HDL. The Verilog RTL code is verified to work at 63 MHz in a Xilinx Virtex II FPGA. The FPGA implementation can process 25 VGA frames (640x480) or 76 CIF frames (352x288) per second.

ÖZET

Son zamanlarda, video sıkıştırma için H.264 / MPEG-4 Bölüm 10 isimli yeni bir uluslararası standart geliştirildi. Bu yeni standart, önceki uluslarası standartlardan önemli derecede daha iyi bir video kodlama verimliliği sağlamaktadır. Değişken blok boyutu hareket tahmini, H.264 video kodlayıcının en işlemsel yoğunluktaki bölümüdür. Tam arama metodu gerçek-zamanlı gerçekleştirmelerde yüksek işlemsel karmaşıklık gerektirdiğinden dolayı, pratik değildir. Bundan dolayı, gerçek-zamanlı gerçekleştirmeleri sağlayabilmek için birçok hızlı arama algoritmaları geliştirilmiştir. Bu tezde, gerçek-zamanlı H.264 / AVC bölüm 10 video kodlama için SAD tekrar kullanma tabanlı hiyerarşik hareket tahmini algoritması kullandık. Bu algoritma, en iyi hareket vektörünü seçmek için Lagrangian değer parametresini (SAD+λR) kullanıyor. Bu algoritmanın gerçek-zamanlı gerçekleştirmesi için bir yüksek performanslı ve düşük maliyetli donanım mimarisi tasarladık. Çeşitli alternatif tasarımları göz önünde bulundurduk ve maliyet/performans analizine göre bu tasarımda karar kıldık. Bu mimari düşük maliyetli ve yüksek performanslı bir donanım ile sonuçlanan yeni bir veri akışı kullanmaktadır. Bu donanım, portatif uygulamalar için komple bir H.264 video kodlama sisteminin bir parçası olarak kullanılmak üzere tasarlandı. Arz edilen mimari Verilog HDL'de gerçekleştirildi. Verilog RTL kodu bir Xilinx Virtex II FPGA'de 63 MHz'de çalışmak üzere doğrulandı. FPGA gerçekleştirmesi, saniyede 25 VGA karesi veya 76 CIF karesi işleyebilmektedir.

To my brothers Mustafa and Özgür…

To my beloved family…

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to Assistant Prof. Ilker Hamzaoglu for his guidance, suggestions and help during this project. I am very glad and grateful to study with such a great supervisor. It was really a pleasure and honor for me.

I would like to thank Dr. Hasan Ates who assisted us with the design of the motion estimation algorithm.

I also want to thank Assistant Prof. Ayhan Bozkurt who participated in my thesis jury.

I would like to thank my partners in H.264 research project, Ozgur Tasdizen and Esra Sahin, for valuable discussions and feedback throughout the project.

I am of course indebted to my whole family, especially my brother Mustafa Yalcin, for their love and support, without them I would not be in such a position.

Lastly, I would like to thank Sabanci University for supporting our project and establishing and maintaining such a perfect university.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ITU	International Telecommunication Union
ISO	International Organization of Standardization
MB	Macroblock
SAD	Sum of Absolute Difference
HDL	Hardware Description Language
RTL	Register Transfer Level
FPGA	Field Programmable Logic Array
YUV	Luminance and Chrominance Color Space
fps	frames per second
Gops	Giga operations per second
MV	Motion Vector
SAD+λR	Lagrangian Cost Parameter
λR	Bit-rate Cost Parameter
JM	Joint Model
CAVLC	Context Adaptive Variable Length Coding
QP	Quantization Parameter
PE	Processing Element
ASIC	Application Specific Integrated Circuit
AVC	Advanced Video Coding
JVT	Joint Video Team

CHAPTER 1

INTRODUCTION

Video processing is used in many applications. In most of these applications, the transmission mediums are not capable of transmitting uncompressed digital video in real-time and the capacity of storage mediums are not enough for storing uncompressed digital video. Video compression makes it possible to use digital video in transmission and storage that would not support uncompressed video [1]. Therefore, video compression systems are used in many commercial products, from consumer electronic devices such as digital camcorders, cellular phones to video teleconferencing systems. These applications make the video compression hardware devices an inevitable part of many commercial products.

To improve the performance of the existing applications and to enable the applicability of video compression to new real-time applications, recently, a new international standard for video compression is developed [2, 3, 4]. This new standard, offering 50% better video compression efficiency than previous video compression standards, is developed with the collaboration of ITU and ISO standardization organizations. Hence it is called with two different names, H.264 and MPEG-4 Part 10.

The top-level block diagram of an H.264 encoder is shown in Figure 1. Motion estimation generates a predicted macroblock (MB) for the current MB in the current frame based on temporal redundancy between the current and reference frames. Intra prediction generates a predicted MB for the current MB in the current frame based on spatial redundancy in the current frame. Mode decision compares the required amount of bits to encode a MB and the quality of the decoded MB for both of these modes (inter and intra), and chooses the mode with better quality and bit-rate performance. In either case, the predicted MB is subtracted from the current MB to generate the residual MB. The residual MB is transformed, quantized, re-ordered in a zig-zag scan order and entropy encoded. This process is repeated for each MB in the current frame. The entropy encoded coefficients together with header information, such as MB prediction mode, motion vectors and quantization step size, form the compressed bit stream. The compressed bit stream is passed to network abstraction layer (NAL) for storage or transmission.

Since a decoder never gets original images, but rather works on the decoded frames, the encoded frame is also decoded and reconstructed in the encoder to ensure that both encoder and decoder use identical reference frames for intra and inter prediction. The reconstructed residual data are generated by inverse quantization and inverse transform. The reconstructed residual data are added to the predicted pixels to create the reconstructed frame. A deblocking filter is applied to reduce the effects of blocking artifacts in the reconstructed frame.

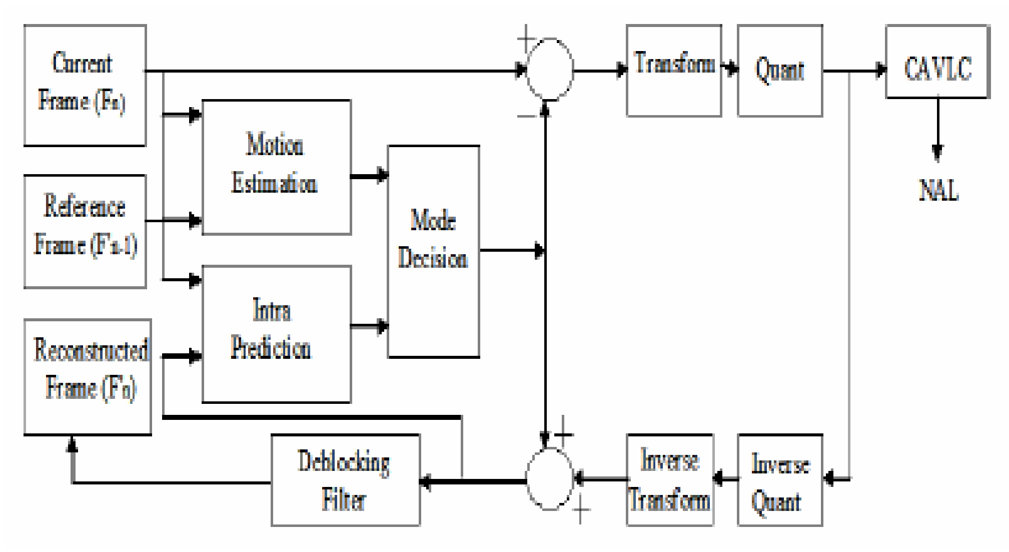


Figure 1. H.264 encoder block diagram

The video compression efficiency achieved in H.264 standard is not a result of any single feature but rather a combination of a number of encoding tools. As it is shown in Figure 1, one of these tools is the variable block size motion estimation used in the baseline profile of H.264 standard. Motion estimation is the most computationally demanding part of the encoders implementing the previous video compression standards. Variable block size motion estimation achieves better coding results than the fixed block size motion estimation used in the previous video compression standards. However, the amount of computation required by variable block size motion estimation is even more than the amount required by fixed block size motion estimation. Therefore, this coding gain comes with an increase in encoding complexity which makes it an exciting challenge to have a real-time implementation of motion estimation for H.264 video coding.

In this thesis, we present a high performance and low cost hardware architecture for real-time implementation of an SAD reuse based hierarchical motion estimation algorithm for H.264 / MPEG4 Part 10 video coding. This hardware is designed to be used as part of a complete H.264 video coding system for portable applications. The proposed architecture is implemented in Verilog HDL. The Verilog RTL code is verified to work at 63 MHz in a Xilinx Virtex II FPGA. The FPGA implementation can process 25 VGA frames (640x480) or 76 CIF frames (352x288) per second.

The rest of the thesis is organized as follows. Chapter 2 explains motion estimation and presents several motion estimation algorithms. Chapter 3 explains our SAD reuse based hierarchical motion estimation algorithm. Chapter 4 describes the proposed hardware architecture in detail. The hardware implementation results are given in Chapter 5. Finally, Chapter 6 presents the conclusions and future work.

CHAPTER 2

MOTION ESTIMATION

As illustrated in Figure 2, motion estimation is the process of searching a search window in a reference frame to determine the best match for a block in a current frame based on a search criterion such as minimum sum of absolute difference (SAD) [1]. The location of a block in a frame is given using the (x,y) coordinates of top-left corner of the block. The search window in the reference frame is the [-p,p] size region around the location of the current block in the current frame. The SAD value for a current block in the current frame and a candidate block in the reference frame is calculated by accumulating the absolute differences of corresponding pixels in the two blocks as shown in the following formula:

$$\text{SAD}_{B_{mxn}}(\mathbf{d}) = \sum_{x=1,y=1}^{m,n} | c(x,y) - r(x+d_x, y+d_y) | \tag{2.1}$$

where B_{mxn} is a block of size mxn, $\mathbf{d}$=*(dx, dy)* is the motion vector (MV), *c* and *r* are current and reference frames respectively. Since a motion vector expresses the relative motion of the current block in the reference frame, motion vectors are specified in relative coordinates. If the location of the best matching block in the reference frame is (x+u, y+v), then the motion vector is expressed as (u,v).

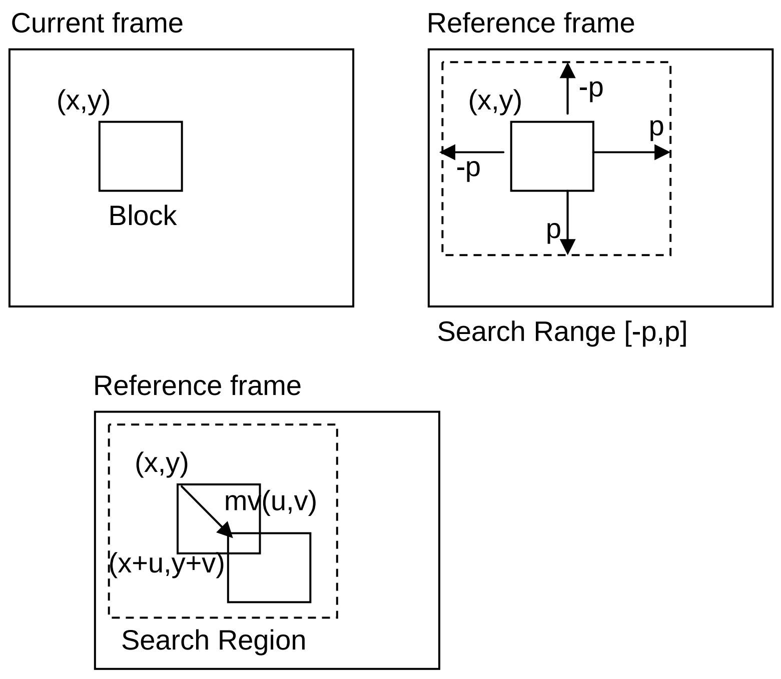


Figure 2. Motion estimation process

The motion estimation and motion compensation flows in a video encoder and a video decoder are shown in Figure 3 [1]. Motion estimation is performed on the luminance (Y) component of a YUV image and the resulting motion vectors are also used for the chrominance (U and V) components. After the motion vector for a block is determined, the residual block (the difference between the current block and the reference block pointed by the motion vector) is calculated by the motion compensation module. The motion vector and the residual block (I(x,y,t) – I(x-u, y-v, t-1)) are coded in the encoder and transmitted. This process is repeated for all the blocks in the current frame. In the decoder, the motion vector and the residual block are decoded. Then, the reference block in the reference frame pointed by the motion vector (I'(x-u, y-v, t-1)) is determined by the motion compensation module, and it is added to residual block. The resulting reconstructed block is stored in the frame memory and it is used for motion compensation for the next frame. This reconstruction is also done in the encoder in order to ensure that encoder and decoder use identical reference frames for motion compensation.

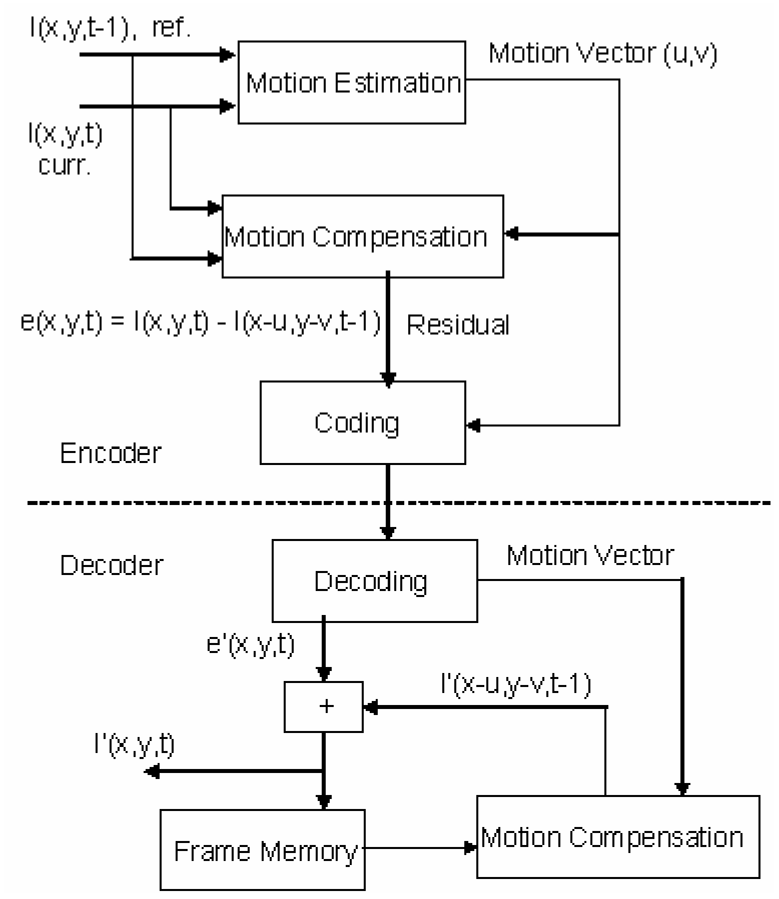


Figure 3. Motion estimation and motion compensation flow

Motion estimation is the most computationally expensive part of the video encoders. Therefore, many fast motion estimation algorithms for real-time implementation have been developed. We will explain several widely used motion estimation algorithms and present a comparison of their computational complexity.

2.1 Full Search Algorithm

Full search algorithm finds the best motion vectors in a search window by searching all of the search locations in that search window. Therefore, it is the most computationally expensive algorithm. Because of this, it is used as a basis for evaluating the performances of the faster motion estimation algorithms. There are $(2p + 1)^2$ search locations in a [-p,p] search window. For a 16x16 MB, the size of the search area is (2p+16)x(2p+16). The search window for p = 4, $(2x4+1)^2$ = 81 search locations, are shown in Figure 4. The size of the search area for p = 4 is 24x24.

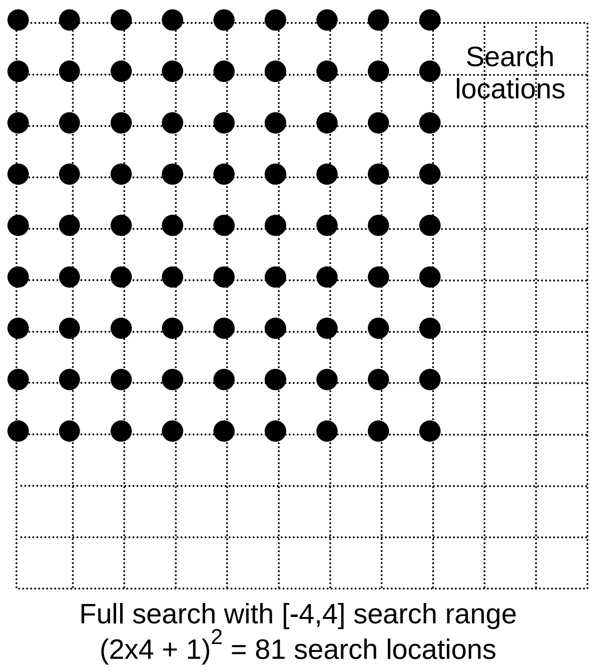

Search locations
Full search with [-4,4] search range
$(2x4 + 1)^2$ = 81 search locations

Figure 4. Full search algorithm

In order to compare the computational complexity of the full search algorithm with the faster algorithms, we will calculate the number of operations per second needed for performing motion estimation for 30 VGA frames per second (fps) using the full search algorithm with p = 16. A VGA frame has 640x480 pixels corresponding to (640/16)x(480/16) = 1200 MBs. In order to process 30 VGA fps, 1200x30 = 36K MBs per second have to be processed. There are $(2p + 1)^2 = (2x16+1)^2 = 1089$ search locations for each MB. At each search location, three operations (subtraction, absolute value and addition) are performed for 16x16=256 pixels, resulting in 256x3=768 operations per search location. Therefore, in total, (36K MBs per second) x (1089 search locations per MB) x (768 operations per search location) = 30 Giga operations per second (Gops) have to be performed.

2.2 Two Dimensional Logarithmic Search Algorithm

Two dimensional (2D) logarithmic search algorithm speeds up the motion estimation process by using less number of search locations than full search algorithm. The algorithm works in several steps [1]. In the first step, for a [-p,p] search window, the center point and the eight major points on the perimeter of the [-p/2,p/2] search area inside the search window are searched. The distance between these eight major points is $d_1 = 2^{k-1}$ where $k = \log_2(p+1)$. In the second step, the search location that produced the best match in the first step is used as the starting point and the search is performed at eight major perimeter points with distance $d_2 = d_1 / 2$. This process is repeated until the k-th step where the eight perimeter search locations are spaced by a distance of one. Therefore, in total, $8k + 1$ search locations (8 perimeter points in each step and one center point in the first step) are searched. The search locations for 2D logarithmic search algorithm with $p = 7$ are shown in Figure 5. In this example, $k = 3$ which means 3-step search is performed with $d_1 = 2^{k-1} = 4$ and $d_2 = d_1 / 2 = 2$ and $d_3 = d_2 / 2 = 1$. It can be noticed that $d_1+d_2+d_3 = p = 7$. The size of the search area for $p = 7$ is 30x30.

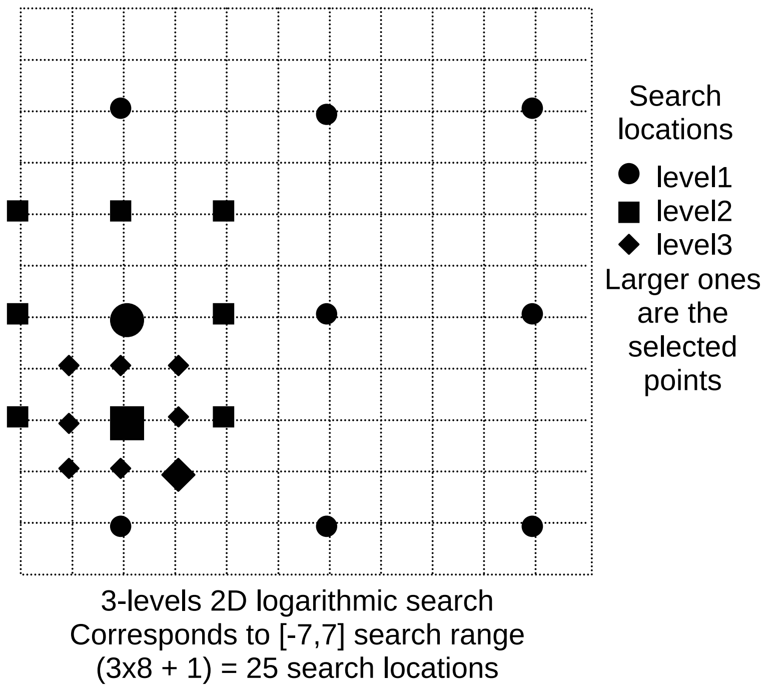

Search locations
● level1
■ level2
◆ level3
Larger ones are the selected points
3-levels 2D logarithmic search
Corresponds to [-7,7] search range
(3x8 + 1) = 25 search locations

Figure 5. Two dimensional logarithmic search algorithm

In order to compare the computational complexity of the 2D logarithmic search algorithm with the other motion estimation algorithms, we will calculate the number of operations per second needed for performing motion estimation for 30 VGA fps using the 2D logarithmic search algorithm with $p = 15$. For the 2D logarithmic search algorithm, only the number of search locations per MB is different than the full search algorithm. For $p = 15$, 4-step search is performed, and there are $8x4 + 1 = 33$ search locations. Therefore, in total, (36K MBs per second) x (33 search locations per MB) x (768 operations per search location) = 910 Mops have to be performed.

2.3 Hierarchical Motion Estimation Algorithm

Hierarchical motion estimation algorithm speeds up the motion estimation process by using less number of search locations and by computing the SAD at a search location using less number of pixels than full search algorithm. The algorithm reduces the number of pixels used at a search location by down-sampling the current MB and the search area and performing the search operation in lower resolution. A 3-level hierarchical motion estimation algorithm is shown in Figure 6 [1].

The algorithm, first, generates the search area and the 8x8 current block in $level_1$ by down-sampling the search area and the current MB in $level_0$ by 2. It, then, generates the search area and the 4x4 current block in $level_2$ by down-sampling the search area and the current block in $level_1$ by 2. If the location of the current MB in $level_0$ is (x,y), the locations of the 8x8 current block in $level_1$ and the 4x4 current block in $level_2$ are (x/2,y/2) and (x/4,y/4) respectively. Then, the algorithm finds the motion vector $mv_2(u_2,v_2)$ by performing full search in $level_2$ for the 4x4 block with a search range p_2. It, then, finds the motion vector $mv_1(u_1,v_1)$ by performing full search in $level_1$ at the location pointed by $2mv_2$, i.e. $(x/2 + 2u_2, y/2 + 2v_2)$, for the 8x8 block with a search range p_1. The algorithm, finally, finds the motion vector $mv_0(u_0,v_0)$ by performing full search in $level_0$ at the location pointed by $2mv_1$, i.e. $(x + 2u_1, y + 2v_1)$, for the 16x16 current MB with a search range p_0. The mv_0 is the motion vector determined by the 3-level hierarchical motion estimation algorithm.

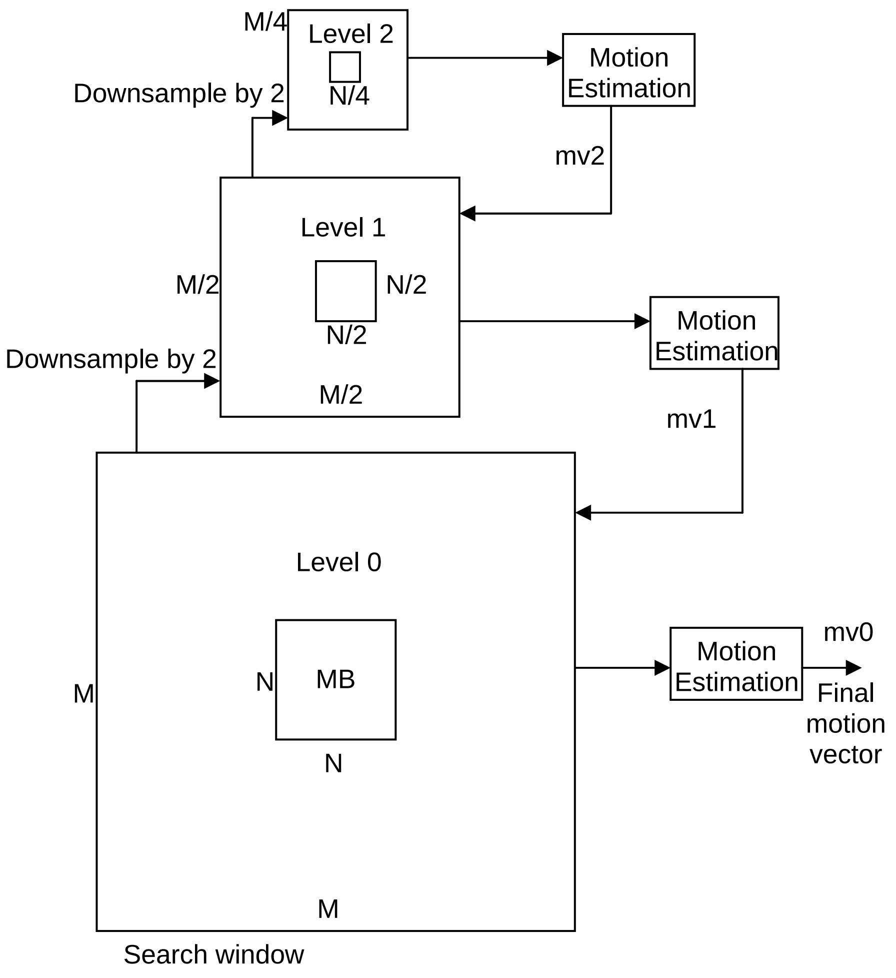
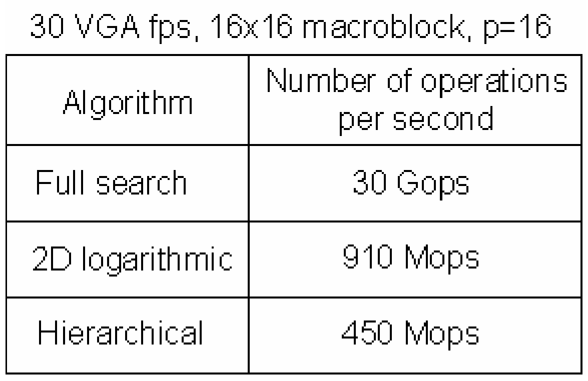


Figure 6. Hierarchical motion estimation algorithm

In order to compare the computational complexity of the hierarchical motion estimation algorithm with the other motion estimation algorithms, we will calculate the number of operations per second needed for performing motion estimation for 30 VGA fps using the 3-level hierarchical motion estimation algorithm with $p_2 = 4$, $p_1 = 1$, $p_0 = 1$. As we said before, in order to process 30 VGA fps, 36K MBs per second have to be processed. In $level_2$, there are $(2x4+1)^2 = 81$ search locations per 4x4 block and 4x4x3=48 operations per search location. Therefore, in $level_2$, (36K blocks per second) x (81 search locations per block) x (48 operations per search location) = 140 Mops have to be performed. In $level_1$, there are $(2x1+1)^2 = 9$ search locations per 8x8 block and 8x8x3=192 operations per search location. Therefore, in $level_1$, (36K blocks per second) x (9 search locations per block) x (192 operations per search location) = 62 Mops have to be performed. In $level_0$, there are 9 search locations per 16x16 MB and 16x16x3=768 operations per search location. Therefore, in $level_0$, (36K MBs per second) x (9 search locations per MB) x (768 operations per search location) = 248 Mops have to be performed. Therefore, in total, 140 + 62 + 248 = 450 Mops have to be performed.

2.4 Comparison of Motion Estimation Algorithms

The computational complexity of the three motion estimation algorithms are summarized in Table 1. The full search method is in general not practical for real-time implementation. The 2D logarithmic search and hierarchical motion estimation algorithms have significantly lower computational cost than full search method. The number of operations 2D logarithmic search algorithm performs is 33 times less than the full search method. The number of operations 3-level hierarchical motion estimation algorithm performs is 66 times less than the full search method. Although the 2D logarithmic search and hierarchical motion estimation algorithms have a lower PSNR performance than full search method, they are usually preferred for real-time implementations because of their much lower computational complexity.


30 VGA fps, 16x16 macroblock, p=16

Algorithm	Number of operations per second
Full search	30 Gops
2D logarithmic	910 Mops
Hierarchical	450 Mops

Table 1. Computational complexity of motion estimation algorithms

CHAPTER 3

SAD REUSE BASED HIERARCHICAL MOTION ESTIMATION ALGORITHM

H.264 standard uses variable block size motion estimation for achieving better video compression efficiency than previous standards. Variable block size motion estimation allows dividing a 16x16 MB into different size partitions and using a different MV for each partition. A 16x16 MB can be divided into two 8x16 or two 16x8 or four 8x8 partitions. Each 8x8 partition can further be partitioned into two 4x8, two 8x4 or four 4x4 partitions. A variable block size motion estimation algorithm, therefore, has to find the best MVs for all partitions of the MB; [1 MV for 16x16 MB] + [2 MVs for 16x8 partitions] + [2 MVs for 8x16 partitions] + [4 MVs for 8x8 partitions] + [8 MVs for 8x4 partitions] + [8 MVs for 4x8 partitions] + [16 MVs for 4x4 partitions] = total 41 MVs.

The best partition for the MB is determined by a mode decision algorithm based on these 41 MVs. The smaller partitions can reduce the amount of residual data. However, using smaller partitions increases the number of motion vectors that have to be transmitted. Since each motion vector requires additional bits to be transmitted, this overhead may outweigh the benefit of reduced residual [2]. Mode decision algorithms usually adopt the partition size based on the frame characteristics, e.g. choosing large partitions in flat, homogeneous regions of a frame and choosing small partitions in areas with high detail and complex motion.

The amount of computation required by full search method is not practical for real-time implementation even for fixed block size motion estimation. Therefore, efficient algorithms are needed to reduce the computational cost for variable block size motion estimation [5, 6, 7]. In this thesis, we have used the SAD reuse based hierarchical motion estimation algorithm for variable block size motion estimation developed by Dr. Ates [7, 8]. The simulation results show that even though our algorithm has a much lower computational cost than full search method, it provides almost as good coding efficiency as full search method.

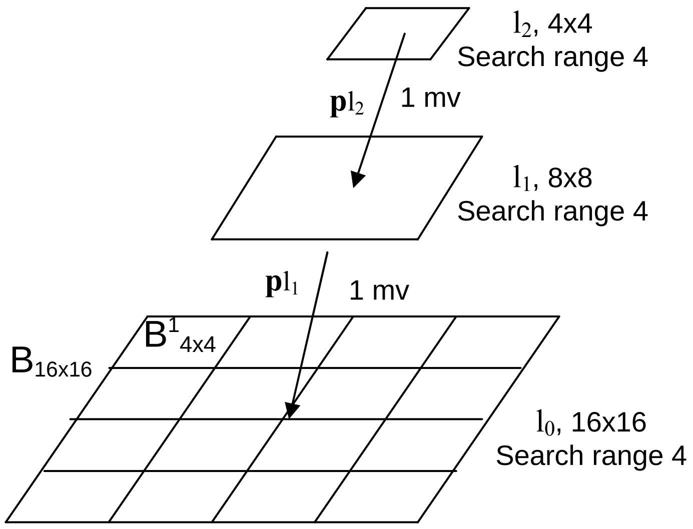


Figure 7. Hierarchical motion estimation algorithm

The algorithm is illustrated in Figure 7. It consists of the following four steps:

1) A 3-level pyramid is constructed using averages of the current MB pixels and the search area pixels. A 4x4 block in level l_2 corresponds to an 8x8 block in level l_1 and a 16x16 MB in level l_0. First, the search area and the 8x8 current block in level$_1$ are generated by down-sampling the search area and the current MB in level$_0$ by 2. Then, the search area and the 4x4 current block in level$_2$ are generated by down-sampling the search area and the current block in level$_1$ by 2.

2) A MV, $\mathbf{p}l_2$, is predicted for the 16x16 MB in level l_0 by performing full search in level l_2 for the 4x4 block within a search range of [-R/4, R/4] ([-R, R] is the search range of the full search method).

3) The MV prediction is refined by performing full search in level l_1 for the 8x8 block at the location pointed by the motion vector $2\mathbf{p}l_2$ in level l_1 within a search range of [-R/4, R/4]. The refined MV prediction is $\mathbf{p}l_1$.

4) The MVs for the 16x16 MB and for all of its partitions are determined by performing full search in level l_0 based on minimizing the Lagrangian cost ($\mathcal{J}(\mathbf{d})$) for all the partitions at both the location pointed by the motion vector $2\mathbf{p}l_1$ and location (0,0) in level l_0 within a limited search range of ([-R/4, R/4]). The Lagrangian cost is computed using the following equations:

$$\mathcal{J}(\mathbf{d}) = \mathrm{SAD}_{\mathrm{B}_{\mathrm{mxn}}}(\mathbf{d}) + \lambda_M R(\mathbf{d} - \mathbf{p}_{med}) \quad (3.1)$$

$$\mathrm{SAD}_{\mathrm{B}_{\mathrm{mxn}}}(\mathbf{d}) = \sum_{x=1, y=1}^{m, n} | c(x, y) - r(x + d_x, y + d_y) | \quad (3.2)$$

where B_{mxn} is a partition of size mxn, (m,n) Є {(4,4), (4,8), (8,4), (8,8), (16,8), (8,16), (16,16)}, $\mathbf{d}$=*(dx, dy)* is the MV, *c* and *r* are current and reference frames respectively, λ_M is the Lagrange multiplier for motion estimation, $\mathbf{p}_{med}$ is the MV prediction used by H.264 video coding standard during the coding process, and $R(\mathbf{d}\text{-}\mathbf{p}_{med})$ specifies the bit-rate spent for coding MV difference information.

The refined MV prediction in level l_1 constitutes a good initial prediction for the 16x16 MB and for all of its partitions in level l_0 when scaled by 2. Therefore, hierarchical motion vector prediction, $2\mathbf{p}l_1$ is used as a MV prediction for the 16x16 MB and for all of its partitions. However, in some cases, $2\mathbf{p}l_1$ is inaccurate for small partitions such as 4x4, using (0,0) vector as an additional MV prediction helps to alleviate this problem.

The full search for a 16x16 MB and for all of its partitions performed in level l_0 requires computing the SADs for all MVs within the search range for all partitions. However, since the full search for a 16x16 MB and for all of its partitions are performed starting at the same location in level l_0 (location pointed by the motion vector $2\mathbf{p}l_1$ or location (0,0)) within the same size search range ([-R/4, R/4]), SADs computed for 4x4 partitions can be reused to compute the SADs for larger partitions, e.g. 8x8, 16x16. In other words, for a given MV **d**=*(dx, dy)*, SAD of B_{mxn} can be decomposed into the SADs of its 4x4 partitions:

$$\mathrm{SAD}_{B_{mxn}}(\mathbf{d}) = \sum_{\substack{k=1,\\ l=1}}^{m/4,n/4} \sum_{\substack{x=4k-3,\\ y=4l-3}}^{4k,4l} |c(x,y) - r(x+d_x, y+d_y)| \qquad (3.3)$$

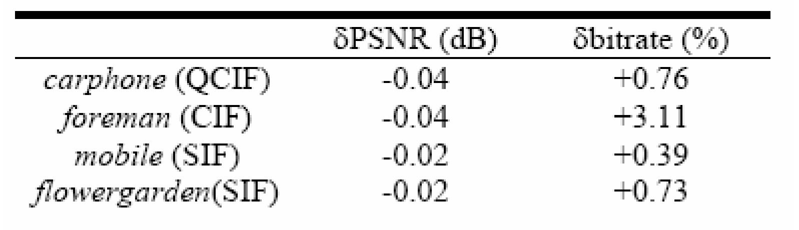
Since all summations on the right are evaluated at the same MV **d**, computing SAD_{Bmxn}(**d**) requires computing the SADs of all its 4x4 partitions for MV **d** and adding them up. This SAD reuse technique decreases the total number of computations significantly.

In order to compare the computational complexity of our algorithm with full search method, we will calculate the number of operations per second needed for performing motion estimation for 30 VGA fps using both our algorithm and full search method with R = 16. As we said before, in order to process 30 VGA fps, 36K MBs per second have to be processed. The SAD reuse based full search method with R = 16 performs (36K MBs per second) x (2x16+1 = 33, 33^2 = 1089 search locations per MB) x (48 operations per search location) = 1.88 Gops for each 4x4 partition of a 16x16 MB. Therefore, in total, full search method performs 16 x 1.88 = 30.1 Gops.

The total number of operations our algorithm performs for processing 30 VGA fps with R = 16 can be calculated as follows. In level l_2, there are $(2x4+1)^2$ = 81 search locations per 4x4 block and 4x4x3=48 operations per search location. Therefore, in level l_2, (36K blocks per second) x (81 search locations per block) x (48 operations per search location) = 140 Mops have to be performed. In level l_1, for each 4x4 partition of the 8x8 block same number of operations as the 4x4 block in level l_2 is performed. Therefore, in level$_1$, 140 x 4 = 560 Mops have to be performed. In level l_0, for each 8x8 partition of the 16x16 MB same number of operations as the 8x8 block in level l_1 is performed. Therefore, in level l_0, 560 x 4 = 2.24 Gops have to be performed. For (0,0) location search, same number of operations as the level l_0 search is performed. Therefore, for (0,0) location search, 2.24 Gops have to be performed as well. Therefore, in total, our algorithm performs 0.14 + 0.56 + 2.24 + 2.24 = 5.18 Gops.

The number of operations our algorithm performs is 5.8 times less than the full search method. And, it still achieves almost as good coding efficiency as full search method. We can further reduce the computational cost of our algorithm by slightly reducing the quality and bit-rate performance. If we eliminate (0,0) location search, our algorithm will perform 10.25 times less number of operations than the full search method. If we use a search range of [-4, 4] in level l_2 and a search range of [-2, 2] in levels l_1 and l_0, our algorithm will perform 30 times less number of operations than the full search method.

The SAD reuse based hierarchical motion estimation algorithm is integrated into the Joint Model (JM) Reference Software Version 7.4 [9]. The updated software is then used to simulate the hierarchical motion estimation algorithm for R=16 using video sequences carphone (QCIF), foreman (CIF), mobile (SIF) and flowergarden (SIF) at 30 fps. All frames except the first one are coded as P-frames. One reference frame is allowed. The Context Adaptive Variable Length Coding (CAVLC) entropy coder is used with quantization parameter values QP = 24, 28, 32, 36. For comparison to full search method, average PSNR loss in dB and percentage changes in bit-rate are reported in Table 2. In addition, at equal bit-rates, PSNR loss is observed to be less than 0.2 dB for all the tested sequences. These results confirm that even though our algorithm has a much lower computational cost than full search method, it provides almost as good coding efficiency as full search method.



	δPSNR (dB)	δbitrate (%)
carphone (QCIF)	-0.04	+0.76
foreman (CIF)	-0.04	+3.11
mobile (SIF)	-0.02	+0.39
flowergarden(SIF)	-0.02	+0.73

Table 2. Performance comparison with full search method

CHAPTER 4

PROPOSED HARDWARE ARCHITECTURE

In this chapter, the proposed hardware architecture for real-time implementation of the SAD reuse based hierarchical motion estimation algorithm described in chapter 3 is explained. The proposed hardware implements the algorithm for the case where R=16, therefore the search ranges used in all 3 levels (l_0, l_1 and l_2) are [-4, 4]. The search window for a [-4, 4] search range contains 9x9 = 81 search locations; 2x4 + 1 = 9 rows and 2x4 + 1 = 9 search locations in each row.

4.1 Current Macroblock and Search Window Storage

The structure of the current and reference frames used for motion estimation are shown in Figure 8. In this thesis, CIF size frames are used. A CIF frame has 352x288 pixels corresponding to 22x18 MBs. The motion estimation process is performed for each 16x16 MB in the current frame. For each 16x16 MB in the current frame, a 64x64 search window from the reference frame is used for motion estimation which means the MVs will be in the range [-24,24]. The current MB and the 64x64 search window are stored in block RAMs in the FPGA.

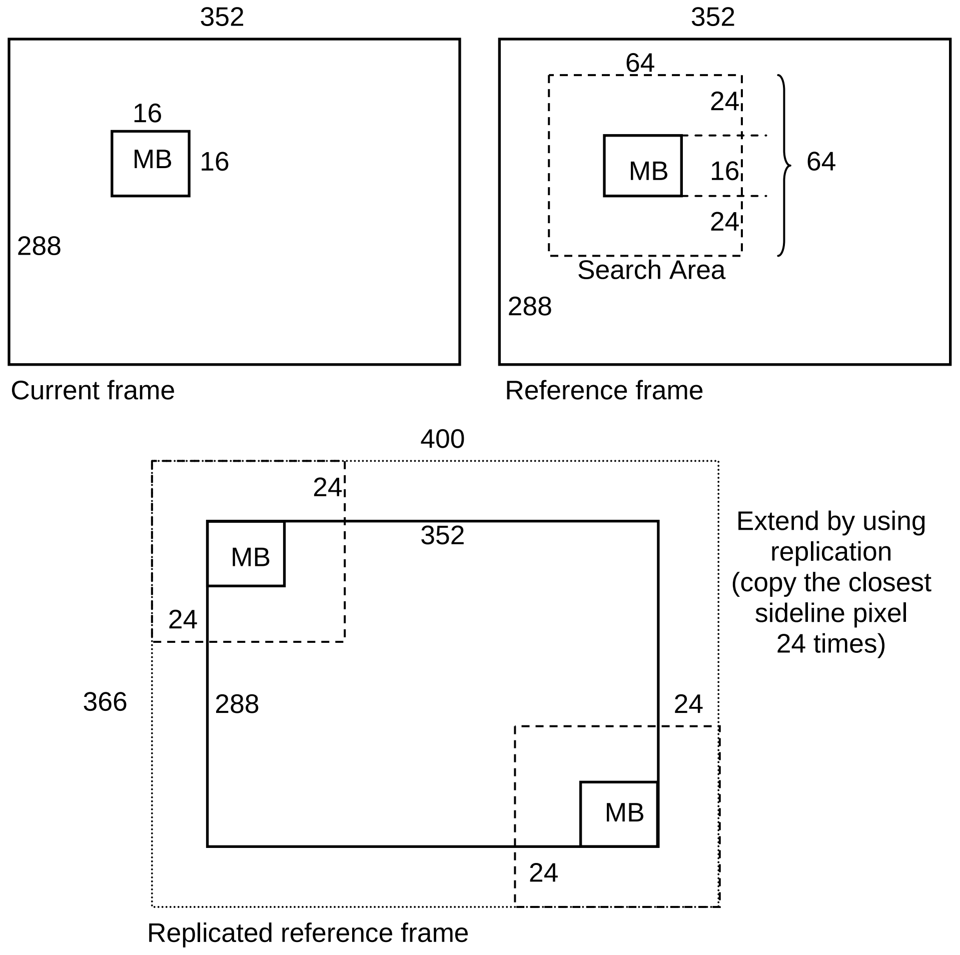

352
16
MB
16
288
Current frame
352
64
24
MB
16
24
64
Search Area
288
Reference frame
400
24
352
MB
24
366
288
24
MB
24
Extend by using replication (copy the closest sideline pixel 24 times)
Replicated reference frame

Figure 8. Current and reference frames

The MV range can be calculated by considering the points on the perimeter of the search range in each level; MV_2= 4 in level l_2 corresponds to 16 in level l_0, MV_1 = 4 in level l_1 corresponds to 8 in level l_0, and level l_0 has MV_0 = 4. This corresponds to a MV range of [-28,28] (16+8+4=28), and a search area of 72x72 (16+28+28=72). However, one block RAM in a Xilinx Virtex II FPGA can store 64x32 pixels. If two block RAMs are used, 64x64 pixels can be stored. Clipping the search area from 72x72 to 64x64 is more efficient than using a third block RAM. Therefore, we reduce the search area from 72x72 to 64x64 by clipping four pixels from left, right, top and bottom. This, in turn, reduces the MV range from 28 to 24. The clipping operation takes place before level l_0 search. It is only needed if the MV from level l_1 points to a location outside the 64x64 search area in level l_0. According to our simulation results, this happens very rarely. Since, in most cases, the MV from level l_1 points to a location inside the 64x64 search area, the clipping operation causes a negligible reduction in the performance. Therefore, we preferred reducing the MV range from [-28,28] to [-24,24] by clipping, instead of using a third block RAM.

As shown in Figure 8, when the current MB is located at the boundary of the current frame, the 64x64 search area exceeds the reference frame by 24 pixels. Therefore, the 352x288 pixels reference frame has to be extended by 24 pixels in each direction. This is done by using replication method. In order to construct the extended reference frame, in each direction, the closest pixel on the perimeter is replicated 24 times producing the 400x336 pixels reference frame.

After finishing the motion estimation process for a MB, we proceed to the motion estimation process for the next MB in the current frame. Therefore, the next MB from the current frame and the corresponding next search window from the reference frame have to be provided to the motion estimation hardware. As shown in Figure 9, the next search window and the current search window overlap by a ratio of 3/4. 64x48 pixels in the next search window overlap with the current search window, only 64x16 pixels of the next search window are different from the current search window. This overlap is exploited and only the non-overlapping 64x16 pixels are transferred for the next search window.

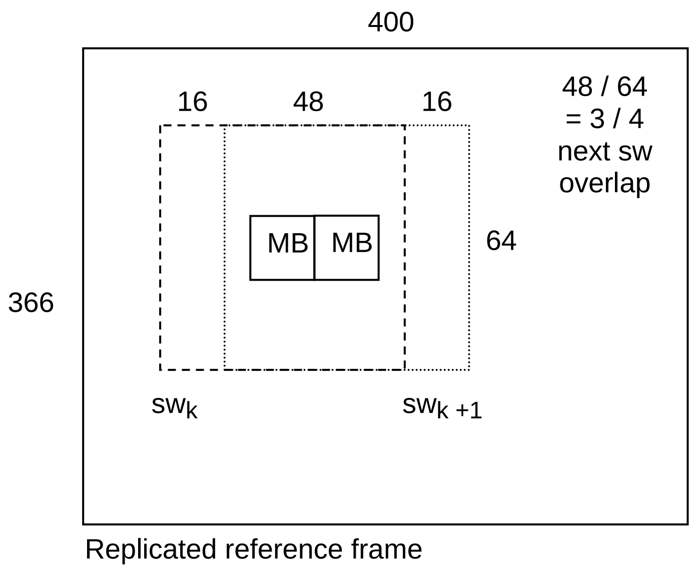


Figure 9. Search window overlap

4.2 Averaging Unit

The proposed hardware first constructs a 3-level pyramid by using the averaging datapath shown in Figure 10 [8]. The datapath is used for generating the current block and search window values in levels l_1 and l_2 by calculating the average of the corresponding pixels in the current MB and search window in level l_0. Each averaging unit calculates the average of 4 pixels in level l_0. The resulting values are stored in register files and they are used to perform full search for the 8x8 block in level l_1 within a search range of [-4, 4]. As soon as the outputs of averaging units A1-A4 are ready, the averaging unit A5 calculates the average of the results produced by A1-A4 which corresponds to the average of 16 pixels in level l_0. The resulting values are stored in register files and they are used to perform full search for the 4x4 block in level l_2 within a search range of [-4, 4].

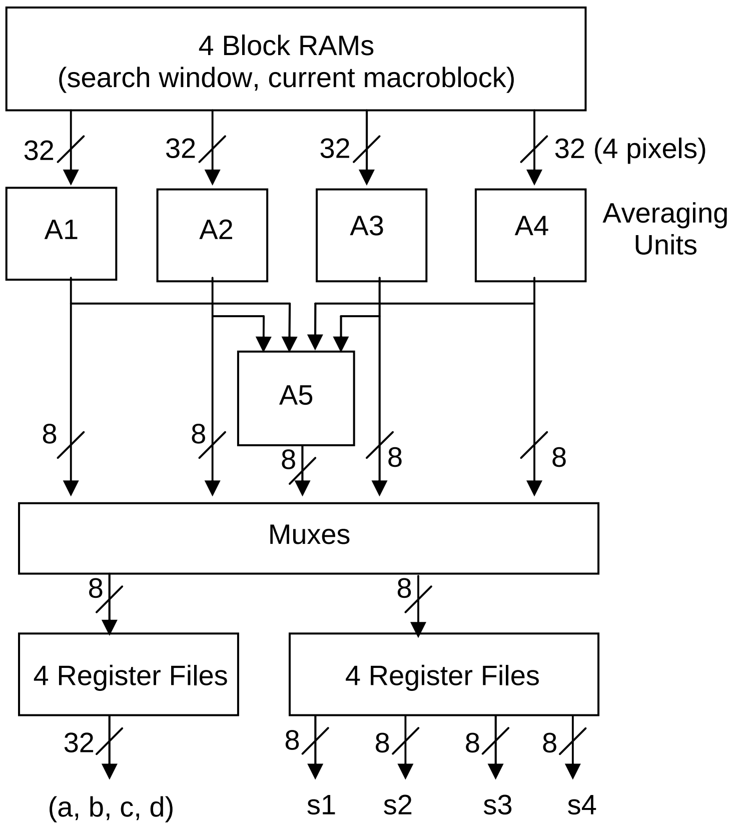


Figure 10. Averaging datapath

The output of the averaging datapath is the current block pixels and the search window pixels required for search operations of levels, and these are the inputs to the 36 processing elements (PE) in the hierarchical motion estimation datapath, shown in Figure 11. The output is provided by the register files. The challenging part of this design is providing the required pixels to the motion estimation search datapath at the right time. This challenge results in a complex control unit, especially for the register files. Although we could provide the pixels of the level l_0 directly from the block RAMs, the pixels of the level l_0 are also provided by the register files in order to reduce the complexity, since the register files already have the control flow algorithm to provide data, after these pixels are written to the register files from the block RAMs. This causes a slight increase in the number of clock cycles, but the gain of reducing the complexity is more important. As mentioned before, the averaging process takes place once and the values for both level l_2 and level l_1 are calculated together.

The averaging process for all the pixels in the current MB and the search window takes 280 clock cycles. At the end of the averaging process, data for level l_2 search is ready in the register files and do not need any further setup. The data setup in the register files for level l_1 search takes 60 cycles. The data transfer from the block RAMs to the register files for level l_0 search takes 150 cycles. Similarly, the data transfer from the block RAMs to the register files for (0,0) location search takes 150 cycles. In total, the averaging process and data setup for the motion estimation datapath take 280+60+150+150 = 640 cycles.

4.3 Hierarchical Search Hardware

The proposed motion estimation hardware performs both the hierarchical MV prediction in levels l_2 and l_1, and motion estimation with SAD reuse in level l_0 using the datapath shown in Figure 11 [8]. The datapath uses 36 PEs divided into four separate groups. Each group has an array of 9 PEs. As we will explain in this section, the reason for using 36 PEs divided into four separate groups is to have an efficient real-time implementation of the motion estimation with SAD reuse in level l_0. The hierarchical

MV prediction in levels l_2 and l_1 are implemented by utilizing the hardware resources used for the motion estimation with SAD reuse in level l_0.

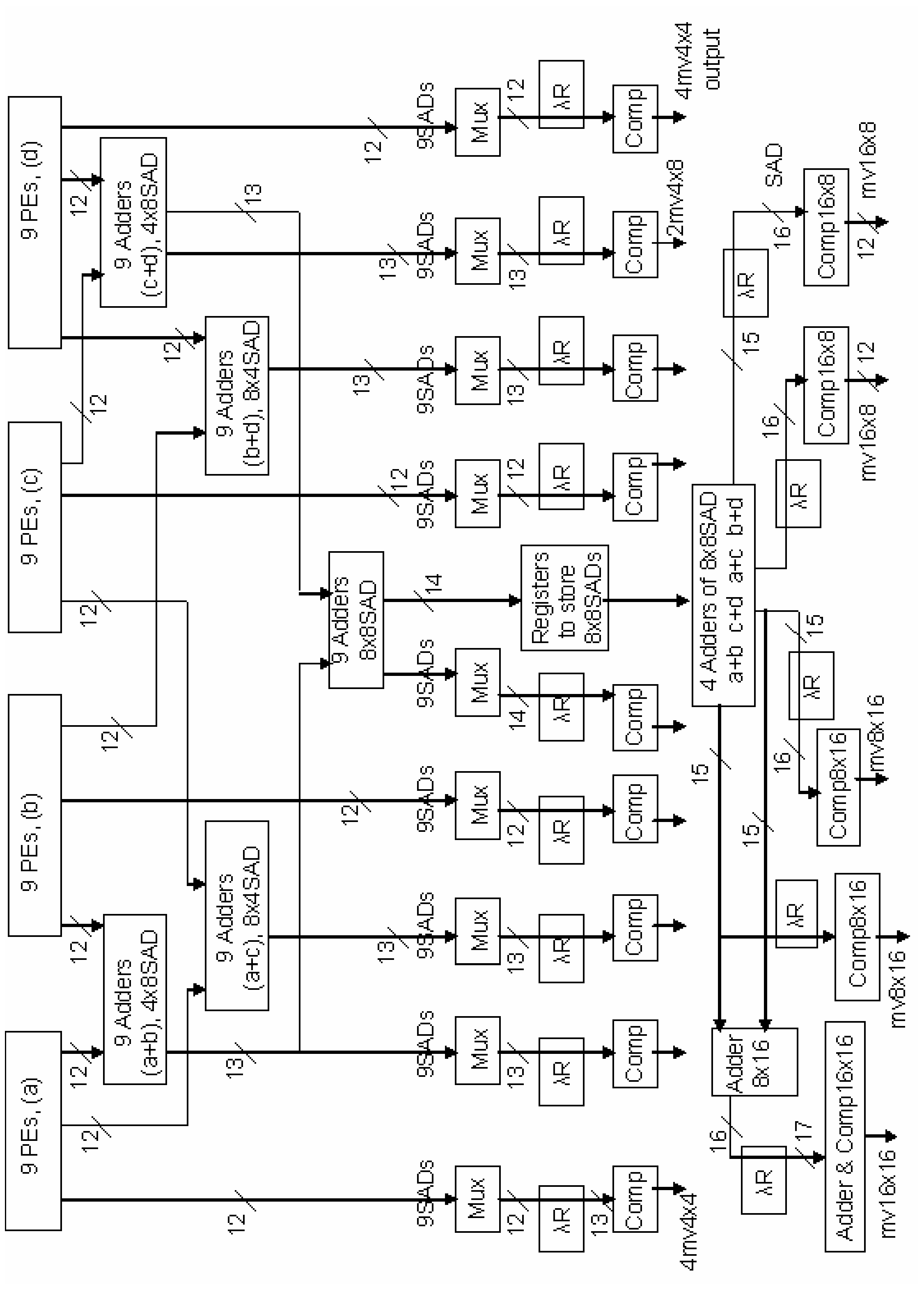


Figure 11. Hierarchical motion estimation datapath

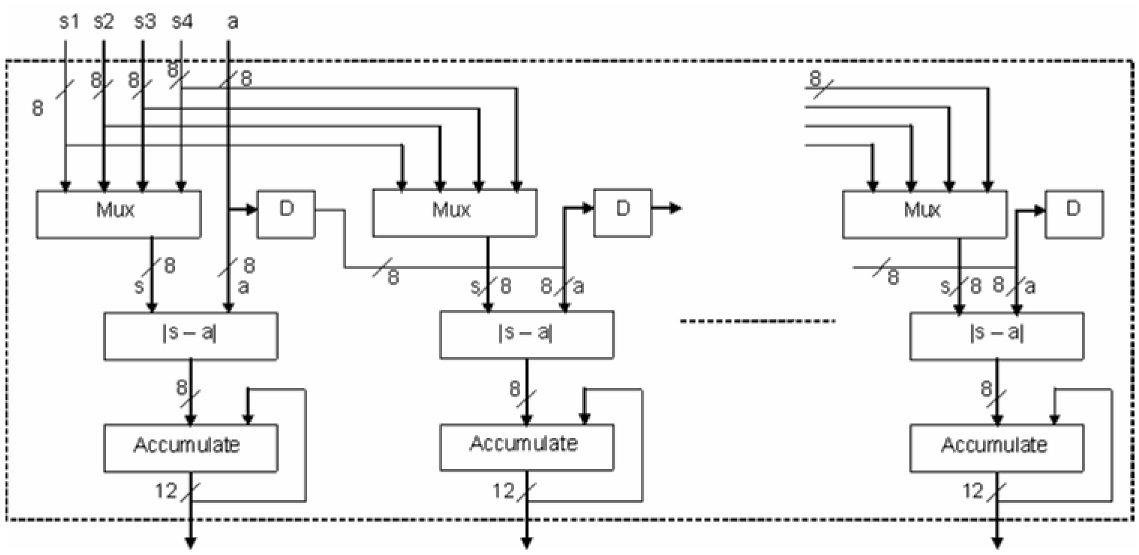


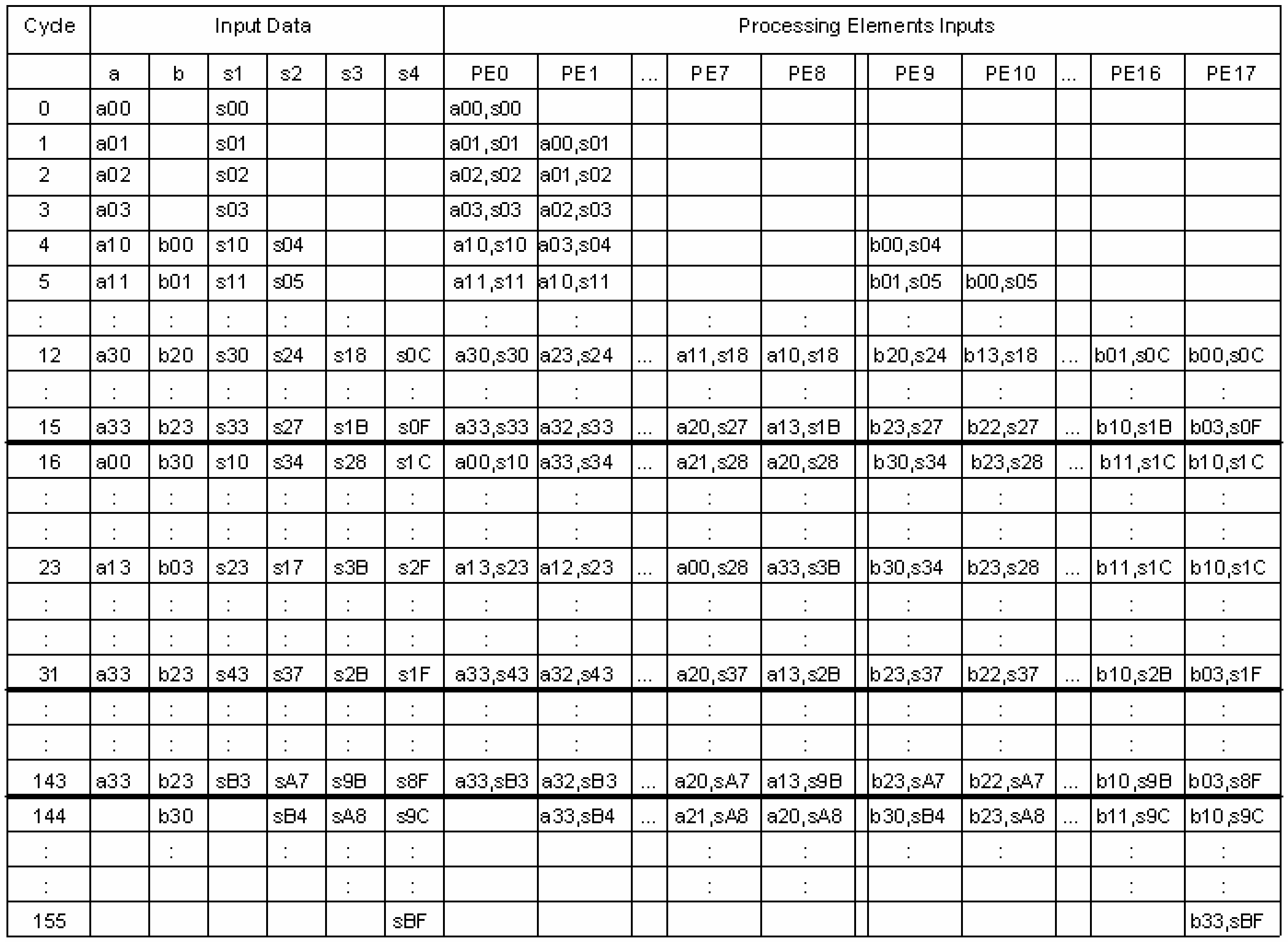
Figure 12. Processing element group

The architecture of a PE and the organization of PEs in a group are shown in Figure 12. Each PE in a PE group gets the current pixel and four search pixels as inputs and selects the appropriate search pixel by a multiplexer. Then, it calculates the absolute difference between the current pixel and the selected search pixel, and accumulates this difference in a register. This is repeated 16 times for the sixteen current block pixels and the corresponding search pixels in order to calculate the SAD value for one search location of a 4x4 block. Each PE sends the current pixel to the next PE, since the next PE starts calculating the SAD value for the same 4x4 block at the next horizontal search location in the search window with one cycle delay. Since there are 9 horizontal search locations in one row of the search window for a 4x4 block, 9 PEs in a PE group are used for calculating the SAD values for the same 4x4 block for all the horizontal search locations in one row of the search window in parallel.

The datapath is first used for the hierarchical MV prediction in level l_2 by performing full search for the 4x4 block in level l_2 within a search range of [-4, 4]. All 36 PEs in the datapath are used to perform the full search as follows. Each PE is used to calculate the SAD value for one search location in the search window. Since there are 9 search locations in one row of the search window, a PE group is used to calculate the SAD values for the search locations in one row of the search window. After each PE group finishes calculating the SAD values for the search locations in one row of the search window, it starts calculating the SAD values for the search locations in another row of the search window. Therefore, each PE group together with a multiplexer and comparator is used to find the minimum SAD in two rows of the search window. All 4 PE groups are, therefore, utilized to find the motion vector $\mathbf{p}l_2$ with the minimum SAD in the search window. The level l_2 search takes 42 clock cycles.

The datapath is then used for the hierarchical MV refinement in level l_1 by performing full search for the 8x8 block at the location pointed by the motion vector $2\mathbf{p}l_2$ in level l_1 within a search range of [-4, 4]. Since there are four 4x4 partitions (a, b, c, and d) in a 8x8 block and there are 9 search locations in one row of the search window, each PE group is used to calculate the SAD values for a 4x4 partition for the search locations in one row of the search window. Each PE in a group calculates the SAD value for its 4x4 partition for one search location in one row of the search window. PE groups 0, 1, 2, and 3 are used for partitions a, b, c, and d respectively. After each PE group finishes calculating the SAD values for its 4x4 partition for the search locations in the current row of the search window, it starts calculating the SAD values for its 4x4 partition in the next row of the search window. After the corresponding processing elements in each PE group, e.g. processing element 0 in each PE group, calculate the SAD value for a search location for its 4x4 partition, the 4x8 SAD adders and 8x8 SAD adders in the datapath are used to calculate the SAD value (SADa + SADb + SADc + SADd) for that search location for the 8x8 partition in the same cycle. The multiplexer and comparator at the outputs of the 8x8 SAD adders are used to find the minimum SAD for the 8x8 partition and the corresponding motion vector $\mathbf{p}l_1$ in the search window. The level l_1 search takes 156 clock cycles.

The datapath is finally used for the motion estimation with SAD reuse in level l_0. It is used to perform full search based on minimizing the Lagrangian cost for the 16x16 current MB and for all of its partitions at both the location pointed by the motion vector $2\mathbf{p}l_1$ and location (0,0) within a search range of [-4, 4] to determine the 41 best motion vectors for all partitions of the MB. The datapath is designed to use the SAD reuse technique for performing full search for a 16x16 MB and for all of its partitions within a search range of [-4, 4]. Each PE group in the datapath together with a multiplexer and comparator is used to perform full search for a 4x4 partition of the 16x16 MB within a search range of [-4, 4]. Since there are 9 search locations in one row of the search window, 9 PEs are grouped together to calculate the SAD values for a 4x4 partition for the search locations in one row of the search window. Each processing element in a group calculates the SAD value for a 4x4 partition for one search location in one row of the search window.



Cycle	Input Data						Processing Elements Inputs									
	a	b	s1	s2	s3	s4	PE0	PE1	...	PE7	PE8	PE9	PE10	...	PE16	PE17
0	a00		s00				a00,s00									
1	a01		s01				a01,s01	a00,s01								
2	a02		s02				a02,s02	a01,s02								
3	a03		s03				a03,s03	a02,s03								
4	a10	b00	s10	s04			a10,s10	a03,s04				b00,s04				
5	a11	b01	s11	s05			a11,s11	a10,s11				b01,s05	b00,s05			
:	:	:	:	:	:		:	:		:	:	:	:		:	
12	a30	b20	s30	s24	s18	s0C	a30,s30	a23,s24	...	a11,s18	a10,s18	b20,s24	b13,s18	...	b01,s0C	b00,s0C
:	:	:	:	:	:	:	:	:		:	:	:	:		:	:
15	a33	b23	s33	s27	s1B	s0F	a33,s33	a32,s33	...	a20,s27	a13,s1B	b23,s27	b22,s27	...	b10,s1B	b03,s0F
16	a00	b30	s10	s34	s28	s1C	a00,s10	a33,s34	...	a21,s28	a20,s28	b30,s34	b23,s28	...	b11,s1C	b10,s1C
:	:	:	:	:	:	:	:	:		:	:	:	:		:	:
:	:	:	:	:	:	:	:	:		:	:	:	:		:	:
23	a13	b03	s23	s17	s3B	s2F	a13,s23	a12,s23	...	a00,s28	a33,s3B	b30,s34	b23,s28	...	b11,s1C	b10,s1C
:	:	:	:	:	:	:	:	:		:	:	:	:		:	:
:	:	:	:	:	:	:	:	:		:	:	:	:		:	:
31	a33	b23	s43	s37	s2B	s1F	a33,s43	a32,s43	...	a20,s37	a13,s2B	b23,s37	b22,s37	...	b10,s2B	b03,s1F
:	:	:	:	:	:	:	:	:		:	:	:	:		:	:
:	:	:	:	:	:	:	:	:		:	:	:	:		:	:
143	a33	b23	sB3	sA7	s9B	s8F	a33,sB3	a32,sB3	...	a20,sA7	a13,s9B	b23,sA7	b22,sA7	...	b10,s9B	b03,s8F
144		b30		sB4	sA8	s9C		a33,sB4	...	a21,sA8	a20,sA8	b30,sB4	b23,sA8	...	b11,s9C	b10,s9C
:		:		:	:	:				:	:	:	:		:	:
:					:	:				:	:				:	:
155						sBF										b33,sBF

Figure 13. Data flow for processing elements PE0-PE17

As it is shown in Figure 13, in order to reduce the number of current block and search window register ports and number of accesses to these registers, each PE in a group starts calculating its SAD value one cycle later than the previous PE in that group so that PEs can reuse the current block value accessed by the first PE in the group and several PEs can use the same search window value in the same cycle [8]. Since PE0 starts working in cycle 0, it finishes calculating its first SAD in cycle 15. The last PE in that group, PE8, finishes calculating its SAD in cycle 8 + 15 = 23. After each PE finishes calculating an SAD value for a 4x4 partition in the current row of the search window, it starts calculating an SAD value for the same 4x4 partition in the next row of the search window. Since there are 9 rows in the search window, the minimum SAD for a 4x4 partition and the corresponding motion vector is found in 8 + 9x16 = 152 cycles.

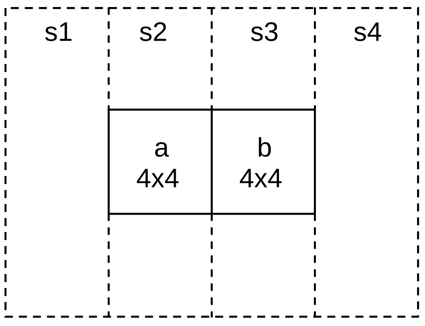


Figure 14. Search window overlap of two neighboring 4x4 partitions

Since the full search for a 16x16 MB and for all of its partitions are performed starting at the same location in level l_0 (location pointed by the motion vector $2\mathbf{p}l_1$ or location (0,0)) within the same size search range ([-4, 4]), the search windows of two neighboring 4x4 partitions (a, b) of the MB overlap as shown in Figure 14. The search window regions s1, s2 and s3 are used for partition a, and the search window regions s2, s3 and s4 are used for partition b. Therefore, the search window regions s2 and s3 are shared by both a and b partitions. In order to exploit this for reducing the number of search window register ports (from 3+3=6 to 4) and the number of accesses to search window registers, the full search for partitions a and b are performed simultaneously by using PE group 0 for partition a and PE group 1 for partition b. As it is shown in Figure 13, the processing elements in PE group 1 starts calculating their SADs 4 cycles later than the corresponding processing elements in PE group 0 so that several PEs in group 0 and group 1 can use the same search window value (in regions s2 or s3) in the same cycle. Therefore, the minimum SAD for partition b and the corresponding motion vector is found in 4+152 = 156 cycles.

As the PE groups 0 and 1 perform the full search for partitions a and b, PE groups 2 and 3 perform the full search for partitions c and d simultaneously based on the same data flow shown in Figure 13. Therefore, the minimum SADs for 4x4 partitions a, b, c, d and the corresponding motion vectors are found in 156 cycles.

After the corresponding processing elements in each PE group, e.g. processing element 0 in each PE group, calculate the SAD value for a search location for its 4x4 partition, the 4x8 SAD, 8x4 SAD and 8x8 SAD adders in the datapath are used to calculate the SAD values for that search location for the 4x8 (a+b and c+d), 8x4 (a+c and b+d), and 8x8 (a+b+c+d) partitions by reusing the SAD values of the 4x4 partitions. In other words, as the full search for 4x4 partitions a, b, c, and d are performed, the full search for two 4x8 (a+b and c+d), two 8x4 (a+c and b+d), and one 8x8 (a+b+c+d) partition are also performed in parallel by using the 4x8 SAD, 8x4 SAD and 8x8 SAD adders and the multiplexers and comparators at their outputs in the datapath. Therefore, by using the SAD reuse technique, the minimum SADs for two 4x8, two 8x4 and one 8x8 partition and the corresponding motion vectors are found as well in the same 156 cycles.

After the full search for the first four 4x4 partitions are performed, the four PE groups are used to perform the full search for the next four 4x4 partitions of the MB. Again, by using the SAD reuse technique, the full search for the corresponding two 4x8, two 8x4, and one 8x8 partition are performed in parallel. Since there are four 8x8 partitions in a MB, this process is repeated 4 times. Therefore, full search for all 4x4, 4x8, 8x4 and 8x8 partitions are performed in 4x156 = 624 clock cycles.

As the full search for the four 8x8 partitions are performed, the full search for 8x16, 16x8 and 16x16 partitions are also performed in parallel by using the 8x16 SAD, 16x8 SAD and 16x16 SAD registers, adders and comparators in the datapath. The SAD reuse operations for partitions larger than 8x8 are performed by using an adder for each partition, and the minimum SAD for each partition is obtained by using a comparator for each partition. For the first three 8x8 partitions, at the end of full search for each 8x8 partition, all 8x8 SADs are stored in a register file. During the full search for the second 8x8 partition, first 8x16 SADs are calculated by reusing the first and second 8x8 partition SADs, and they are stored in a register file. During the full search for the third 8x8 partition, first 16x8 SADs are calculated by reusing the first and third 8x8 partition SADs. During the full search for the fourth 8x8 partition, second 8x16 SADs are calculated by reusing the third and fourth 8x8 partition SADs, second 16x8 SADs are calculated by reusing the second and fourth 8x8 partition SADs, and 16x16 SADs are calculated by reusing the first and second 8x16 partition SADs. Therefore, by using the SAD reuse technique, the minimum SADs for 8x16, 16x8 and 16x16 partitions and the corresponding motion vectors are found as well in the same 624 clock cycles.

After the full search for the 16x16 current MB and for all of its partitions at the location pointed by the motion vector $2\mathbf{p}l_1$ within a search range of [-4, 4] are performed, the full search for the same MB and for all of its partitions are performed at location (0, 0) within a search range of [-4, 4] by using the same datapath with the same data flow. This process takes 624 clock cycles as well.

Finally, the first 41 motion vectors determined by the full search for the 16x16 current MB and for all of its partitions at the location pointed by the motion vector $2\mathbf{p}l_1$ and the other 41 motion vectors determined by the full search for the same MB and for all of its partitions at location (0, 0) are compared based on the Lagrangian cost criterion (SAD+λ*R) and the 41 motion vectors with minimum cost are output from the motion estimation hardware.

4.4 Bit-rate Cost (λR) Calculation Hardware

Since the final 41 motion vectors are obtained in level l_0, we use the Lagrangian cost criterion (SAD+λ*R) only in level l_0 and (0,0) location search. For level l_2 and l_1 search, we use minimum SAD criterion. Therefore, bit-rate cost calculation (λ*R) hardware is not used and λ*R is taken as zero in levels l_1 and l_2 search. In level l_0 and the (0,0) location search, proper λ*R value is calculated and added to the SAD value before each comparator as shown in Figure 11.

λ is an input to the motion estimation module and same λ value is used for the entire frame. R represents the bit-rate cost (the number of bits required for coding the motion vector difference ($\mathbf{p}_{cand}$ - $\mathbf{p}_{med}$)) and proper R value for each SAD is calculated in the motion estimation hardware. $\mathbf{p}_{cand}$ is the motion vector for the current search location and $\mathbf{p}_{med}$ is the predicted motion vector for the current search location. The $\mathbf{p}_{cand}$ and $\mathbf{p}_{med}$ values for each search location is calculated and provided to the corresponding comparators in the proper cycles.

After obtaining $\mathbf{p}_{cand}$ and $\mathbf{p}_{med}$, 4*($\mathbf{p}_{cand}$ - $\mathbf{p}_{med}$) is calculated and the number of bits required for coding this motion vector difference is determined. Since we don't perform sub-pixel (half-pixel and quarter-pixel) motion estimation, the difference is multiplied by 4 in order to create an interval of 4 between the consecutive integer pixels. The number of bits required for coding x and y coordinates of the motion vector difference are calculated separately, and they are added to obtain the number of bits required for coding the motion vector. If the motion vector difference is 0, then it is coded using only one bit. Otherwise, if the absolute value of the difference is n bits, then it is coded using 2n+1 bits [2]. The number of bits required for coding the motion vector difference (R) is multiplied by λ and added to SAD to obtain the Lagrangian cost criterion (SAD+λ*R).

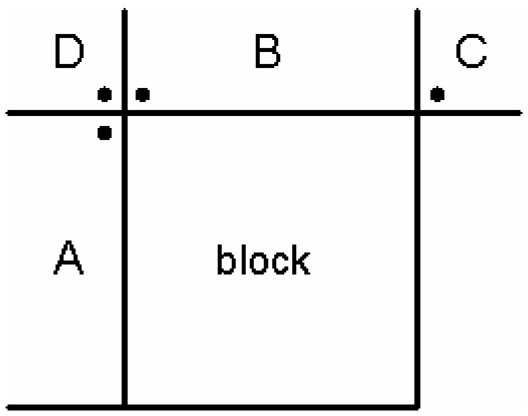


Figure 15. Motion vector prediction

The $\mathbf{p}_{med}$ for a block depends on MV_A, MV_B and MV_C (the motion vectors of three neighbor blocks A, B and C) as shown in Figure 15. MV_A is the motion vector for the partition of left neighbor block A including the top-right pixel of this block. MV_B is the motion vector for the partition of top neighbor block B including the bottom-left pixel of this block. MV_C is the motion vector for the partition of top-right neighbor block C including the bottom-left pixel of this block. These motion vectors are provided as inputs to the motion estimation hardware. The $\mathbf{p}_{med}$ is determined based on the availability of these motion vectors as follows [4]:

1) if MV_C is not available, then $MV_C = MV_D$
2) if MV_A is not available, then $MV_A = 0$
   if MV_B is not available, then $MV_B = 0$
   if MV_C is not available, then $MV_C = 0$
3) if MV_B and MV_C are not available, then $\mathbf{p}_{med} = MV_A$
   else $\mathbf{p}_{med}$ = median (MV_A, MV_B, MV_C)

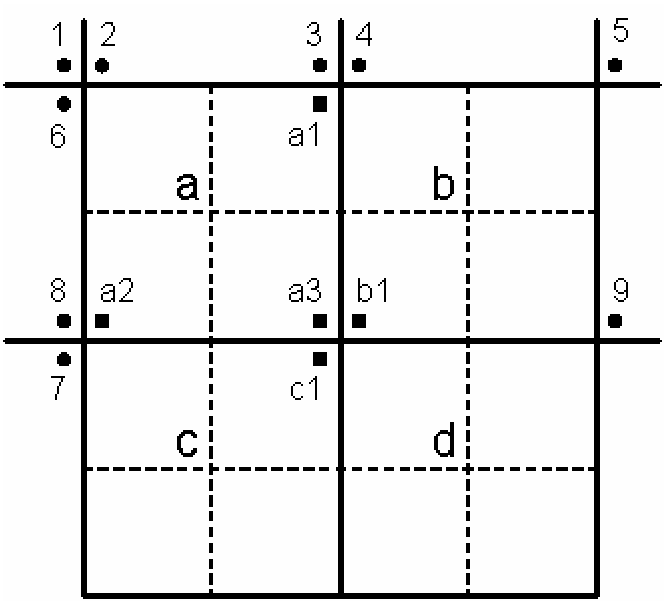


Figure 16. Motion vector prediction for 8x8 blocks

The $\mathbf{p}_{med}$ is calculated for each 8x8 block in a 16x16 MB before processing that 8x8 block, and this value is also used for the smaller partitions (4x4, 4x8 and 8x4) of that 8x8 block. The motion vectors used to determine the $\mathbf{p}_{med}$ for each 8x8 block are shown in Figure 16. The motion vectors A_a, B_a, C_a and D_a used for the 8x8 block a are the motion vectors for pixels 6, 2, 4 and 1 respectively. The motion vectors A_b, B_b, C_b and D_b used for the 8x8 block b are the motion vectors for pixels a1, 4, 5 and 3 respectively. The motion vectors A_c, B_c, C_c and D_c used for the 8x8 block c are the motion vectors for pixels 7, a2, b1 and 8 respectively. The motion vectors A_d, B_d, C_d and D_d used for the 8x8 block d are the motion vectors for pixels c1, b1, 9 and a3 respectively.

The circle shaped motion vectors are the motion vectors of the neighbor MBs and they are provided as inputs to the motion estimation hardware. The square shaped motion vectors are the motion vectors of the partitions inside the current 16x16 MB and they are calculated during the motion estimation process. These motion vectors are always available. Since the motion vector for pixel 9 is from the next MB, it is never available. Therefore, due to the availability, the motion vector for pixel a3 is used for C_d instead of the motion vector for pixel 9. Since the motion vector for pixel b1 is always available for C_c, the motion vector for pixel 8 is never used. Therefore, 7 motion vectors from neighbor MBs are required for the current 16x16 MB and will be provided as inputs to the motion estimation hardware. In addition, 5 motion vectors from the partitions inside the current 16x16 MB are required and will be calculated during the motion estimation process.

The $\mathbf{p}_{med}$ for 8x8 block a is calculated using the motion vectors from neighbor MBs. Then, motion estimations for the 8x8 block a and for all of its partitions are performed based on Lagrangian cost criterion (SAD+λ*R). Then, mode decision for this 8x8 block is performed by comparing the SAD+λ*R values for the 4x4, 4x8, 8x4 and 8x8 modes and selecting the mode with the minimum SAD+λ*R value. After the mode decision, the motion vectors for pixels a1, a2 and a3 (the motion vectors inside this block required for processing the following 8x8 blocks) are obtained based on the modes of the partitions including these pixels. This process is repeated for the 8x8 blocks b, c and d.

The $\mathbf{p}_{med}$ for 8x8 block b is calculated using the motion vector for pixel a1 and the motion vectors from neighbor MBs. Then, motion estimations for the 8x8 block b and for all of its partitions are performed based on Lagrangian cost criterion (SAD+λ*R). Then, mode decision for this 8x8 block is performed. After the mode decision, the motion vector for pixel b1 (the motion vector inside this block required for processing the following 8x8 blocks) is obtained based on the mode of the partition including this pixel.

The $\mathbf{p}_{med}$ for 8x8 block c is calculated using the motion vectors for pixels a2 and b1 and a motion vector from a neighbor MB. Then, motion estimations for the 8x8 block c and for all of its partitions are performed based on Lagrangian cost criterion (SAD+λ*R). Then, mode decision for this 8x8 block is performed. After the mode decision, the motion vector for pixel c1 (the motion vector inside this block required for processing the following 8x8 blocks) is obtained based on the mode of the partition including this pixel.

The $\mathbf{p}_{med}$ for 8x8 block d is calculated using the motion vectors for pixels b1, c1 and a3. Then, motion estimations for the 8x8 block d and for all of its partitions are performed based on Lagrangian cost criterion (SAD+λ*R).

4.5 Performance Analysis

Assuming that the current MB and search window pixels are ready in the FPGA block RAMs, the proposed motion estimation hardware determines the 41 best motion vectors for all partitions of a MB in 2100 clock cycles; 640 (averaging and data setup) + 14 (transitions between the levels) + 42 (level l_2 search) + 156 (level l_1 search) + 624 (level l_0 search) + 624 ((0,0) location search) = 2100. It is possible to reduce this total cycle count significantly by increasing the area and/or slightly reducing the quality and bit-rate performance. We have reported the performances of several alternative architectures in Table 3.

Architecture	Cycle count
Perform (0,0) location search	
36 PEs	2100
72 PEs	1190
144 PEs	735
Eliminate (0,0) location search	
36 PEs	1330
72 PEs	805
144 PEs	543

Table 3. Performances of alternative architectures

If 72 PEs are used in the datapath, instead of 36 PEs, the number of cycles spent for data setup and search operations in all levels will be reduced by half, but the averaging operations will still take 280 cycles. Therefore, if 72 PEs are used, the total cycle count will decrease to 1190; (2100-280) / 2 = 910 + 280 = 1190. If 144 PEs are used in the datapath, the total cycle count will further decrease to 735.

If the (0,0) location search in level l_0 is not performed, the total cycle count will be reduced by 770 cycles (146 cycles for data setup + 624 cycles for search). Therefore, the total cycle count will decrease to 2100-770 = 1330 cycles. In addition, if 72 PEs are used in the datapath, the total cycle count will decrease to 805. If instead 144 PEs are used in the datapath, the total cycle count will further decrease to 543.

Before the motion estimation process for a MB can be started, the current MB and search window pixels have to be transferred to the block RAMs in the FPGA. The number of cycles needed for this transfer depends on the memory bandwidth available in the complete H.264 video coding system in which our motion estimation hardware is used. For example, if the H.264 video coding system is implemented on ARM Versatile/PB926EJ-S development board, we can transfer 32 bits on the bus in one cycle. Therefore, transferring 64x64 search window takes 64x64x8 / 32 = 1024 clock cycles. As we mentioned before, the search window for the next MB and the current search window overlap by a ratio of 3/4. Since we exploit this overlap and transfer only the 64x16 non-overlapping pixels for the next search window, transferring the next search window takes 1024 / 4 = 256 clock cycles. We cannot exploit this overlap for the MBs in the first column of a frame, e.g. for 18 MBs in a CIF frame. Therefore, for a CIF frame, transferring the search window pixels into the block RAMs in the FPGA takes (256x378 + 1024x18) / 396 = 290 clock cycles per MB.

We implemented a software model for the SAD reuse based hierarchical motion estimation algorithm in C. We measured the execution time of the software model on an ARM926EJ-S processor using ARM AXD debugger and ARMulator. We used ARMulator with the default memory model which models a zero wait state memory system. The ARM926EJ-S processor executes 1.6M instructions to perform the motion estimation for a MB and this takes 2.5M cycles. The averaging for a frame takes 4.2M cycles. Therefore, the total cycle count of the software model on an ARM926EJ-S processor for a frame is 2.5x396 + 4.2 = 1G. On the other hand, the total cycle count of our hardware architecture for a frame is (2100+290) x 396 = 950K. This means, in terms of total cycle count, our proposed hardware architecture is 1050 times faster than the software model.

4.6 Comparison with Previous Work

A hardware architecture for real-time implementation of a variable block size motion estimation algorithm for H.264 video coding is presented in [10]. This hardware architecture implements the SAD reuse based full search algorithm by using 256 PEs and it takes 1583 cycles for a search range of horizontal p = 24 and vertical p = 16. Our architecture, on the other hand, implements an SAD reuse based hierarchical motion estimation algorithm by using only 36 PEs and it takes 2100 cycles for a search range of [-24,24] (both horizontal and vertical p = 16). Their hardware architecture achieves higher performance than our hardware design at the expense of a much higher hardware cost; 256 PEs as opposed to 36 PEs. Our hardware design is a more cost-effective solution for portable applications. If we compare the two designs based on (Area x Speed) criterion, our design is (256 x 1583) / (36 x 2100) = 5.36 times better than their design. In addition, their design uses the SAD as the cost criterion for selecting the motion vectors. In comparison, our design uses the Lagrangian cost criterion (SAD+λ*R) which produces better quality results.

CHAPTER 5

HARDWARE IMPLEMENTATION

The proposed architecture is implemented in Verilog HDL. The implementation is verified with RTL simulations using Mentor Graphics ModelSim SE. For the RTL simulation, a testbench is written for sending the current MB and corresponding search window pixels to the block RAMs before the motion estimation process for each MB, and for checking the motion vectors produced by the motion estimation hardware. A software model for the SAD reuse based hierarchical motion estimation algorithm is implemented in C. The software model is used for verifying the RTL design by comparing their outputs for a randomly generated CIF size current frame and a reference frame. Both the software model and the RTL design are simulated for all 396 MBs in the current frame, and both the output motion vectors and the corresponding minimum SAD+λR values for all MBs are compared. The outputs of the software and hardware simulations for the entire frame exactly matched, verifying the RTL design.

The Verilog RTL is then synthesized to a 2V8000ff1152 Xilinx Virtex II FPGA with speed grade 5 using Mentor Graphics Leonardo Spectrum. The resulting netlist is placed and routed to the same FPGA using Xilinx ISE Series 5.2i. The FPGA implementation is verified to work at 63 MHz under worst-case PVT conditions with post place and route simulations. The post place and route simulation is performed as explained in the Xilinx Synthesis and Simulation Design Guide [11]. First, a post place and route simulation model is generated. Then, a testbench specific to Xilinx Virtex II FPGA is written. The testbench instantiates a global module that declares the global signals and a startup module that deactivates the global set/reset and tri-state signals. Finally, the post place and route simulation is performed for an entire CIF frame using the same current and reference frames used for software and RTL simulations. As in the case of RTL simulation, the testbench sends the current MB and corresponding search window pixels to the block RAMs before the motion estimation process for each MB, and checks the motion vectors produced by the motion estimation hardware. The outputs of the post place and route simulation exactly matched the outputs of software and RTL simulations for the entire frame, verifying the placed and routed design.

The FPGA implementation can process a VGA frame in 40 msec. (1200 MBs * 2100 clock cycles per MB * 15.87 ns clock cycle = 40 msec) Therefore, it can process 1000/40 = 25 VGA frames (640x480) per second. The FPGA implementation can process a CIF frame in 12.2 msec. (396 MBs * 2100 clock cycles per MB * 15.87 ns clock cycle = 13.2 msec) Therefore, it can process 1000/13.2 = 76 CIF frames (352x288) per second.

The FPGA implementation including all datapaths and control units, input, output and internal RAMs and register files uses the following FPGA resources; 14255 Function Generators, 7128 CLB Slices, 5220 Dffs/Latches, 13 Block RAMs, and 7 Block Multipliers (used for calculating λ*R), i.e. %15.3 of Function Generators, %15.3 of CLB Slices, %5.4 of Dffs/Latches, %7.7 of Block RAMs, and %4.1 of Block Multipliers. The equivalent gate count for the overall design is 1.18M.

The averaging and hierarchical motion estimation datapaths shown in Figures 11 and 12 use following FPGA resources; 5400 Function Generators, 2700 CLB Slices, 2926 Dffs/Latches, 13 Block RAMs, and 7 Block Multipliers (used for calculating λ*R), i.e. %5.8 of Function Generators, %5.8 of CLB Slices, %3 of Dffs/Latches, %7.7 of Block RAMs, and %4.1 of Block Multipliers. The equivalent gate count for the two datapaths is approximately 480K.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, a high performance and low cost hardware architecture is designed for real-time implementation of an SAD reuse based hierarchical motion estimation algorithm for H.264 / MPEG4 Part 10 video coding. This hardware is designed to be used as part of a complete H.264 video coding system for portable applications. The proposed architecture is implemented in Verilog HDL. The Verilog RTL code is verified to work at 63 MHz in a Xilinx Virtex II FPGA. The FPGA implementation can process 25 VGA frames (640x480) or 76 CIF frames (352x288) per second.

Our hardware design can be integrated into a complete H.264 video coding system as a future work. The inputs to our design are the current MB and the search window pixels, therefore, during the integration only the memory bandwidth and bus speed in the specific implementation platform should be considered. The hardware design can be implemented as an ASIC in order to increase the operating frequency and the number of frames processed per second.

Our hardware architecture can be extended to perform sub-pixel accurate motion estimation. It can also be extended to perform motion estimation using multiple reference frames. Improved motion estimation algorithms, e.g. algorithms with different motion vector selection criterion or with different search ranges, can also be implemented by using the proposed datapath with a new control unit.

REFERENCES

[1] Vasudev Bhaskaran and Konstantinos Konstantinides, Image and Video Compression Standards, Algorithms and Architectures, Kluwer Academic Publishers, 1997

[2] Iain E. G. Richardson, H.264 and MPEG-4 Video Compression, Wiley, 2003

[3] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC Video Coding Standard", IEEE Trans. on Circuits and Systems for Video Technology, vol. 13, no. 7, pp. 560-576, July 2003

[4] Joint Video Team (JVT) of ITU-T VCEG and ISO/IEC MPEG, Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification, ITU-T Rec. H.264 and ISO/IEC 14496-10 AVC, May 2003

[5] H. C. Tourapis and A. M. Tourapis, "Fast Motion Estimation within the H.264 Codec", Proc. IEEE Int. Conf. Multimedia and Expo, vol. 3, pp. 517-520, July 2003

[6] Jae Hun Lee and Nam Suk Lee, "Variable Block Size Motion Estimation Algorithm and its Hardware Architecture for H.264/AVC", Proc. IEEE ISCAS, 2004

[7] H. F. Ates and Y. Altunbasak, "SAD Reuse in Hierarchical Motion Estimation for the H.264 Encoder", Proc. IEEE Int. Conf. on Acoustics, Speech and Signal Processing, March 2005

[8] Sinan Yalcin, Hasan Ates, and Ilker Hamzaoglu, "A High Performance Hardware Architecture for an SAD Reuse based Hierarchical Motion Estimation Algorithm for H.264 Video Coding", Proc. Int. Conf. on Field Programmable Logic and Applications, August 2005.

[9] Joint Video Team (JVT) of ITU-T VCEG and ISO/IEC MPEG, Joint Model (JM) Reference Software Version 7.4, http://iphome.hhi.de/suehring/tml

[10] Yu-Wen Huang, Tu-Chih Wang, Bing-Yu Hsieh, and Liang-Gee Chen, "Hardware Architecture Design for Variable Block Size Motion Estimation in MPEG-4 AVC/JVT/ITU-T H.264", Proc. IEEE ISCAS, May 2003

[11] Synthesis and Simulation Design Guide, Xilinx Inc., http://www.xilinx.com