Variable Block Size Motion Estimation Hardware for Video Encoders

LI Man Ho

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

©The Chinese University of Hong Kong
Nov 2006

Abstract of thesis entitled:
Variable Block Size Motion Estimation Hardware for Video Encoders
Submitted by LI Man Ho
for the degree of Master of Philosophy
at The Chinese University of Hong Kong in Nov 2006

Multimedia has experienced massive growth in recent years due to improvements in algorithms and technology. An important underlying technology is video coding and in recent years, compression efficiency and complexity have also improved significantly. Applications of video coding have moved from set-top boxes to internet delivery and mobile communications.

H.264/AVC is the latest video coding standard adopting variable block size, quarter-pixel accuracy, motion vector prediction and multi-reference frames for motion estimations. These new features result in higher computation requirements than that for previous coding standards. In this thesis, we propose a family of motion estimation processors to balance tradeoffs between the performance, area, bandwidth and power consumption on an field programmable gate array (FPGA) platform.

We combine algorithmic and arithmetic optimizations for motion estimation. At the algorithmic level, we compare different algorithms and analyze their complexities. At the arithmetic level, we explore bit-parallel and bit-serial designs, which employ non-redundant and redundant number systems. In our bit-serial design, we study tradeoffs between least significant bit first (LSB-first) and most significant first (MSB-first) modes.

Finally, we offer a library of motion estimation processors to suit different applications. For bit-parallel processors, we offer 1-dimensional, 2-dimensional systolic based architectures. Together with tree architectures and our proposed bit-serial architecture, our family of processors is able to cover a range of applications.

The bit-serial processor is able to support full search, three step search and diamond search. An early termination scheme has been introduced to further shorten the encoding time, and the standard technique is further optimized via H.264/AVC motion vector prediction.

In this thesis, the first reported MSB-first bit-serial variable block size motion estimation processor is introduced. It operates at a maximum clock frequency of 420 MHz. This processor is capable of performing common intermediate format (CIF) resolution full search encoding in real time at 23068 macroblocks per second within a -16 to 15 search range and occupies 2133 slices on a Xilinx Virtex-II Pro FPGA.

Our architectures were implemented on an FPGA platform and comparisons made. The result implementations are able to support H.264/AVC variable block size motion estimation for resolutions from CIF to high-definition television (HDTV) in real time.

摘要

近年，多媒體在算法上及技術上經歷了很大的發展。視訊編碼是一個重要的基本技術以及在近年來在壓縮效能及複雜性上有明顯的進步。視訊編碼的應用已從以前的視訊轉換器演變到現在的網際網路傳送和流動通訊。

H.264/AVC是最新的視訊編碼標準在位移預測應用了變塊尺寸、1/4 像素精度、移動向量預測以及多參考幀。這些新引進的特徵導致比以前的視訊編碼標準有更大的計算效能要求。在這論文中，我們提出了一個在現場可程式化邏輯閘陣列平台上建造位移預測處理器系列去平衡效能、面積、頻寬和功率消耗量間的取捨。

我們在運動估計中結合了算法及算術的優化。在算法優化層面上，我們比較了不同算法以及分析了它們的複雜性。在算術層面上，我們探索了應用冗餘數字系統和不冗餘數字系統的並行位運算和序列位運算。在我們的序列位運算設計中，我們在最不重要位元爲先地做法和最重要位元爲先地做法間作出了取捨。

最後，我們提供了一個能處理不同應用的位移預測處理器系列。在序列位運算處理器中，我們提供了應用一次元、二次元心縮陣列的結構。連同樹狀結構及我們提出的序列位運算結構，我們的處理器系列能夠處理一系列的應用。

序列位運算處理器能夠支持全域搜尋、三步搜尋及鑽石搜尋。提早結束技巧被提出去縮短編碼時間，以及經由 H.264/AVC 移動向量預測而被優化。

這篇論文發表了第一個最重要位元爲先地做法序列位運算變塊尺寸位移預測處理器。它能夠在在 400MHz 下運作。這處理器能夠在 CIF 尺寸下進行以-16 至 15 搜尋區域每秒 23068 宏塊的即時全域搜尋編碼以及使用了 Xilinx Virtex-II Pro 現場可程式化邏輯閘陣列平台 2133 塊硬體資源。

我們在現場可程式化邏輯閘陣列平台上建設了硬體結構以及作出了比較。我們的處理器系列實踐能夠支持即時的 H.264/AVC 變塊尺寸運動估計於 CIF 至 HDTV 解像度。

Acknowledgement

Firstly, I would like to give a sincere thank to my supervisor, Professor Philip Leong, who encouraged and challenged me throughout my research program, guided me on writing conference papers, thesis and put effort on our workings. Without his effort, this dissertation could not be written. Since the final year project, Professor Leong gave me vast amount of resources to grow me up as an equipped research student.

Secondly, I would like to thank Professor Lee Kin Hong, who offers me an opportunity to teach Computer Architecture which helps me a lot in my research. At the same time my presentation performance and teaching skills are improved significantly.

Moreover, I would like to thank my Mphil classmate, Mr. Lau Wai Shing, Mr. Wong Chun Kit, Mr. Cheung Yu Hoi Ocean and Mr. Lam Yuet Ming for their aid in my daily research problems.

Finally, I would like to thank my dearest family and Elaine for giving me fully support on daily cares, financial subsidies and a cheerful life from time to time.

This work is dedicated to my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Digital video coding has gradually increased in importance since the 90's when MPEG-1 [16] first emerged. It has had large impact on video delivery, storage and presentation. Compared to analog video, video coding achieves higher data compression rates without significant loss of subjective picture quality. This eliminates the need of high bandwidth as required in analog video delivery. With this important characteristic, many application areas have emerged. For example, set-top box video playback using compact disk, video conferencing over IP networks, P2P video delivery, mobile TV broadcasting, etc. The specialized nature of video applications has led to the development of video processing systems having different size, quality, performance, power consumption and cost.

Digitization of video scenes was an inevitable step since it has many advantages over analog video. Digital video is virtually immune to noise, easier to transmit and is able to provide a more interactive interface to users. Furthermore, the amount of video content, e.g. TV content, can be made larger through improved video compression because the bandwidth required for analog delivery can be used for more channels in a digital video delivery system. With today's sophisticated video compression systems, end users can also stream video, edit video and share video with friends via the internet or IP networks. In contrast, analog signals are difficult to manipulate and transmit.

Generally speaking, video compression is a technology for transforming video signals that aims to retain original quality under a number of constraints, e.g. storage constraint, time delay constraint or computation power constraint. It takes advantage of data redundancy between successive frames to reduce the storage requirement by applying computational resources. The design of data compression systems normally involves a tradeoff between quality, speed, resource utilization and power consumption.

In a video scene, data redundancy arises from spatial, temporal and statistical correlation between frames. These correlations are processed separately because of differences in their characteristics. Hybrid video coding architectures have been employed since the first generation of video coding standards, i.e. MPEG. MPEG consists of three main parts to reduce data redundancy from the three sources described above. Motion estimation and compensation are used to reduce temporal redundancy between successive frames in the time domain. Transform coding, also commonly used in image compression, is employed to reduce spatial dependency within a frame in the spatial domain. Lastly, entropy coding is used to reduce statistical redundancy over the residue and compression data. This is a lossless compression technique commonly used in file compression.

Hardware video compression systems can be implemented in application-specific integrated circuit (ASIC) and field programmable gate array (FPGA) technologies and, depending on the desired quality, real-time video encoding can be realized in both hardware and software technologies. Advances in codecs have continued since we have to enable video delivery over new mediums such as IP networks. As a result, H.264 [48] and MPEG-4 were developed to suit these network applications through enhanced compression efficiency and picture quality under very low bit-rates. Unfortunately, the complexity of the latest video codecs for network applications has increased a lot over previously defined standards such as MPEG-1 and MPEG-2 [17]. Real-time and low power encoding requirements create great challenges for software and hardware engineers.

1.1 Motivation

Among all the blocks in a video coder, motion estimation is the most demanding [8]. It is also the critical part that affects the video quality and compression efficiency. For this reason, many algorithms and architectures have been proposed to optimize this process. With advancement of video codec standards, the requirements of motion estimation have increased and thus both software (algorithmic) and hardware (architectural) optimizations must be continuously improved to cope with the increased complexity.

Power and speed are important considerations in codecs, especially running on mobile devices. Pure software implementations of video codecs usually result in large power consumption and low speed. To solve these issues hardware support is often employed. Hardware can use parallelism to obtain higher performance. Together this reduces the high data bandwidth and the instruction fetching associated with software, which are a major source of power wastage. As a result, hardware implementations consume lower power than a corresponding software implementation. Realization of video codec in hardware plays an important role in mobile applications.

Since the introduction of advanced motion estimation technique in the latest video codecs such as MPEG-4 and H.264, previous motion estimation architectures are no longer fully applicable. As a result, a family of new motion estimation architectures has been proposed to fit different application requirements while still efficiently utilizing resources.

1.2 The objectives of this thesis

Hardware assistance for video coding has become an important tool for optimizing system performance. Among hybrid video coding architectures, motion estimation introduces most of the complexity. Although many fast algorithms have been proposed to reduce the computational complexity, quality and compression performance still may not meet the application requirements. The complexity of motion estimation has greatly increased compared to MPEG-1 or MPEG-2 since the introduction of variable block size motion estimation in MPEG-4. Together with multi-reference frames, sub-pixel motion estimation support, prior architectures have become unsuitable.

In this thesis a family of motion estimation architectures and algorithms are studied and analyzed. Implementation of those architectures on an FPGA platform was effected to measure the efficiency of different architectural approaches. The VLSI architectures suggested in this thesis are also applicable to ASIC design. We also analyze the impact of computer arithmetic to motion estimation. With optimization of the arithmetic, we discovered efficient ways to implement motion estimation for low to high-end applications.

A full range of applications are supported by this work from low demand applications like low-resolution video conferencing and mobile digital video broadcasting, to high demand applications like HDTV video encoding. A family of motion estimation architectures is suggested to target different applications while efficiently utilizing computational resources, silicon area and power. Using the reconfigurable feature of FPGAs, we can provide the most efficient option to users which meets their requirement without changing the underlying hardware device.

In commercial design situations, the architecture selected is often suboptimal because of limited design time and resources. This work provides an overview of architectural alternatives to realize products in a more efficient manner. In terms of computer architecture and computer arithmetic, this thesis provides an architecture design space to explore optimizations for different kinds of application domains. The idea suggested in this work can also be applied to other areas such as transform or filtering in video encoder. With such implementation information, good tradeoffs between architecture and algorithm can be done to deliver a design with satisfactory performance, power and occupied silicon area.

1.3 Contributions

This thesis presents a family of FPGA-based motion estimation architectures for variable block size motion estimations. The following novel contributions result from this work.

- A study of the complexity, quality and performance associated with different motion estimation algorithms jointly with hardware architectures for H.264/AVC was made. Implementations of hardware architectures were done on a Xilinx FPGA platform. To the best of my knowledge, no study considering all these issues has been previously reported in the literature.

- Employing computer arithmetic optimizations for motion estimation. Although MSB-first bit serial architectures with early termination have been proposed [46], this is the first reported architecture supporting variable block size motion estimation.

- The initialization scheme proposed in section 4.5.3 is an improvement over the standard one and enables earlier termination.

- A family of architectures capable of supporting the latest codec standard H.264/AVC was developed which can meet the requirements of different kinds of applications under different constraints, e.g. performance, area, bandwidth, quality and power consumption.

1.4 Thesis structure

The main theme of this thesis is to study different implementation methods for motion estimation in the latest and future video coding standards. The reference coding standard in this thesis is H.264/AVC or MPEG-4 part 10. All the work stated in this thesis assumes the definition given in H.264/AVC standard.

Chapter 2 reviews the background of digital video compression and explains the role of motion estimation and the underlying algorithms. A number of commonly used motion estimation algorithms are introduced in detail. In terms of efficiency, performance and data dependency, a comparison between different algorithms are made. Lastly, the new added features in H.264/AVC are presented and their effects are analyzed.

Chapter 3 contains background associated with computer arithmetic for motion estimation. Number systems and corresponding algorithms for addition, absolute difference, multi-operand addition and comparison are presented using both bit-parallel and bit-serial (MSB-first or LSB-first) approaches.

Chapter 4 presents various VLSI architectures for variable block size motion estimation. Concepts including systolic arrays and tree architectures are described for bit-parallel systems. Next, the implementation of bit-serial system is presented in a top-down manner. Lastly, metrics of different architectures are analyzed and capabilities of these architectures are presented.

Chapter 5 presents the results of this thesis. Several comparisons have been made on different architectures and technologies. Lastly, conclusions are drawn in chapter 6.

Chapter 2

Digital video compression

2.1 Introduction

Compression is the process of compacting data into a smaller size in terms of number of bytes in digital media. Text files, pictures, voice, and in fact any data that contains redundancy can be made smaller by employing compression. Since an uncompressed video scene can occupy a large amount of storage space, video compression has an important role in the digital world.

Every compression system involves complementary units, an encoder (compression unit) and a decoder (decompression unit). The encoder exploits the redundancy among the given data and converts it to a compressed data stream. The decoder interprets the compressed data stream and restores it into the original format. To ensure the compressed size is satisfactory small, the original data may not be exactly the same as the original data hence some details may be lost. These kinds of compression systems are also called lossy compression systems.

The compression system must be well defined and the compressed data stream format known in both the encoder and decoder. The encode/decode pair is often described as a codec (coder/decoder). MPEG-1, MPEG-2, H.263 [3], H.264, etc are codecs defined by standardization parties. Standardization parties include moving picture expert group (MPEG), a group of the international organization for standardization; international telecommunication union (ITU-T); video coding expert group (VCEG), etc.

Video compression exploits three kinds of data redundancy within a video scene. In a standard lossy video compression such as H.264, the temporal redundancy between adjacent frames is the most important redundancy in video-type data and a large proportion of data dependency can be reduced through motion estimation. Spatial redundancy, which can be exploited within a frame, is reduced via transform techniques. It also constitutes a significant portion of redundancy since there is usually a high correlation between neighbouring pixels. Lastly, statistical redundancy, which must occur in any kind of data source, is reduced by an entropy coder in the last stage of the encoder.

2.2 Fundamentals of lossy video compression

In digital video, lossy compression is often employed to ensure good compression performance. Although the quality is inevitably degraded, there is minimal impact on perceived quality since only the high-frequency component is eliminated and human eyes are not sensitive to these components. The video encoding system consists of two kinds of compression units, namely lossy and lossless. Lossless data compression is a class of data compression algorithms in which the exact original data can be reconstructed. In contrast, lossy data compression must introduce some data loss during compression. The lossy compression unit contains temporal and spatial redundancy compressors. The statistical compressor in the last stage is a lossless one. The video coder contains five stages in total [42], handling different kinds of compression. In the following subsection each compression unit is discussed. Figure 2.1 shows the block diagram of a hybrid video encoder with the three compressor types described [42].

2.2.1 Video compression and human visual systems

A video scene consists of multiple objects each with their shapes, textures, illuminations, colors, etc. The motion, color and brightness of a scene are interpreted by the human visual systems. Some aspects are less sensitive to human eyes than others. For example, human eyes can't detect blue as well as green. In dark environments, only black and white can be detected. Moreover, the image formed in the retina remains for 10 to 30 milli-seconds. As a result human eye cannot detect the difference between 60 fps and 120 fps video. Human vision cannot perceive very detailed objects, i.e. high resolution objects. The objective of video compression is to exploit these properties of the human visual system, maximizing compression efficiency while keeping the impact of objective quality loss to a minimum. Since the non-sensitive components are high frequency components, e.g. color, frame rate, their removal results in high compression ratios.

2.2.2 Representation of color

We need at least three basic colors: red, blue and green, to display a true color picture. As a result, each pixel consists of at least three distinct channels of information.

Common color representations methods include RGB and YUV [42]. RGB is a representation that includes three basic colors together with brightness information. It is a common method for monitor displays. YUV is a representation which divides the color space into luminance (brightness) and chrominance (color). The 3 original colors can be derived from YUV data. Since humans are less sensitive to color than luminance, color information can be suppressed compared to luminance data without significant quality loss. For example, in 4:2:0 YUV, we have four times the luminance information than chrominance. Other representations such as 4:2:2 and 4:4:4 depend on the sampling frequencies. In the video compression, YUV representation is a better method to specify a pixel since it more closely tracks human perception and can enhance compression efficiency.

2.2.3 Sampling methods - frames and fields

A video can be sampled as frames or fields. This is called progressive sampling and interlaced sampling respectively. For each time instance, frames consist of both odd-numbered lines and even-numbered lines forming a picture. Fields consist of either odd-numbered lines or even-numbered lines consecutively. As a field occupies half the data of a frame, fields can be sampled at twice the rate of a frame. The advantage of interlaced sampling is it gives the viewer smoother motion, at the same time the data rate is reduced. The downside is the introduction of interlacing artifacts when displayed on a progressive scan monitor. Both sampling methods are employed in video compression and the best choice depends on the application.

2.2.4 Compression methods

Since a video scene is rectangular, block-based coding is a suitable choice of processing element. However, other compression methods also exist. One example is wavelet coding which is employed in image encoding in JPEG2000 [19]. A three dimensional version of wavelet coding for video compression was suggested [15] which gives better visual quality but the computation complexity is much higher. Another method is called arbitrary-shaped coding which is employed in MPEG-4 [18]. This is based on different moving objects whose motion is combined to form

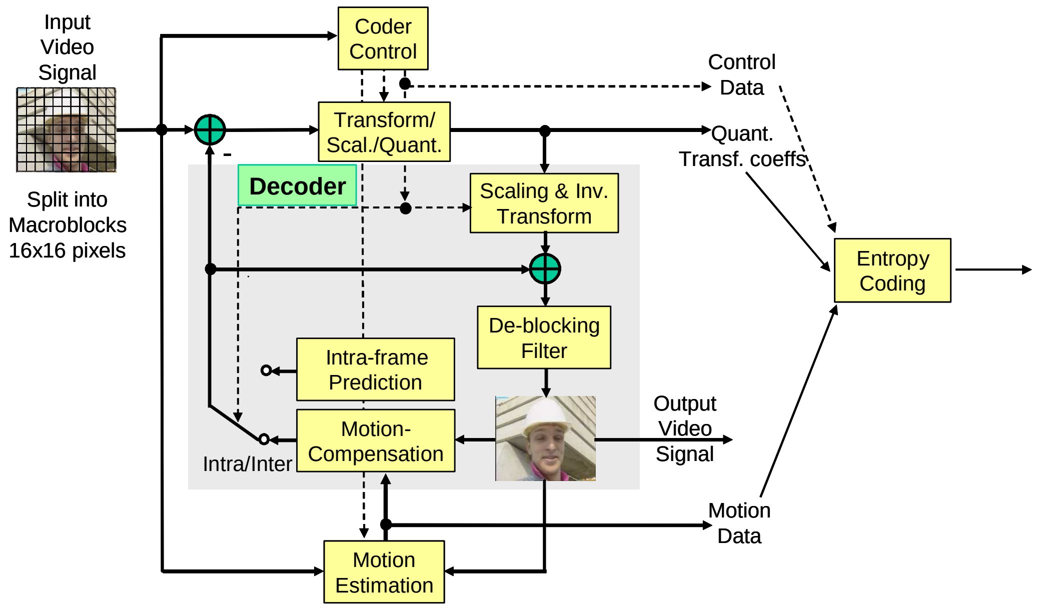

Figure 2.1: Hybird video coder for H.264/AVC

a complete frame. The performance is improved by accurate prediction in motion estimation. As a tradeoff, it has increased complexity.

2.2.5 Motion estimation

The motion estimation unit, shown in figure 2.1, is the first stage. The uncompressed video sequence input undergoes temporal redundancy reduction by exploiting similarities between neighbouring video frames. Temporal redundancy arises since the difference between two successive frames are usually similar, especially for high frame rates, because the objects in the scene can only make small displacements. With motion estimation, the difference between successive frames can be made smaller since they are more similar. Compression is achieved by predicting the next frame relative to the original frame. The predicted data are the residue between the current and reference pictures, and a set of motion vectors which represent the predicted motion direction. The process of finding the motion vector is optimal or suboptimal depending on the block matching algorithm chosen. Since the correlation between successive frames is inherently very high, the compression in this stage has large impact on the overall performance of the whole system. The motion predicted frames are usually called P-frames (Predicted frames). The other type of predicted frame is called B-frames (Bi-predicted frames). In this case the frame is predicted from two or more reference frames previously decoded.

2.2.6 Motion compensation

The motion compensation unit constructs a compensated frame, also called a residue frame, from the original frame and motion vectors. It than calculates the residue between the compensated frame and the reference frame. It is often employed together with motion estimation unit to reduce temporal redundancy of a video sequence. On the decoder side, the compensation unit acts as a reconstruction engine that combines the residue and motion vector to form the original frame. The frame is divided into subblocks so that the engine will act on each subblock sequentially until the whole picture is constructed.

2.2.7 Transform

The transform unit [32] reduces spatial redundancy within a picture. Its input is the residue picture calculated by the motion estimation unit. Since the residue picture has high correlation between neighbouring pixels, the transformed data is easier to compress than the original residue data since the energy of the transformed data is localized. The transformed data are called transform coefficients and they are passed to the quantization unit. The transformation can be done by many methods, including the Cosine Transform, Integer Transform, Karhunen-Loeve

Transform etc. Details of these can be found in [41].

2.2.8 Quantization

The quantization unit is the only lossy compression unit in the system. It serves to eliminate high frequency transform coefficients so that the quantized transform coefficients are more easily compressed. The elimination of high frequencies is justified because of the insensitivity of human vision to high frequency components. Subjectively, the quality of a video scene after quantization will not be significantly degraded if the bit-rate is not highly constrained. In each video coding standard, there exists a defined set of quantization parameters for providing the best compression-to-quality ratios for different applications.

2.2.9 Entropy Encoding

The entropy encoding unit is the last stage in a video compression system. In this stage, mostly statistical redundancy remains in the data. The motion vectors output by the motion estimation unit and quantized transform coefficients from the transform unit are accepted in this stage to produce the compressed bit stream that can be transmitted or stored. Typically, there are two kinds of entropy coder. The first is a variable-length-coder [4] in which the statistical information is initially defined. Second is an arithmetic coder [33] in which the statistical information is determined online. Most modern entropy coders are content-adaptive [33]. The compressed data can be optimized adaptively independent of the nature of the video scene.

2.2.10 Intra-prediction unit

The intra-prediction unit is activated when the difference between consecutive frames is too large, as occurs in a scene change

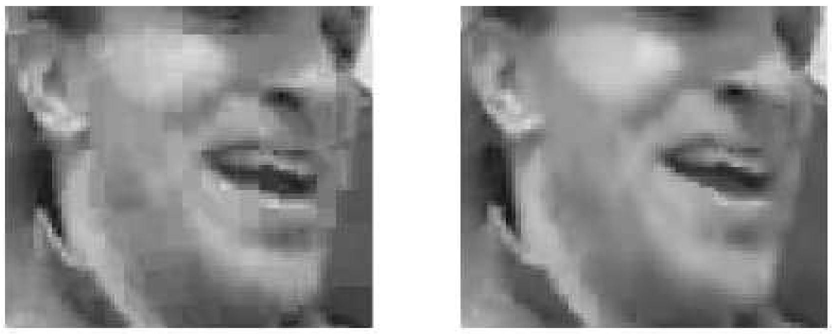

Figure 2.2: Improvement made by deblocking filter - Left: improved

or very fast moving pictures. In this case the frame is predicted by predefined block patterns instead of motion estimation and compensation. The output bit stream is usually smaller when such effects occur. H.264/AVC supports 13 prediction patterns in its intra-prediction unit [42]. The frames coded by intra-prediction unit are usually called I-frames (Intra-predicted frames).

2.2.11 Deblocking filter

Since most of the video coder employs block-based motion compensation, blocking artifacts may be visible when the scene is reconstructed. The lower the bit-rate, the more pronounced the blocking effect. To reduce this blocking artifact, a deblocking filter is included within the encoding loop. It employs adaptive filtering techniques so that edges are correctly filtered. Including the deblocking filter improves the visual quality in terms of objective (PSNR, section 2.3.5) and subjective judgment (human vision). The effect of the deblocking filter is illustrated in figure 2.2

Compression stage	Proportion
Motion vector search	67.31%
Mode selection	8.19%
Rate distortion opt.	3.37%
Transform and quantization	6.95%
entropy coder	6.19%
deblocking filter	0.03%
Others	7.96%

Table 2.1: Complexity profile of each compression stage in H264/AVC

2.2.12 Complexity analysis of on different compression stages

Each unit in the video coder contributes additional computational complexity to the overall system. Among all compression units, the motion estimation unit occupies most of the computation resources. In a software implementation, this is more than 65 percent of the total computation time. The transform unit, entropy coder and deblocking filter add up to 15 percent. The remaining 20 percent is due to mode selection and other overheads. Thus, there is no doubt that motion estimation unit can be accelerated using hardware. Table 2.1 shows the profiling of H.264/AVC encoding on a Pentium-III platform by [8].

2.3 Motion estimation process

2.3.1 Block-based matching method

Block-based matching method is the most widely used motion estimation method for video coding since pictures are normally rectangular in shape and block-division can be easily done. Usually, standards bodies, e.g. MPEG, defines the standard block sizes for motion estimation. This can be 16 by 16, 8 by 8, etc,

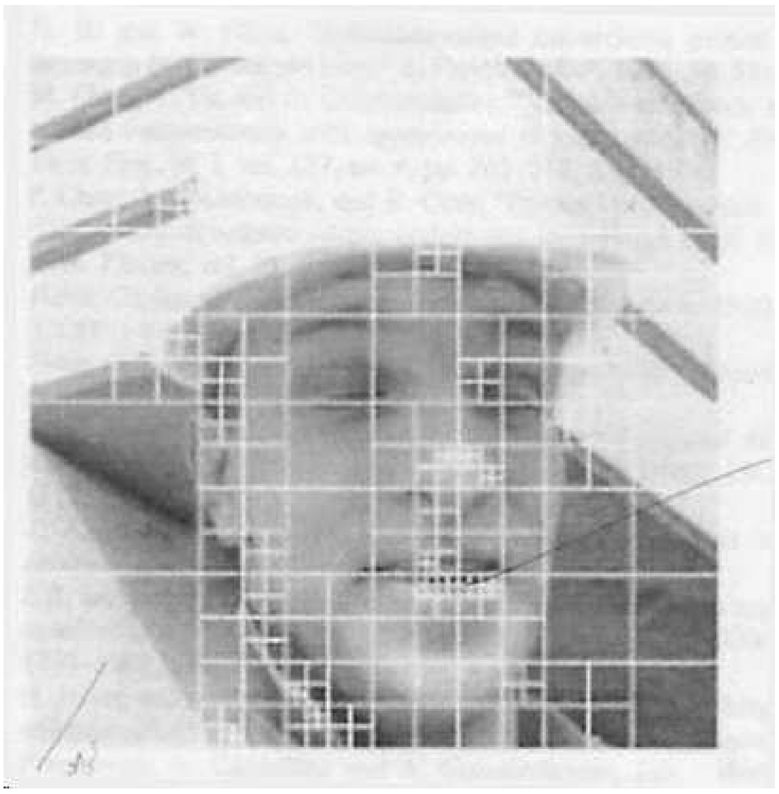

Figure 2.3: Selection of block sizes within a frame

depending on the target application of the video codec. In the latest codec standards such as MPEG-4 or H.264/AVC, variable block sizes are supported which can be 4 by 4, 8 by 8 and 16 by 16. The goal of motion estimation is to predict the next frame from the current frame by associating the motion vector to picture macroblocks as accurately as possible. The block size determines the quality of prediction [12] and thus the accuracy. Figure 2.3 shows the distribution of block sizes within a picture. It is easy to see that the detailed region is associated with small blocks whereas the large uniform region is associated with large blocks.

2.3.2 Motion estimation procedure

After motion estimation, a picture residue and a set of motion vectors are produced. The following procedure is executed for each block (16x16, 8x8 or 4x4) in the current frame.

1. For the reference frame, a search area is defined for each block in the current frame. The search area is typically sized at 2 to 3 times the macroblock size (16x16). Using the fact that the motion between consecutive frames is statistically small, the search range is confined to this area. After the search process, a "'best"' match will be found within the area. The "'best"' matching usually means having lowest energy in the sum of residual formed by subtracting the candidate block in search region from the current block located in current frame. The process of finding best match block by block is called block-based motion estimation.

2. When the best match is found, the motion vectors and residues between the current block and reference block are computed. The process of getting the residues and motion vectors is known as motion compensation.

3. The residues and motion vectors of best match are encoded by the transform unit and entropy unit and transmitted to the decoder side.

4. At decoder side, the process is reversed to reconstruct the original picture.

Figure 2.4 shows an illustration of the above procedure. In modern video coding standards, the reference frame can be a previous frame, a future frame or a combination of two or more previously coded frames. The number of reference frames needed depends on the required accuracy. The more reference frames referenced by current block, the more accurate the prediction is.

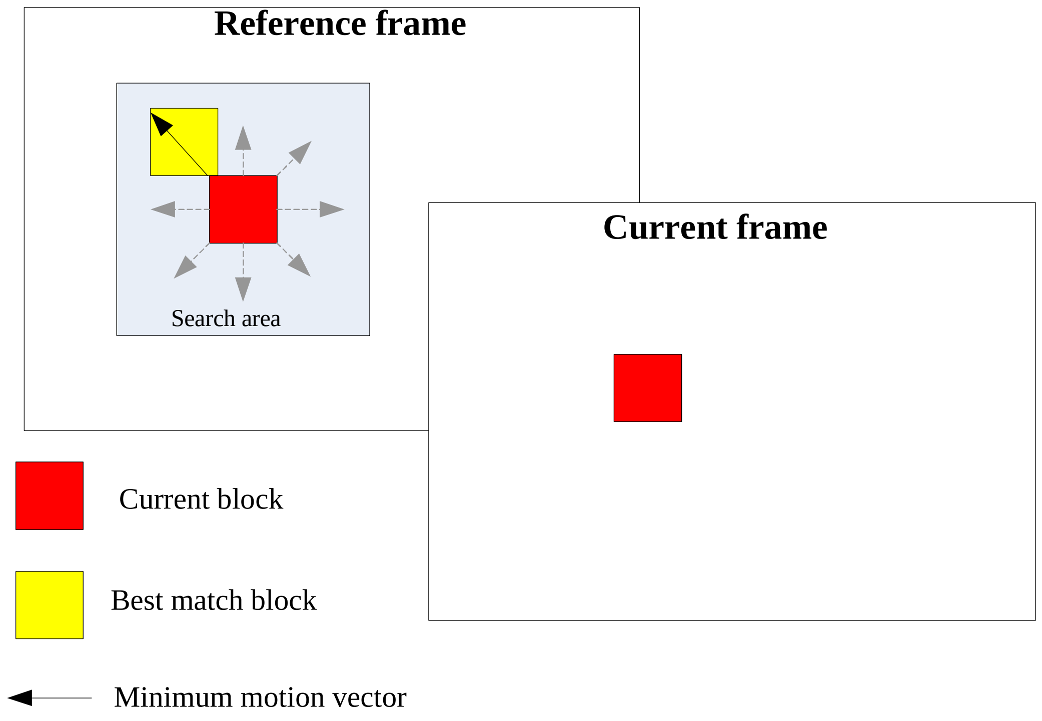

Figure 2.4: Motion estimation and motion vector

2.3.3 Matching Criteria

Block-based motion estimation obtains the best match by minimizing a cost function. Various cost functions have been proposed and analyzed in the literatures, varying in complexity and efficiency. In this section, the mean absolute difference (MAD), mean square error (MSE), sum of absolute difference (SAD) and sum of absolute transformed difference (SATD) block matching criteria are explained. I_n and I_{n-1} in the formula below represent the macroblock in current and reference frame respectively. m and n are the search location motion vector and N is the block size. k and l represent the index of macroblocks.

Mean absolute difference (MAD)

The MAD cost function [43] is defined as:

$$MAD(k,l;m,n)=\frac{1}{N^2}\sum_{i=0}^{N}\sum_{i=0}^{N}|I_n(k+i,l+j)-I_{n-1}(k+i+m,l+j+n)| \tag{2.1}$$

The advantage of MAD cost function is its simplicity and ease of implementation in hardware. Unfortunately, MAD tends to overemphasize small differences, giving an inferior result to MSE.

Mean absolute error (MSE)

MSE [47] is a cost function measuring the energy remaining in the difference block. The MSE cost function is defined as:

$$MSE(k,l;m,n)=\frac{1}{N^2}\sum_{i=0}^{N}\sum_{i=0}^{N}(I_n(k+i,l+j)-I_{n-1}(k+i+m,l+j+n))^2 \tag{2.2}$$

The advantage of MAD cost function is its accuracy but its complexity is high for both software and hardware implementations.

Sum of absolute difference (SAD)

SAD [43] is the most common matching criteria chosen in video coding because of its low complexity, good performance and ease of hardware implementation. The SAD cost function is defined as:

$$SAD(k,l;m,n)=\sum_{i=0}^{N}\sum_{j=0}^{N}|I_n(k+i,l+j)-I_{n-1}(k+i+m,l+j+n)| \tag{2.3}$$

The only difference between SAD and MAD is that SAD takes the sum of all pixels while MAD measures the average pixel value. Since the block size is constant during subtraction, the average value per pixel is not important. A divide operation is saved and the overall computation is simplified.

Sum of absolute transformed difference (SATD)

SATD is another way to compute the residues between two blocks, where the pixel values are pre-transformed by Hadamard Transform [42]. Since the transformed coefficient is closer to the final bit stream, it offers better matching accuracy than SAD. The SATD cost function is defined as:

$$SATD(k,l;m,n)=\sum_{i=0}^{N}\sum_{j=0}^{N}|T_n(k+i,l+j)-T_{n-1}(k+i+m,l+j+n)| \tag{2.4}$$

In the above equation, T_n and T_{n-1} represents the transformed coefficient of each block in the current and reference frames respectively. Although it offers significant improvement in prediction quality, transform hardware must be added within the motion estimation loop and hardware complexity is increased. On the other hand, the latency and thus the performance of motion estimation will be degraded due to the added transform hardware. This technique is applied to H.264/AVC when performing quarter pixel accuracy motion estimation. The high accuracy is needed since it is the final stage of motion estimation [42].

2.3.4 Motion vectors

To represent the motion of each block, a motion vector is defined as the relative displacement between the current candidate block and the best matching block within the search window in the reference frame. It is a directional pair representing the displacement in horizontal (x-axis) direction and vertical (y-axis direction). The maximum value of motion vector is determined by the search range. The larger the search range, the more bits needed to code the motion vector. Designers need to make tradeoffs between these two conflicting parameters. The motion vector is illustrated in figure 2.4.

Traditionally one motion vector is produced from each macroblock in the frame. MPEG-1 and MPEG-2 employ this property. Since the introduction of variable block size motion estimation in MPEG-4 and H.264/AVC, one macroblock can produce more than one motion vector due to the existence of different kinds of subblocks. In H.264, 41 motion vectors should be produced [42] in one macroblock and they are passed to rate-distortion optimization to choose the best combination. This is known as mode selection.

2.3.5 Quality judgment

The quality of a video scene can be determined using both objective and subjective approaches. The most widely used objective measure is the peak-signal-to-noise-ratio (PSNR) [42] which is defined as:

$$PSNR = 10log_{10}\Big[\frac{255^2}{MSE}\Big] \tag{2.5}$$

where the MSE is the mean square error of the decoded frame and the original frame (refer to section 2.3.3 for the exact formula). The peak value is 255 since the pixel value is 8 bits in size.

The higher the PSNR, the higher the quality of the encoding. The PSNR and bit-rate are usually conflicting, the most appropriate point being determined by the application. Although PSNR can objectively represent the quality of coding, it does not equal the subjective quality. Subjective quality is determined by a number of human testers and a conclusion is drawn based on their opinions. There exist cases for which high PSNR results in low subjective quality [42]. However, in most cases, PSNR provides a good approximation to the subjective measure and we use this measure in the rest of the thesis.

2.4 Block-based matching algorithms for motion estimation

Block-based matching algorithms are processes for finding minimum motion vectors in the motion estimation process. Different kinds of algorithm could give a different motion vector. Among all algorithms proposed, only full search gives optimal result within the search range. Other algorithms will give near-to-optimal results but significant lower complexity by reducing the number of search points. In this section, we describe several block-matching algorithms which are commonly used in modern coding standards.

2.4.1 Full search (FS)

The full search algorithm finds a global optimal motion vector from the entire candidate blocks within the search window. If our search window is 48 by 48 pixels and the block size is 16 by 16, there will be $16 * 16 = 1024$ candidates that need to undergo SAD computation. FS exhaustively searches all candidates until a minimum motion vector is found and it is the algorithm with the simplest data flow and control. It is suitable for hardware implementation since the high computational complexity can be overcome by parallelism and high bandwidth can be overcome by systolic architectures.

For software implementations, full search is often ignored [8] as contemporary microprocessors cannot handle full search with acceptable performance, especially for real time applications.

The number of search locations to be examined by full search is directly proportional to the square of the search range r. The number of search location in search area is $(2r + 1)^2$. So the algorithm complexity of full search is $O(r^2)$.

2.4.2 Three-step search (TSS)

The three step search algorithm has been proposed by Koga [23] and implemented by Lee et al [27]. This algorithm makes an assumption that the residue values increases radically from the absolute minimum point within the search area. This algorithm searches for the direction of the greatest decrease in residue and from there, continues to find the minimum point.

In the first step, TSS compares the nine search points surrounding the center point with step size p equal to or larger than half of the maximum search range r. Among the 9 search points, a minimum is selected and becomes the center of the next step. Next, the step size is halved and 8 new search points (excluding the center) are searched and again a minimum is selected. The step size is halved again and search continues until the step size is equal to one. The minimum search point is found at this stage.

From the above described procedure, it can be observed that the TSS constantly divides the search step size by two and is therefore a logarithmic search. The total number of search points is $[1 + 8log(p)]$. Except for the first step, the 8 search points are calculated in each iteration and the algorithmic complexity for TSS is $O(8log(p)) = O(log(r/2))$ where p is the initial step size and typically equals to $r/2$.

This algorithm has advantages of much lower complexity than FS in terms of number of candidates to evaluate and efficient implementations in both software and hardware. Even for software implementations, this algorithm [25] can offer real time encoding. It has the drawback of degraded quality since it can easily be trapped into a local minimum. Also, its large initial step size can lead to poor results. Furthermore, its data dependencies restrict parallelism in hardware implementations.

2.4.3 Two-dimensional Logarithmic Search Algorithm (2D-log search)

Two-dimensional logarithmic search is another logarithmic search proposed by Jain and Jain [21]. It has less search points than TSS but its prediction is more accurate. It also defines a step size at the beginning and terminates when the step size equals to one.

First the algorithm begins by calculating the SAD in the center of search area and another four points $\pm p$ pixels away from center in the horizontal and vertical directions. If the minimum SAD is located at the center, the step size is halved. Otherwise, one of the 4 search points will become the center and another 4 search points $\pm p$ pixels away from the new center will be calculated. In this case the step size is kept until the minimum SAD is located at the center. When the step size is reduced to 1, instead of four search points, the nine search points surrounding the current center are searched and a minimum point is found.

The complexity of 2D-log search is similar to TSS and equals to $O(log(r/2))$. The difference is that the Big-O constant is lower than that of TSS.

This algorithm has advantages of better prediction quality than TSS with some additional control overhead introduced. Again, its data dependencies do not favor fully parallel hardware implementations.

2.4.4 Diamond Search (DS)

The diamond search algorithm was proposed by Zhu and Ma in 1997 [56, 57] and, as its name suggests, employs diamond search patterns. It is commonly employed for fast searches since it provides the best quality to complexity ratio. It makes use of two diamond shapes for searching: a large diamond with 9 search points and a small diamond shape with 5 points.

This algorithm initially employs a large diamond shape in the beginning and searches for the minimum SAD location. If the minimum location is not at the center, the new center is reset to the minimum location just found and the search is continue using a large diamond pattern until a minimum value is located at the center. Then the algorithm switches to small diamond pattern and the minimum point is found in this final stage.

The complexity of this algorithm is lower than the entire logarithmic search described above. Its complexity is in order of $O(log(r/2))$. The average search points in DS is less than 20 for a normal scene [49].

DS gives better prediction quality and lower complexity than TSS and 2D-log search. It is the best choice for software implementations. For hardware implementation, the two diamond sizes add a small but acceptable complexity to control circuits.

2.4.5 Fast full search (FFS)

In the full search algorithm, the SAD is computed for all candidate positions. When a smaller value is found, it is recorded as the current minimum SAD. It is possible to speed up the rejection of incorrect candidates via mathematical techniques such as Successive Elimination Algorithms (SEA) [29], progressive norm successive algorithm (PNSA)[37] without any quality loss. Partial distortion elimination (PDE) [13] is a method of early comparison that compares the partial SAD with the current minimum SAD. The speedup is possible because these techniques employ other matching criteria, e.g. SEA, PNSA and partial SAD, which are easier to calculate than SAD and early elimination for impossible candidates can be achieved.

The algorithms suggested above focus on mathematical optimizations. In hardware implementations, arithmetic optimizations can also improve the efficiency of full search. For example, in a bit serial implementation of SAD, computation can be saved by employing early termination in the comparison stage. The early termination is brought by the fact that comparison is a most-significant-bit-first process. In the next chapter we will explain the early termination technique in detail.

2.5 Complexity analysis of motion estimation

In video compression applications, we define the complexity of an algorithm in terms of the number of required operations and express the complexity as MOPS (Million operations per second). In the following subsection we will compare the complexity of motion estimation algorithm in terms of MOPS. The following assumptions have been made for the comparisons.

1. The macroblock size is 16 by 16.
2. The SAD cost function requires $2*(16*16)$ data loads, $16*16 = 256$ subtraction operations, 256 absolute operations, 256 accumulate operations, 1 compare operation and 1 data store operation. In total, 1282 operations are needed for one SAD computation.
3. CIF resolution is $352 * 288$ and HDTV 720p resolution is $1280 * 720$. The number of macroblocks in a CIF frame is 396 and for HDTV 720p is 3600.
4. The frame rate is 30 frames per second.
5. The total number of operations required to encode CIF video in real time is $1282 * 396 * 30 * (\#of searchpoints)=$ $15.23016 * (\#of searchpoints)$ MOPS. To encode HDTV 720P video signal, $138.456 * (\#of searchpoints)$ MOPS is needed.

Algorithm	Num of searching point when search range = ±16	CIF	HDTV
FS	1024	15600	142000
TSS	33	502	4600
2D-log Search	30	456	4200
DS	25	381	3500

Table 2.2: Complexity of block-based searching algorithms (measured in MOPS)

2.5.1 Different searching algorithms

Assume that the search range is confined to ±16. Table 2.2 shows the number of search points needed for each of the algorithms described above. The last 2 columns show the number of operations needed for CIF and HDTV 720P resolutions respectively. The large computational requirements limit the ability of general purpose processors to perform these searches in real time. In modern general purpose processors such as Intel Pentium-4, its performance is a few giga operations per second (GOPS) and a full search is totally impractical in software. Digital signal processors, ASIC or FPGA technologies are the only choices available for FS.

2.5.2 Fixed-block size motion estimation

In the first generation coding standards, the block size is confined to 8 by 8 or 16 by 16. A large block size favors encoding of a uniform area whereas small block sizes favor detailed area encoding [12]. Within a picture, detailed uniform areas coexist and fixed block sizes must sacrifice prediction quality to reduce complexity.

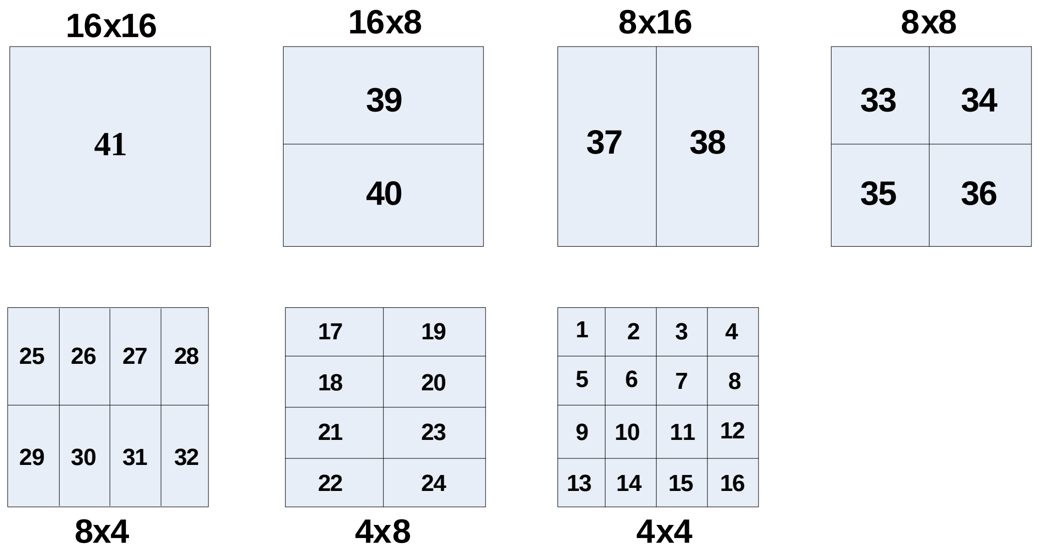

Figure 2.5: Sub-macroblock partitions in H.264/AVC

2.5.3 Variable block size motion estimation

In order to adaptively select a suitable block size for picture macroblock, variable block size motion estimation has been added in the latest codec standards, e.g. H.264. In H.264 [48], each picture (frame) is segmented into macroblocks. Each macroblock is further divided into sub-blocks with 7 different types of block sizes (4x4, 4x8, 8x4, 8x8, 8x16, 16x8 and 16x16) as shown in Figure 2.5. Each macroblock has in total 41 types of sub-blocks to cover the whole macroblock. In variable block size motion estimation, for each type of subblock, a motion vector is produced. In total 41 motion vectors are calculated per macroblock.

Since the number of motion vector is increased from 1 to 41 in variable block size motion estimation, the number of comparison operations in the computation of a SAD is also increased. The number of operations to find the motion vectors is 1282+40=1322 operations. In a software implementation this is not a big increase but for hardware implementation, this increase is significant as the number of comparators is increased from 1 to 41, contributing a significant hardware cost.

2.5.4 Sub-pixel motion estimation

Sub-pixel motion estimation involves searching sub-sample interpolated positions as well as integer-sample positions. Since the motion of a macroblock between two successive frames can be half way between two integer positions in most cases, the addition search on sub-pixel accuracy block improves the overall quality of the prediction. In H.264/AVC [48], quarter-pixel accuracy motion estimation is supported in addition to half-pixel accuracy.

In the first stage, motion estimation finds the best match on the integer sample. The encoder search half-sample positions immediately next to this best match to see whether the match can be improved. If required, quarter-pixel samples are searched for further improvement. The added complexity is due to the interpolation of sub-pixels and added search positions. Assume interpolation for the half-pixel accuracy pixel needs 11 operations for each pixel and 3 operations are needed for quarter pixel accuracy. We have 9 more search points for half-pixel accuracy and 9 more for quarter pixels. We have in addition $9 * 256 * (11 + 3) = 32256$ filtering operations per macroblock and the number of search point is increased by 18. Figure 2.6 shows the integer position, half-pixel and quarter-pixel positions of search candidates.

2.5.5 Multi-reference frame motion estimation

In previous standards, for prediction of macroblocks, only references to the immediate previously coded I-picture or P-picture are required. In H.264/AVC, this restriction is released to enable efficient coding by allowing selection among a larger number of

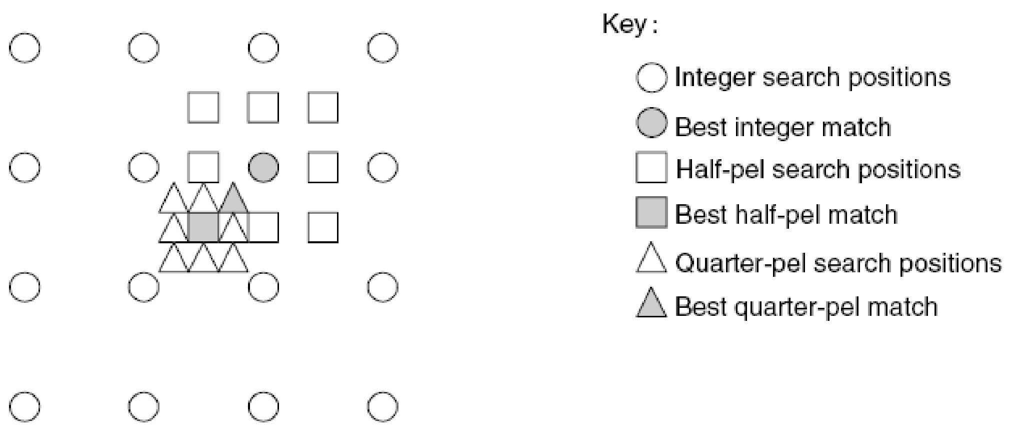

Figure 2.6: Integer, half-pixel and quarter-pixel motion estimation search positions (pel stands for pixel)

pictures that have been decoded and stored. Up to five reference frames can be used in H.264/AVC [48].

Motion estimation algorithms are modified accordingly in this case. Depending on the searching algorithm, the complexity can be increased by five times in the worst case.

2.6 Picture quality analysis

The performance of motion estimation algorithms: FS, TSS, 2D-log and DS are simulated in software using Matlab. The simulation result on the standard Foreman sequence is shown in figure 2.7. This video consists of a slow moving background and detail facial motions. As expected, full search performs the best among all algorithms. For fast search algorithms, DS performs better than TSS and 2D-log search. As a result, DS is a common choice in fast search motion estimation.

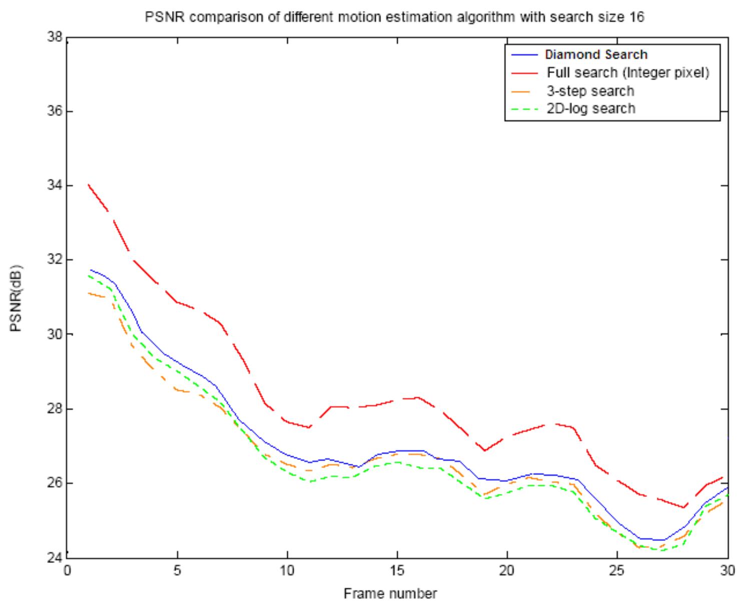

Figure 2.7: Matlab simulation on the quality of different motion estimation algorithms on Foreman

2.7 Summary

In this chapter we gave background information on video coding with an emphasis on motion estimation. The motion estimation procedure, matching criteria, quality judgment were presented. We also reviewed algorithms for performing motion vector prediction, comparing their complexities and qualities. Advanced motion estimation features such as fractional motion estimation, multi-reference fames were also discussed.

Chapter 3

Arithmetic for video encoding

3.1 Introduction

Computer arithmetic is an important area of digital computer organization concerned with the realization of arithmetic functions to support computer architectures as well as arithmetic algorithms for software implementation. Architectural and algorithmic optimizations are studied to maximize the efficiency of arithmetic operations.

In general purpose processor, efficient digital circuits for mathematical primitives such as $+, -, \times, \div, \sqrt{}, log, sin, cos$ are employed. Implementing a complex problem, e.g. motion estimation, using general purpose processor, the designer searches an appropriate algorithm and implements it in an acceptable complexity using combinations of mathematical primitives provided. The optimization is often restricted since the maximum parallelism cannot be exploited.

To address this issue, algorithm specific optimizations can be applied to greatly improve efficiency. In the following section we use motion estimation as an example to show how the arithmetic circuit can be optimized using arithmetic approach.

3.2 Number systems

We have been familiar with decimal numbers in calculation for thousand of years, whereas number representation for computer systems have only been developed this past century. In digital systems, numbers are encoded by means of binary digits. As the representation has significant effect on algorithm and circuit complexity, a suitable representation should be correctly chosen for any special applications if designers want to fully exploit the optimizations.

In this section we will introduce the non-redundant number systems such as the sign-and-magnitude and complement number schemes. For redundant number systems we introduce carry save and signed-digit number schemes. Other representations also exist, e.g. residue number system, logarithmic number system and floating point number system. Since they are not employed in our applications, interested readers can refer to an arithmetic textbook [10] [28] [40] for further details.

3.2.1 Non-redundant Number System

A number N can be represented by a string of n digits with r being the radix as follows.

$$N = (d_{n-1}d_{n-2}...d_1d_0)_r$$

d_i, where $0 \leq i \leq n-1$, is a digit and $d_i \in \{0, 1, ..., r-1\}$. It is a positional weighted system where the position of d_i matters. The value of N is defined by:

$$\begin{aligned} N &= d_{n-1} \times r^{n-1} + d_{n-2} \times r^{n-2} + ... + d_1 \times r^1 + d_0 \times r^0 \\ &= \sum_{i=0}^{n-1} d_i \times r_i \end{aligned}$$

d_{n-1} is called the most significant digit (MSD) and d_0 is called the least significant digit (LSD). In binary encoding system, each digit is called a bit and the above terms become MSB and LSB.

Sign-and-magnitude number representation

There are several ways to represent a negative number in a non-redundant number system. In standard mathematical notation, $\pm$ sign is appended to the front of the string of digits to indicate the number is positive or negative. In case of computer systems, a common convention to represent negative sign is appending a "1" to the MSD of binary string to represent a negative number and a "0" for a positive one. Such representation is called sign-and-magnitude representation scheme. As a result, the number of bits representing the number is increased by 1. A Sign-and-magnitude number S is presented as:

$$S = (d_n d_{n-1} d_{n-2} ... d_1 d_0)_r$$

where $d_n = 0$ when it is positive and $d_n = 1$ otherwise. The value of S is defined by:

$$\begin{aligned} S &= (-1)^{d_n} \times (d_{n-1} \times r^{n-1} + d_{n-2} \times r^{n-2} + ... + d_0 \times r^0) \\ &= (-1)^{d_n} \sum_{i=0}^{n-1} d_i \times r_i \end{aligned}$$

Advantages of sign-and-magnitude representation include its conceptual simplicity, symmetric range and simple negation by flipping the sign bit. A disadvantage is that the addition of number needs to be handled differently when signs of two numbers are different.

Complement representation

Another way to represent a signed number is to employ a complement number representation. The negation of a number X is represented as unsigned value $M - X$ where M is a suitably large constant greater than X. Representing integers in range $[-N, +P]$ requires $M \geq N + P + 1$ or $M = N + P + 1$ for maximum coding efficiency. For example to code $[-7, +8]$, $M = 7 + 8 + 1 = 16$. The value X of a r's complement number is defined as follows:

$$\begin{aligned} X &= -d_n \times r^n + d_{n-1} \times r^{n-1} + d_{n-2} \times r^{n-2} + ... + d_0 \times r^0 \\ &= -d_n \times r^n + \sum_{i=0}^{n-1} d_i \times r_i \end{aligned}$$

Addition and subtraction can be performed easily in hardware using complement forms since the adder and subtractor can be combined in a unit. For this reason, 2's complement scheme is often used in computer systems.

3.2.2 Redundant number system

The number systems introduced in the previous sections belong to non-redundant, positional weighted systems. Each digit has only a positive value which is less than radix r. In this section we present another number system, the redundant number system including the signed-digit number scheme and carry save number scheme. These number systems are redundant as a value can be represented more than one way. In the signed-digit number scheme, each digit can either have a positive or a negative value bounded by $\pm\alpha$. In carry-save number scheme, each digit is a positive value which is bounded by $2r - 1$.

Signed-Digit number

Signed-digit number is a redundant number representation satisfying the following constraints. Suppose we have a signed-digit number SD. It is represented by:

$$SD = (x_{n-1}x_{n-2}...x_1x_0)_r$$

where radix $r \geq 2$, $-\alpha \leq x_i \leq \alpha$ and $\left\lceil \frac{r-1}{2} \right\rceil \leq \alpha \leq r-1$.

To represent the signed-digit number in minimum redundancy [10], we set $\alpha = \left\lfloor \frac{r}{2} \right\rfloor$. To value of SD is defined as:

$$\text{value of SD} = \sum_{i=0}^{n-1} x_i \cdot r^i \text{ where } x_i \in \{-\alpha, ..., -1, 0, 1, ..., \alpha\}$$

.

Carry-save number

Carry save number is also a number system with redundancies. Suppose we have a carry save number CS, it is represented by:

$$CS = (x_{n-1}x_{n-2}...x_1x_0)_r$$

where radix $r \geq 2$, $0 \leq x_i \leq 2r-1$ since $carry \in \{0,1\}$. The value of a carry save number is defined as:

$$\text{value of CS} = \sum_{i=0}^{n-1} x_i \cdot r^i \text{ where } x_i \in \{0, 1, ..., 2r-1\}$$

.

In the redundant number system, addition time can be independent of word length. Since the length of maximum carry propagation can be reduced, for which the adder can work at higher efficiency. It favors operations for which both the input and output operands are in redundant representations.

The drawback is the increase in the number of bits required for representing a number. Moreover, it makes the magnitude comparison and sign detection more complex than that in non-redundant number systems. Finally, it takes time to convert redundant numbers to non-redundant one when non-redundant results are desired.

Application of redundant number systems

Applications of the redundant number system appear in many areas such as cryptography [36], digital signal processing [44], etc. These fields involve calculation of either very long operands or multi-operand operations which can benefit from redundant number representations. For example, RSA decryption involves 1024-bit arithmetic and is slow in non-redundant number systems. Redundant number systems can be used to reduce time complexity. Digital signal processing (DSP) operations often involves filtering which employs multi-operand operations. Multi-operand addition can be speeded up by employing redundant number systems. In these applications, since the performance speedup outweighs area overhead, redundant number systems are commonly chosen. For motion estimation, which involves many multi-operand operations, the advantages of redundant number systems such as low latency and high pipelinability are compelling.

3.3 Addition/subtraction algorithm

Addition and subtraction are basic operations in computer arithmetic and also in our problem domain - motion estimation. Many algorithms and architectures have been proposed in previous literatures to perform addition or subtraction in hardware. Both non-redundant and redundant approaches are applicable to these algorithms. Since the radix-2 is commonly used in computer system, it is assumed in all algorithms below.

3.3.1 Non-redundant number addition

This is the simplest addition algorithm in computer arithmetic. Suppose we have two n-bit operands, x and y, to be added, such that $0 \leq x, y \leq 2^n - 1$. Together with carry-in $c_{in} \in \{0, 1\}$ as inputs, output sum s is calculated where $0 \leq s \leq 2^n - 1$ and carry out $c_{out} \in \{0, 1\}$. We have,

$$x + y + c_{in} = 2^n c_{out} + s \tag{3.1}$$

The solution to this equation 3.1 is,

$$\begin{aligned} s &= (x + y + c_{in}) \bmod 2^n \\ \text{and} & \\ c_{out} &= \begin{cases} 1 & \text{if } (x + y + c_{in}) \geq 2^n \\ 0 & \text{otherwise} \end{cases} \\ &= \lfloor (x + y + c_{in})/2^n \rfloor \end{aligned}$$

When $n = 1$, the addition algorithm reduces to a primitive module called full-adder (FA) which is a fundamental element to build big adders in hardware. The recursion of the above equation for $n = 1$ gives implementation of word adder built on FA arrays.

3.3.2 Carry-save number addition

A carry save representation allows us to eliminate long carry propagations in a non-redundant number addition algorithm. Suppose we have three radix-2 n-bit non-redundant numbers x, y, z such that $0 \leq x, y, z \leq 2^n - 1$ as inputs and produces as outputs the sum vector sv where $0 \leq sv \leq 2^n - 1$ and the carry vector cv where $0 \leq cv \leq 2^{n+1} - 1$, we have:

$$x + y + z = cv + sv$$

where

$$\begin{aligned} sv_i &= x_i + y_i + z_i \bmod 2 \\ cv_{i+1} &= \lfloor (x_i + y_i + z_i)/2 \rfloor \\ cv_0 &= 0 \\ 0 \leq & \; i \; \leq n-1 \end{aligned}$$

The sum of three numbers is in carry save format $\{cv, cs\}$. To convert it to a non-redundant number, a final non-redundant addition is needed.

3.3.3 Signed-digit number addition

The objective of signed-digit addition, like carry save addition, is to eliminate the carry propagation for long word length numbers. Suppose we have two signed-digit numbers, x and y. The procedure of performing signed-digit addition consists of two steps.

1. Compute the interim sum (w) and transfer (t) such that

$$x_i + y_i = w_i + rt_{i+1} \tag{3.2}$$

   at digit level $0 \leq i \leq n-1$ while n is the number of digit and r is the radix. w acts like the sum in the carry-save addition but instead of being values between 0 and r, it is bounded by $-\alpha \leq w \leq \beta$. t acts like a carry to the next position with $t \in \{-1, 0, 1\}$ instead of $0, 1$ in carry-save representation scheme.

2. Compute the sum s of w and t such that for $0 \leq i \leq n-1$,

$$s_i = w_i + t_i$$

The carry-free property can be ensured by avoiding carry generations of adding w and t. Given $-a \leq s \leq a$, the following constraints should be satisfied on values of w and t so that no carry is produced. The constraint is given below.

$$-a + t^- \leq w_i \leq a - t^+$$

where

$$-t^- \leq t_{i+1} \leq t^+$$

To implement a radix-2 signed-digit addition, this constraint is never met because if the radix is 2, $a = 1$ and $-1 + t^- \leq w_i \leq 1 - t^+$ implies either $w_i = 0$ or $t_{i+1} = 0$ which is not possible to satisfy equation 3.2 [10].

To allow the signed-digit addition in radix-2 operands, a recoding modification is made [10]. As the signed-digit addition can be viewed as a recoding of digit set of $x_i + y_i \in \{-2, -1, 0, 1, 2\}$ into digit set of $\{-1, 0, 1\}$, two recodings are performed. First, the digit set is recoded from $\{-2, -1, 0, 1, 2\}$ to $\{-2, -1, 0, 1\}$.

$$x_i + y_i = 2h_{i+1} + z_i$$

such that $h_i \in \{0, 1\}$ and $z_i \in \{-2, -1, 0\}$. The sum of $2h_i$ and z_i has a digit set $\{-2, -1, 0, 1\}$. Then this digit set is recoded to $\{-1, 0, 1\}$ by setting

$$z_i + h_i = 2t_{i+1} + w_i$$

such that $t_i \in \{-1, 0\}$ and $w_i \in \{0, 1\}$.

Details and examples of radix-2 signed-digit addition are presented in [10]. In this work we employ a 2-level recoding approach to implement our signed-digit multi-operand addition in our bit-serial architecture in chapter 4.

3.4 Bit-serial algorithms

Bit-serial arithmetic [10] has the advantages of smaller silicon area, low pin count and higher operating frequency. Since the routing in bit-serial designs can be much shorter than corresponding bit parallel implementation, the throughput can be competitive to a bit-parallel approaches. Moreover, performance per gate in a bit-serial design is often much higher. It is often a good choice in implementation platforms where storage elements are abundant, as is the case for FPGAs.

Bit-serial implementations can be categorized into two processing orders: Least-significant-bit first (LSB-first) and Most significant bit first (MSB-first). MSB-first arithmetic is also called on-line arithmetic in the literature since it adds an on-line delay from input to output. Some operations like addition and subtraction are more easily implemented LSB-first while comparisons, square roots and divisions are more efficient MSB-first.

3.4.1 Least-significant-bit (LSB) first mode

LSB-first mode is an arithmetic computation starting from the least weighing digit. LSB-first arithmetic can be employed in non-redundant number systems or redundant number systems. With a word size of N bits, the number of iterations to produce the result is N. LSB-first arithmetic can produce the output bits without input-to-output delays when LSB-first favored algorithms are implemented. Once the input bits are ready, the output bits of corresponding inputs can be calculated immediately in the next computation cycle.

LSB-first favored algorithms include additions, subtractions, multiplications, etc. As a result LSB-first mode is applicable to almost all the simple primitive mathematical operations. For example, a LSB-first addition algorithm for radix-2 is shown in figure 3.1

```
Algorithm LsbBsAdd
Input: Operands $x$, $y$ and $carry_0$
Output: Sum $s$ and carry out $carry_N$
1.  for ($i$ from 1 to $N$)
2.      do $s_i = (x_i + y_i + carry_{i-1})$ mod 2;
3.         $carry_i = (x_i + y_i + carry_{i-1})$ >> $log_2$ 2;
4.  return $s$ and $carry_N$
```

Figure 3.1: LSB-first bit-serial addition algorithm

where $carry_0 = 0$ and the result is $N + 1$ bits including the carry out. This addition takes N cycles to complete.

3.4.2 Most-significant-bit (MSB) first mode

MSB-first arithmetic, also known as on-line arithmetic, is a bit-serial scheduling technique in which the calculation is started from the largest weighing digit [11]. Its idea is to perform computation overlapped with digit by digit communications of operands. Division and square root operations can be implemented efficiently using MSB-first calculation. An important characteristic of on-line arithmetic is that an on-line delay δ is injected such that digit j input will complete calculations at $j+\delta$ cycle.

Addition and subtraction must employ redundant number representation to be computed in a MSB-first manner. A commonly used redundant number set is signed-digit numbers. For example, a MSB-first addition algorithm for radix-r ($r > 2$) is shown in figure 3.2:

where $w_N = 0$ and $z_0 = 0$. The result is in signed-digit format and 1 cycle on-line delay is introduced. In contrast, two cycles on-line delay are introduced for radix-2 two operand additions [10].

```
Algorithm MsbBsAdd
Input: Operands $x$, $y$
Output: Sum $z$
1.  for ($i$ from N-1 down to -1)
2.      do $w_i = (x_i + y_i)$ mod r;
3.          $t_{i+1} = (x_i + y_i)$ >> $log_2$ r;
4.          $z_{i+1} = w_{i+1} + t_{i+1}$;
5.  return $z$
```

Figure 3.2: MSB-first bit-serial addition algorithm

3.5 Absolute difference algorithm

This section describes how the operation "Absolute difference" ($|a - b|$) can be done in bit-parallel and bit-serial approaches. The procedure of absolute difference is described as follows.

1. Subtraction of the two numbers a and b.
2. Check if the result is negative.
3. Convert the negative number to a positive result if the difference is negative.

The following subsection describes how to realize these three steps in non-redundant and redundant number systems.

3.5.1 Non-redundant algorithm for absolute difference

In the first step, a subtraction is done by converting b to its two's complement format and performing the addition. This involves an (N+1)-bit addition. The sign of the difference can be determined by looking at the most significant bit of the difference. Lastly, the result is converted to its 2's complement representation if it is negative. At the hardware level, the worse case is two (N+1)-bit additions and a sign-detection.

Non-redundant number system is commonly used in bit parallel absolute difference implementation since it can determine the sign in a short time. But for SAD in motion estimation, the absolute difference calculated is not necessary presented in non-redundant numbers since multi-operand additions in the subsequent stage can accept redundant inputs. As a result, the absolute difference is modified to:

1. Perform $\overline{A} + B$ where $\overline{A}$ is a bit-wise NOT on A.
2. Check if carry out $= 1$, if yes, $B \leq A$.
3. If carry out $= 1$, set $(\overline{A}, B)$ as output. Set $(A, \overline{B})$ as output.

In this way, result is represented by two N-bit pairs. The worse case is one N-bit addition and one comparison only (neglecting the cost of the invertor operation). This approach has been used in a SAD processor proposed in [53].

3.5.2 Redundant algorithm for absolute difference

When we perform absolute difference in a signed-digit number system, we have to redefine the comparison and absolute value operations. In motion estimation a and b in $|a - b|$ are 8-bit positive numbers representing the intensity of a pixel. The first step $a - b$ can be done by converting $a - b$ into signed-digit format directly.

In signed-digit number system, each significant digit is represented by $\{-1, 0, 1\}$, or $\{x^+, x^-\} \in \{(0,0), (0,1), (1,0), (1,1)\}$ in binary format. $\{(0,0), (1,1)\}$ is redundant and equal to zero value. As a result, $a-b$ can be represented in signed-digit format setting $(x_i^+ = a_i, x_i^- = b_i)$ where $0 \leq i \leq N-1$ and $N = 8$ in this case. Second, $a - b$ in signed-digit format is checked to see if it is a positive or a negative number. Finally, it is converted to positive signed-digit number as output. The conversion is

```
**Algorithm** *SignDectectSignNum*
**Input:** Current block pixel: $a$ Reference block pixel: $b$
**Output:** $|a - b|$ in signed-digit format: $x$
1.  $a\_larger = 0$;
2.  for ($i$ from N downto to 1)
3.      do $x_i = (a_i, b_i)$;
4.  for ($i$ from N downto to 1)
5.      do if ( $x_i = (0, 0)$ or $x_i = (1, 1)$ )
6.             then continue;
7.          if ( $x = (1, 0)$)
8.             then $a\_larger = 1$;
9.                  break;
10.         if ( $x = (0, 1)$)
11.            then $a\_larger = 0$;
12.                 break;
13. if ($a\_larger = 0$)
14.    then for (i from N to 1)
15.            do $x_i = (b_i, a_i)$;
16.    else for (i from N to 1)
17.            do $x_i = (a_i, b_i)$;
18. return $x$
```

Figure 3.3: Signed-digit number based sign detection algorithm

achieved by just interchanging x^+ with x^- if a negative signed-digit number is detected, i.e. $(x_i^+ = b_i, x_i^- = a_i)$. The sign checking for signed-digit numbers is described in figure 3.3. Notice that $N = 8$ is the word-length and the checking is performed in a MSB-first manner.

x is the result of $|a - b|$ in signed-digit format. In hardware, a signed-digit absolute difference requires no addition or subtraction. The sign detection can be done on-the-fly as it is a MSB-first favored algorithm and thus the hardware latency is not significant compared to a conventional two's complement approach.

```
**Algorithm** *SadSeq*
**Input:** Current block pixel: $c$, Reference block pixel: $r$
**Output:** Sum of absolute difference: $SAD$
1.  SAD[0] = 0;
2.  for ($i$ from 1 to $MN$)
3.      do SAD[i] = SAD[i-1] + $|c_i - r_i|$;
4.  return $SAD$
```

Figure 3.4: Sequential SAD computation in general purpose processor

3.6 Multi-operand addition algorithm

Multi-operand addition plays an important role in motion estimation as it comprises most of complexity in a SAD calculation. SAD operation is given by:

$$SAD = \sum_{i=0}^{MN} |c_i - r_i|$$

where M,N are the width and height of a macroblock respectively. The summation can be performed in parallel or sequentially. In general purpose processors, the summation is executed sequentially shown in figure 3.4 but a parallel approach is often chosen for high-end applications. In the following section, three parallel implementations are presented.

3.6.1 Bit-parallel non-redundant adder tree implementation

Multi-operand addition based on a non-redundant adder tree is the easiest way to implement multi-operand additions. Suppose we have an N-bit 2-operand adder (2N-Add) as a calculation element, the $M \times N$-operand adder tree can be built as follows.

1. $\left\lfloor \frac{MN}{2} \right\rfloor$ 2N-Add adders are needed for the first level addition. $\left\lfloor \frac{MN}{2} + 1 \right\rfloor$ result operands are produced.

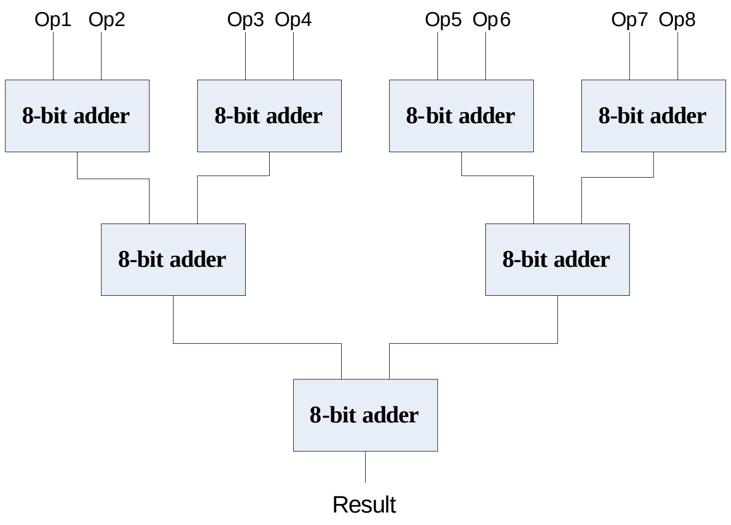

Figure 3.5: Bit-Parallel 4x2-operand adder tree

2. Feed the result operands into another series of $\left\lfloor \frac{\lfloor \frac{MN}{2} + 1 \rfloor}{2} \right\rfloor$ 2N-ADD adders as inputs and produce half number of the result operands calculated at step 1.

3. Continue to add the result operands until number of result operands is equal to 1 and the summation terminates.

This type of adder tree can be pipelined to increase the throughput. Its cycle time depends on the word size and number of operands to be added. For example, if we have 256 8-bit operands to be added, the critical path delay in a hardware pipeline is $(log_2 256) + 8 = 16$: a 16-bit addition delay.

The speedup of this adder tree is $MN/log_2 MN$ over the sequential approach. Assume that the cycle time in the sequential implementation is equal to that of adder tree and no pipeline execution, $MN = 256$ implies the speedup is $256/log_2 256 = 32$, which is a significant improvement over general purpose processors. With pipelined execution, the speedup can be even more significant. An example of 4x2-operand adder tree is shown in figure 3.5.

3.6.2 Bit-parallel carry-save adder tree implementation

Non-redundant number system is employed in non-redundant adder trees. In fact, the cycle time in a non-redundant adder tree can be reduced by employing carry-save adders in which the carry propagation is independent of the operand size. To illustrate the carry-save based multi-operand adder tree, a computation element, 4-to-2 compressor, is used as a primitive calculation. It adds 4 operands and produces 2 result operands. A example of 4-to-2 compressor based adder tree for addition of 16 operands is shown in [10].

When using a 16-bit 4-to-2 compressor as a primitive component, the construction of 4-to-2 compressor based adder trees is similar to that of non-redundant adder trees. The number of adder tree levels is reduced by one in carry-save adder trees. The output of carry-save adder tree is in carry-save format and the final output should be added using a carry propagation adder to convert it into a non-redundant number when desired. As a result, their effective pipeline levels are equal. In contrast, the cycle time of a 4-to-2 compressor is less than that of a non-redundant adder since its carry save nature shortens the carry propagation delay. As a result the speedup over general purpose processors is greater than non-redundant adder trees.

3.6.3 Bit serial signed digit adder tree implementation

The mentioned adder trees above are implemented in a word-parallel manner. A primitive computation element involves N-bit additions and occupies a large silicon area. To address this issue, we try to implement multi-operand additions in bit-serial approach. Only MSB-first approach is discussed here.

Signed-digit representation must be used to enable MSB-first addition. Signed-digit adders [10] are used throughout the adder

Cycle	A^{+}[a]	A^{-}	B^{+}[b]	B^{-}	R^{-}	R^{+}[c]
0	0	1	1	0	0	0
1	0	0	1	0	0	0
2	0	1	1	0	1	1
3	1	0	1	0	0	1
4	0	1	1	0	0	1
5	1	0	0	1	0	0
6	0	1	1	0	1	1
7	0	1	0	1	0	0
8	0	0	0	0	1	0
9	0	0	0	0	0	0

[a] $A = \bar{1}0\bar{1}1\bar{1}1\bar{1}\bar{1}(-151_{10})$
[b] $B = 11111\bar{1}1\bar{1}(245_{10})$
[c] $R = 00011000\bar{1}0(94_{10})$

Table 3.1: Example for online signed-digit adder

tree. Figure 3.6 shows the architecture of a signed-digit bit-serial adder for 2-operand additions. Table 3.1 shows an example of how 2-operand addition is performed for signed-digit inputs.

The construction of adder tree based on signed-digit adders is also similar to non-conventional adder tree but the cycle time is shorter since it is carry-free and bit-serial. The construction of signed-digit adder tree is discussed in [50]. It gives several optimizations for silicon area and delay for different number of operands.

3.7 Comparison algorithms

In motion estimation, each SAD computed should be passed to a comparison unit and checked if its value is less than the current minimum value. The minimum SAD and motion vector is updated when this is the case. In MPEG-1, MPEG-2, the macroblock size is 16 by 16 and the final SAD size is 16 bits.

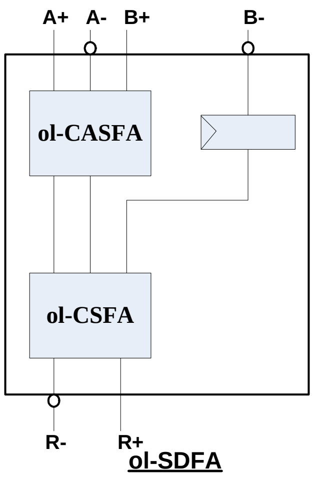

Figure 3.6: Bit-serial signed-digit adder (ol-CSFA stands for on-line carry-save full adder)

A 16-bit comparison should be made. In contrast to MPEG-2, H.264/AVC supports variable block size motion estimation and the number of comparisons for each SAD is increased to 41. The comparisons are divided into 12-bit, 13-bit, 14-bit, 15-bit and 16-bit types for 41 minimum SAD values. As comparison algorithms can be MSB-first or LSB-first, the mode selected affects only the complexity. MSB-first is common since the algorithm can be terminated earlier. In the following sections a MSB-first comparison algorithm is described.

3.7.1 Non-redundant comparison algorithm

In non-redundant number systems, implementing the comparison of two numbers starting from the MSB is simple and the process can be terminated earlier without examining all bits. For example, it can be deduced that “0111” is larger than “0011” in two’s complement format after examining two bits. For negative numbers, the process is the same but the criteria determining which one larger is changed. For example, “1100” is smaller than “1011” in negative representation. In both cases, the distinction is drawn as soon as the first different digit appears.

3.7.2 Signed-digit comparison algorithm

In the signed-digit number system, the comparison of two numbers is not as simple. Because of the property of number redundancy, the distinction cannot be drawn as soon as the first different digit appears. For example, "$01\bar{1}\bar{1}$" is smaller than "0010". Similar cases occur for negative numbers.

An algorithm to perform on-line comparison is suggested [45]. It uses the fact that if the difference of two digit strings being compared is equal to or greater than two, the comparison can be terminated and result can be determined. The proof is shown below [45].

Given two radix-2 signed-digit numbers P and Q which are N-digit numbers such that

$$P = \sum_{l=N-1}^{0} p^l \times 2^l \text{ and } Q = \sum_{l=N-1}^{0} q^l \times 2^l,$$

Suppose the comparison can determine that P is larger than Q at the 2^k digit. P and Q can be divided into

$$P = \sum_{l=N-1}^{k} p^l \times 2^l + \sum_{l=k-1}^{0} p^l \times 2^l$$

$$Q = \sum_{l=N-1}^{k} q^l \times 2^l + \sum_{l=k-1}^{0} q^l \times 2^l$$

where $0 \leq k \leq N-1$, $p^l, q^l \in \{\bar{1}, 0, 1\}$.

$$\begin{aligned}
P>Q \quad &\Rightarrow \quad \textstyle\sum_{l=N-1}^{k} p^l \times 2^l + \sum_{l=k-1}^{0} p^l \times 2^l > \sum_{l=N-1}^{k} q^l \times 2^l + \sum_{l=k-1}^{0} q^l \times 2^l \\
&\Rightarrow \quad \textstyle\sum_{l=N-1}^{k} p^l \times 2^l - \sum_{l=N-1}^{k} q^l \times 2^l > \sum_{l=k-1}^{0} q^l \times 2^l - \sum_{l=k-1}^{0} p^l \times 2^l \\
&\Rightarrow \quad \textstyle\sum_{l=N-1}^{k} p^l \times 2^l - \sum_{l=N-1}^{k} q^l \times 2^l > 2 \times \left(2^k - 1\right) \\
&\Rightarrow \quad \textstyle\sum_{l=N-1-k}^{0} p^l \times 2^l - \sum_{l=N-1-k}^{0} q^l \times 2^l > 2 \times \left(1 - \frac{1}{2^k}\right) \geq 2
\end{aligned}$$

where $max\left(\sum_{l=k-1}^{0} q^l \times 2^l - \sum_{l=k-1}^{0} p^l \times 2^l\right) = 2 \times \left(\sum_{l=k-1}^{0} 2^l\right) = 2 \times \left(2^k - 1\right)$.

As a result, if the comparison of two digit strings reaches to the state that $(p_N p_{N-1} ... p_k - q_N q_{N-1} ... q_k) \geq 2$, the comparison can be terminated at k-th digit without examining all the lower significant digits.

To sum up, in a radix-2 signed-digit comparator, once we find out that the difference is larger than or equal to two, decision can be made and the process can be terminated. As a result, early termination is still valid in signed-digit computation.

3.8 Summary

In this chapter we have explained how different operators can be implemented using different representations and processing orders which can be employed for motion estimation. MSB-first processing of addition, absolute difference, summation and comparison are explained and as we shall see, can lead to bit-level pipeline and high hardware efficiency. Similar techniques can be used in other parts of video coding such as integer transform, deblocking filter, etc.

Chapter 4

VLSI architectures for video encoding

4.1 Introduction

In this chapter variable block size motion estimation architectures are evaluated for the requirements of the H.264/AVC standard. Suitable consideration of different design tradeoffs can lead to an efficient architecture design for a given motion estimation algorithm. The purpose of this chapter is to evaluate motion estimation algorithms, mainly for full-search, from a hardware point of view, assuming H.264/AVC. A new design metric that considers processing speed in terms of throughput, silicon area occupied, memory bandwidth requirement and power consumption is introduced. Various VLSI architectures for full-search motion estimation are evaluated based on this metric. We employ 1-D, 2-D systolic array, tree architectures and signed-digit bit-serial architecture in our family of motion estimation processors. In the next section, our implementation platform is introduced first. In the subsequent sections, different processor architectures will be described in detail. The first-reported MSB-first bit-serial motion estimation processor will be described in detail. The real implementation will be presented in next chapter. A theoretical method to determine the efficiency of different architectures are given in the last section.

4.2 Implementation platform - (FPGA)

A Field Programmable Gate Array (FPGA) [9] is a device that consists of a large array of reconfigurable cells in a single chip. Each of these cells is a computation unit which can be used to implement logic functions. Configurable routing is used to allow inter-cell communication across the reconfigurable cells. Xilinx is a leading commercial company producing FPGA products and the description that follows uses their terminology. A commonly used member of this family is the Virtex series, the flagship product being the Virtex-5 which as based on 65nm process technology.

A typical FPGA contains an array of individual cell called logic cell (LC) interconnected by a matrix of wires and programmable switches. Logic circuits are built based on these cells and interconnect. FPGA also contains dedicated hardware for common-use building blocks like block memories, I/O pins, clock management blocks (DCM) , digital signal processing blocks (DSP) and embedded microprocessors. These building blocks enable system on chip (SOC) development on FPGA platforms.

4.2.1 Basic FPGA architecture

Each logic cell (LC) has look-up tables (LUT), D-type flip flops (DFF) and fast carry logic. An N-input LUT in the FPGA, N=4 or N=6, is a memory-like component that can be programmed to compute any function of up to N inputs. One output is produced for each LUTs. DFFs can be used for registers, pipeline storage, state machines, etc. Fast carry logic is a dedicated feature for speeding up carry-based computation like addition. A basic

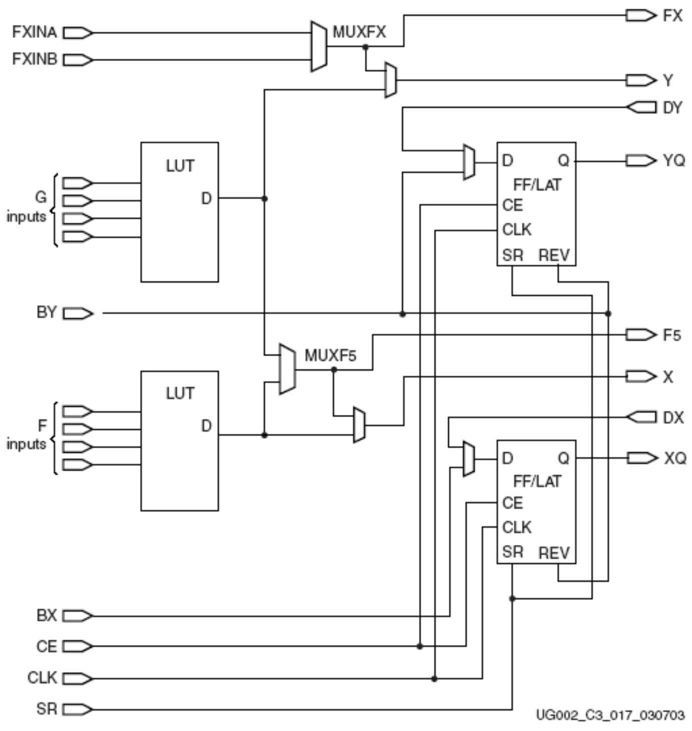

Figure 4.1: FPGA Logic Cell Architecture (Xilinx Virtex-II Pro series)

structure of LC in Xilinx Virtex-II Pro series with two 4-input LUTs, two DFFs or latches, and fast carry logic chain is shown in figure 4.1 [1].

Based on these logic cells and interconnects, an FPGA chip is realized. Together with clock management blocks, I/O blocks, microprocessors in the FPGA, the generic FPGA architecture is produced.

4.2.2 DSP blocks in FPGA device

Recently, DSP block are included in FPGA fabrics [2] for high-speed digital signal processing applications. In the first generation DSP blocks, e.g. Xilinx Virtex-II, only multipliers were included. Later, multiply and add, multiply and subtract were also included for different kinds of DSP applications. Filtering

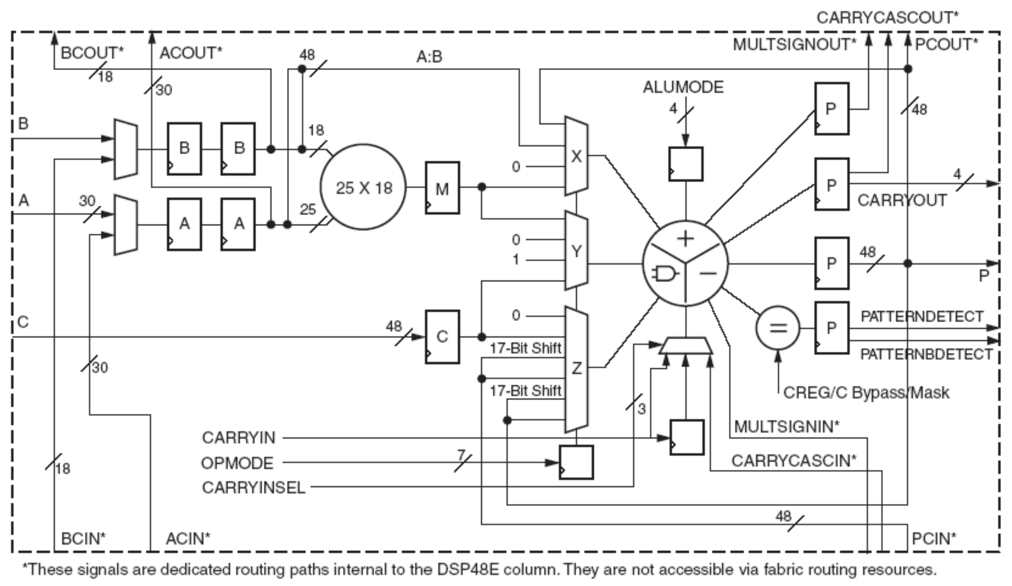

Figure 4.2: DSP architecture in Xilinx Virtex-5 FPGA

and transform operations in video codec benefit from DSP resources. Its high performance and low power consumption make DSP blocks attractive compared with the same functions implemented in configurable logic blocks. The second generation DSP block architecture in Xilinx Virtex-5 FPGAs is shown in figure 4.2 [2].

4.2.3 Advantages employing FPGA

Moore's Law [35] directly benefits development of high-speed, high-density and low-power FPGA devices. Advancement in FPGA architectures and process technologies, have narrowed the performance, area, power gaps between FPGA and ASIC devices [14]. Modern FPGA devices are suitable for complex, large systems that in the past could only be implemented in ASICs. With its reconfigurable nature, designs can be updated at the hardware level easily without replacement of the whole chip, which reduces the overall system cost. Moreover, with advances in dynamic reconfiguration technologies, FPGA can electively load the hardware within the period that the function is needed and detach the hardware afterwards [5]. As a result, less logic resources are idle or wasted compared to ASIC implementations in which all logic is fixed.

FPGAs benefit motion estimation by providing excellent logic performance, large amounts of silicon resources and flexible hardware. With its reconfigurable nature, a motion estimation processor can be customized based on application requirements. Our family of motion estimation processors is a set of architectural choices for reconfiguration. This "load when needed" methodology reduces the required resources and power consumption needed and better fits the application requirement.

4.2.4 Commercial FPGA Device

Many commercial FPGA families are currently available on the market. Xilinx (www.xilinx.com) offers the Virtex and Spartan series, Altera (www.altera.com) offers Stratix and Cyclone series and Actel offers the Fusion series (www.actel.com). All commercial products range from high-end to low-end device. We choose Virtex-II Pro platform from Xilinx for our implementation because it satisfies most of the requirements for our motion estimation processors. These devices are expensive but a less expensive choice, the Spartan series from Xilinx is a good substitute. The downside is that Spartan offers a smaller amount of hardware resources and slower performance. Fortunately, it is enough to fulfill low and mid-end applications in video processing [14].

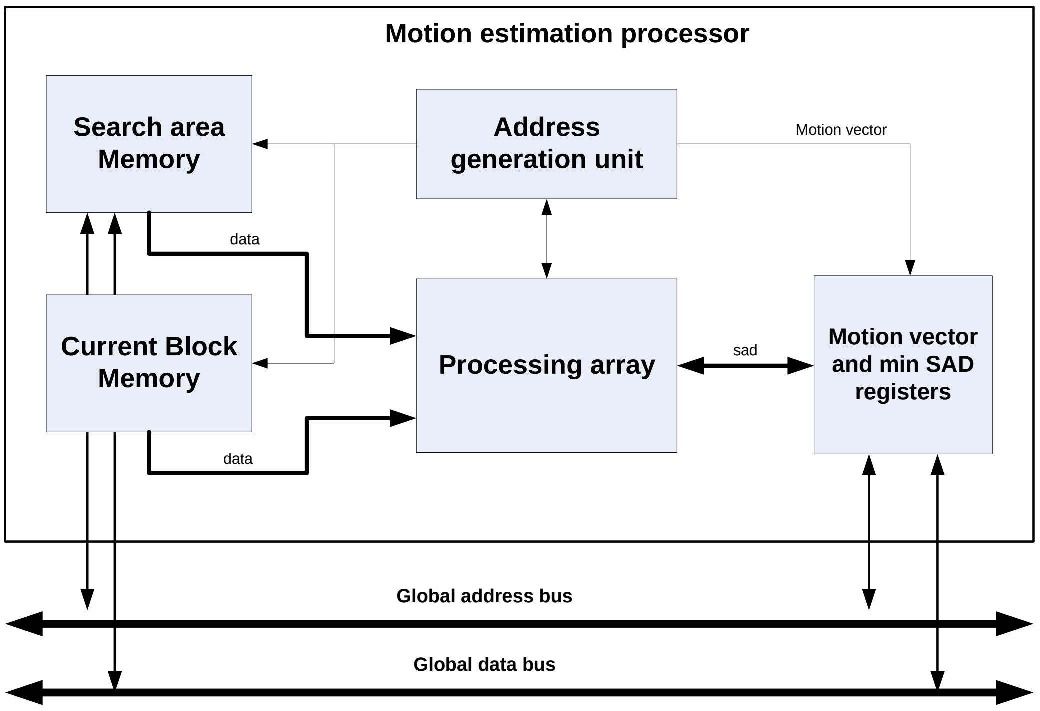

Figure 4.3: Model of motion estimation processor

4.3 Top level architecture of motion estimation processor

H.264/AVC top level architecture for a motion estimation coprocessor is shown in figure 4.3. In the motion estimation processor, there are four fundamental units, namely the pixel memory, processing array, address generation unit and motion vector memory. Typically, there are two memories storing search area and current block pixels. The processing array is designed to calculate the required SAD. The address generation unit calculates the addresses for the following data in memory. For different search algorithms, different address generation schemes are used. The resulting SADs and motion vectors are stored in a small memory accessible via an external bus which acts as a communication bridge between the motion estimation processor and a general purpose processor.

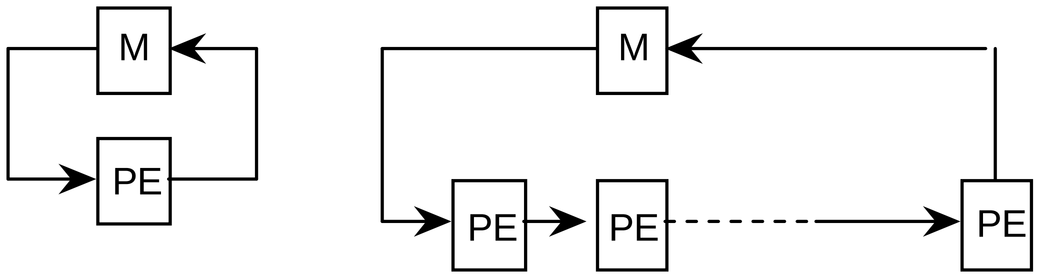

Figure 4.4: Data flow in systolic array over general implementation

4.4 Bit-parallel architectures for motion estimation

In bit-parallel motion estimation architectures, systolic arrays are commonly employed since they can effectively sustain high bandwidth between memory and computation cores while at the same time provide good performance by utilizing many computation elements in parallel. We will give background on systolic arrays and explore alternatives in using systolic arrays for motion estimation.

4.4.1 Systolic arrays

A systolic array [26] is an array processor architecture that consists of a number of identical processing elements inter-connected via local communication links. The computation is performed in a pipelined manner and results are passed through the processing elements. Its advantages are low communication overhead, simplicity and the architecture VLSI implementation. Figure 4.4 demonstrates the data flow in a systolic array as compared with a general approach.

Variable block size motion estimation algorithm

$$SAD_{P,Q}(m,n) = \sum_{i=0}^{P}\sum_{j=0}^{Q}|c(i,j)-r(i+m,j+n)| \tag{4.1}$$

$$-range \leq m,n < +range, \tag{4.2}$$

$$P,Q \in \{4,8,16\}, \tag{4.3}$$

$$u = min_{m,n}\{SAD_{P,Q}(m,n)\} \tag{4.4}$$

$$MV_{min(P,Q)} = (m,n) \tag{4.5}$$

where *range* is the search range having values ±16 or ±32.

Figure 4.5: Variable block size motion estimation algorithm

4.4.2 Mapping of a motion estimation algorithm onto systolic array

The algorithm is first decomposed into basic operations and converted into a form where each result is assigned to a unique variable. Referring to chapter 2, the variable block size motion estimation algorithm for full search is defined in figure 4.5.

Parameters P and Q have been added to reflect the variable block size. In motion estimation, SAD is computed over a four-dimensional index space, i, j, m, n for each macroblock in a frame. Equation 4.1 shows two 2-D index spaces only. The first one is generated by the indexes i, j for calculating $SAD_{P,Q}(m,n)$. The second one is generated by m, n. The minimum SAD is found and a motion vector deduced after exploring all m, n pairs in the full search. The indexes i, j can be projected onto a 1D or 2D systolic array. The number of computation nodes depends on the block size. The other way is to project i, m onto the systolic plane and results in an array dependent on block size and search range. The parallelism can be higher if the search range is larger than the macroblock size. For dif-

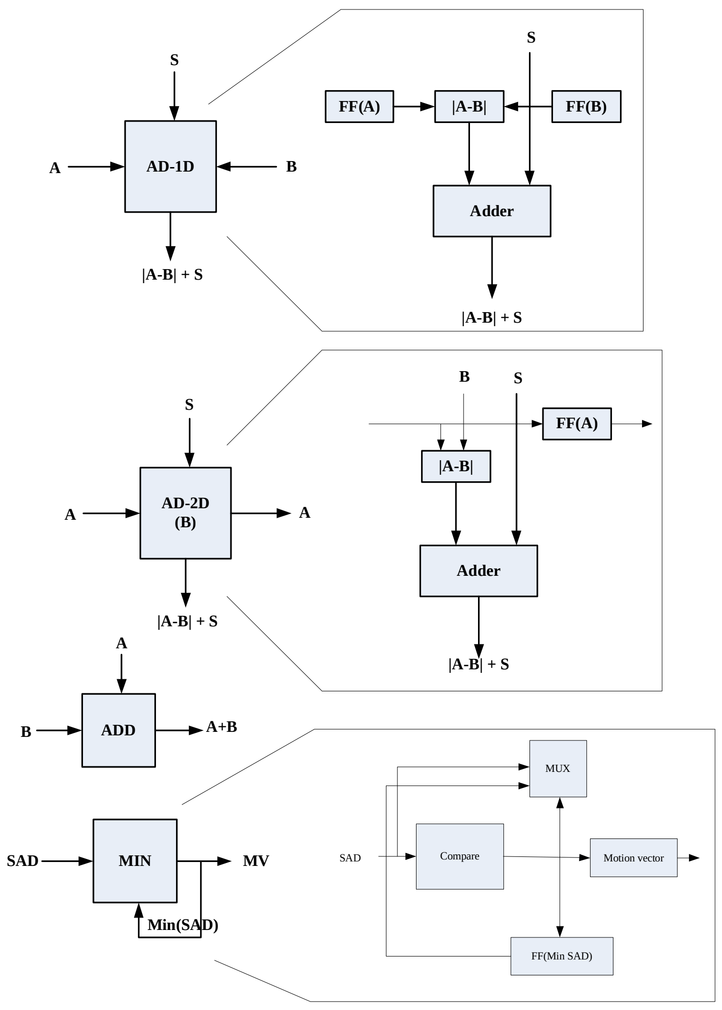

Figure 4.6: Fundamental elements in systolic and tree architectures

ferent mapping alternatives, a number of examples of mapping motion estimation algorithms to systolic arrays is given by [24]. Each computation node handles a subtraction, an absolute value operation, and an accumulation in the systolic architectures.

For the description of motion estimation architectures some basic elements are defined. They are the processing element (PE) such as AD-1D and AD-2D, the ADD node and MIN node. The detailed dataflow for these nodes are shown in figure 4.6. In figure 4.6, FF standards for flip flop and the absolute difference, addition and comparison units are implemented in a bit-parallel

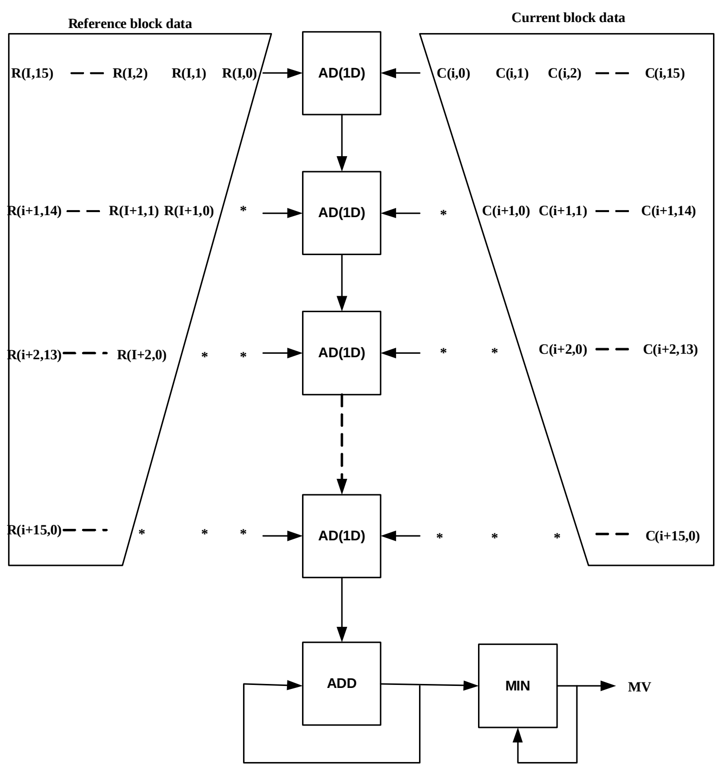
Reference block data
Current block data
R(I,15) R(I,2) R(I,1) R(I,0)
R(i+1,14) R(I+1,1) R(I+1,0)
R(i+2,13) R(I+2,0)
R(i+15,0)
AD(1D)
C(i,0) C(i,1) C(i,2) C(i,15)
C(i+1,0) C(i+1,1) C(i+1,14)
C(i+2,0) C(i+2,13)
C(i+15,0)
ADD
MIN
MV

Figure 4.7: 1-D systolic architecture

fashion.

4.4.3 1-D systolic array architecture (LA-1D)

Figure 4.7 depicts the 1D systolic processing array. $r(x, y)$ and $c(x, y)$ stand for reference block from search area and current block respectively. It is classified as a local accumulation architecture since it involves summation within the processing node. This design requires external memories to store the search area and current block, resulting in a high memory bandwidth requirement.

Each AD PE calculates the absolute difference of two pixels from the reference block and current block, adds the result to the already calculated partial sum for the same search position given by the neighbour PE, and passes the result to the next

PE. At the end of the chain of PEs, the SAD calculation is finished. The result is compared with the previous minimum SAD result within the MIN element. This architecture is scalable for search range and block size. The area used is small but the performance is slow compared to a 2D-array. It is suitable for low-end applications.

4.4.4 2-D systolic array architecture (LA-2D)

The 2D systolic processing array is a two dimensional version of the LA-1D architecture. Figure 4.8 shows its datapath and timing of the data flow. The reference data is passed horizontally from one AD-2D to the next. Data reuse is possible by making use of delay lines and by moving data from one PE to the next. This architecture offers the advantage of further reducing memory bandwidth compared to the LA-1D architecture. The current data is initially shifted to each PE and will be stored and reused until the current block motion vector is found. This architecture requires large area but the performance is high because of the number of parallel computations. As a result, it is suitable for high-end applications.

4.4.5 1-D Tree architecture (GA-1D)

The global accumulation architecture, GA-1D, is also referred to as a “tree architecture”. Figure 4.9 below shows this architecture for a 4x4 macroblock size. The absolute difference of a previous and current pixel is calculated in the absolute value PE and the result is accumulated in an adder-tree external to the PE array. The adder tree is usually implemented as a non-redundant adder-tree or carry-save adder tree with pipeline registers inserted between the stages. The previous data are fed continuously into the PEs whereas the current block data is loaded when the current block is changed and are kept in

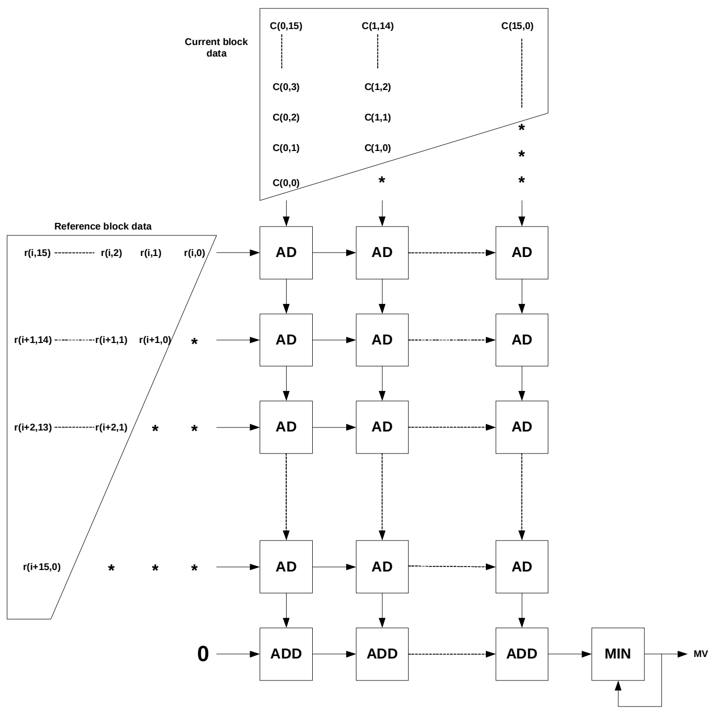

Figure 4.8: 2-D systolic architecture

registers if local caches are added.

This architecture reduces memory bandwidth for current pixel data compared to the LA-1D architecture. This architecture can also be fully pipelined in an FPGA or ASIC. In addition, advanced features such as variable block size motion estimation can be easily supported in this architecture by adding immediate registers to store up SADs of subblocks.

4.4.6 2-D Tree architecture (GA-2D)

The GA-2D architecture depicted in figure 4.10 is the two dimensional extension of the GA-1D architecture, in which NxN PEs are used, as illustrated for a 4x4 block size case in figure 4.10. As shown in the figure, the search area pixels are fed continuously into the PEs whereas the current block data is loaded only once when the current block is changed.

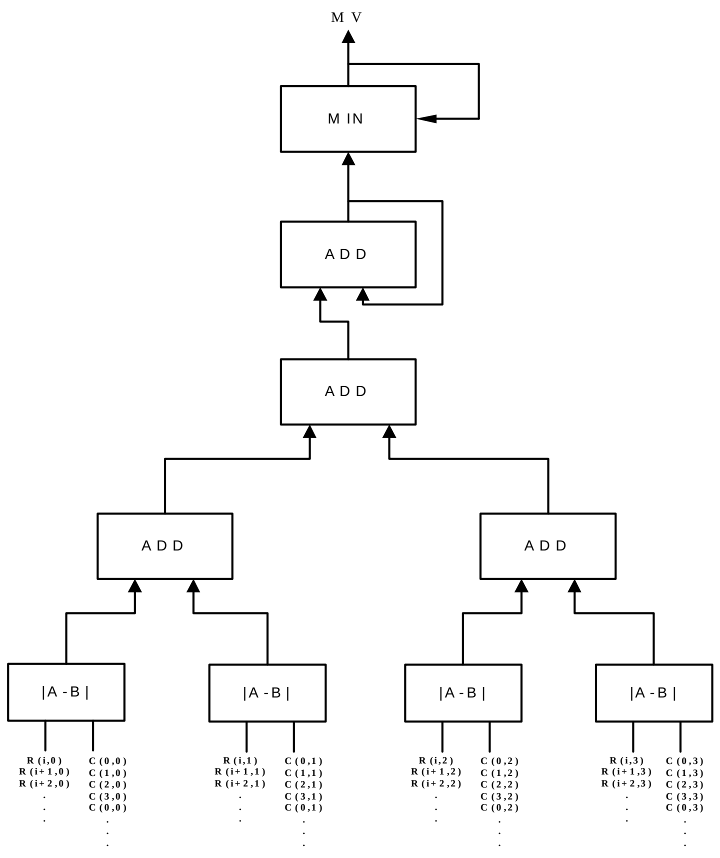
M V
M IN
A D D
A D D
A D D
A D D
|A - B |
|A - B |
|A - B |
|A - B |
R (i,0) R (i+1,0) R (i+2,0)
C (0,0) C (1,0) C (2,0) C (3,0) C (0,0)
R (i,1) R (i+1,1) R (i+2,1)
C (0,1) C (1,1) C (2,1) C (3,1) C (0,1)
R (i,2) R (i+1,2) R (i+2,2)
C (0,2) C (1,2) C (2,2) C (3,2) C (0,2)
R (i,3) R (i+1,3) R (i+2,3)
C (0,3) C (1,3) C (2,3) C (3,3) C (0,3)

Figure 4.9: 1-D Tree architecture

The local communication between PEs reduces memory bandwidth on reference data compared with GA-1D. Since it avoids local accumulation and can be fully pipelined efficiently in FPGA, the performance of this architecture is the best among four implementation alternatives.

4.4.7 Variable block size support in bit-parallel architectures

There are two methods to support variable block sizes in bit-parallel architectures. For local accumulation architectures like LA-1D and LA-2D, first method is to include 16 partial SAD registers in each node for 4x4 subblocks within a 16x16 mac-

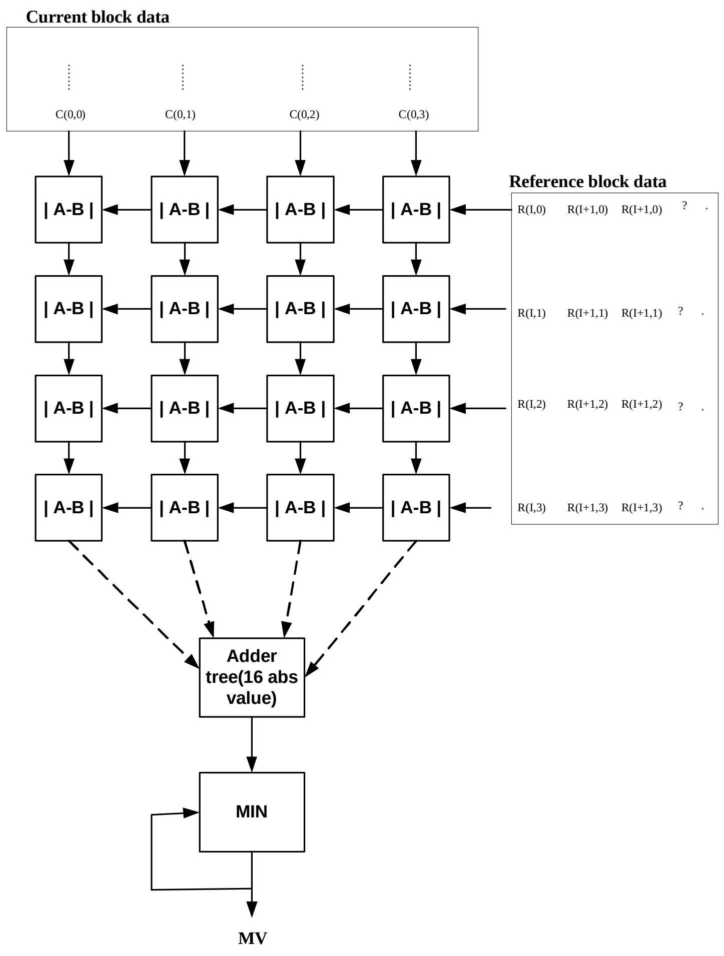
Current block data
C(0,0)
C(0,1)
C(0,2)
C(0,3)
Reference block data
R(I,0) R(I+1,0) R(I+1,0) ? ·
R(I,1) R(I+1,1) R(I+1,1) ? ·
R(I,2) R(I+1,2) R(I+1,2) ? ·
R(I,3) R(I+1,3) R(I+1,3) ? ·
| A-B |
Adder tree(16 abs value)
MIN
MV

Figure 4.10: 2-D tree architecture

roblock so that each register stores its corresponding subblock SAD [6]. The second method is to divide the systolic array into its smallest possible block size architecture (sub-systolic array). For example, a 16x16 systolic array is divided into sixteen 4x4 systolic arrays, each handling a 4x4 SAD. The sixteen 4x4 SADs are then combined to form a large SAD via SAD-reuse technique through an adder tree [7]. The first method imposes a large register overhead on each processing node and can have a large impact on silicon area. The second may increases the required bandwidth as the local communications between sub-systolic arrays are broken.

In tree architectures, the implementation is easier since there is no partial SAD stored in the systolic elements. The support of variable block sizes can be done in the adder tree by adding intermediate registers connected to comparison units in different stages of the adder tree. Similarly, using the SAD-reuse technique, variable block size motion estimation is supported without significantly sacrificing memory bandwidth and silicon area.

4.5 Bit-serial motion estimation architecture

4.5.1 Data Processing Direction

In bit-serial implementations, data processing from the MSB and LSB leads to the same result but performance may be different. Generally, addition and subtraction are LSB-first favored algorithms but can be implemented in MSB-first by deploying a redundant number system. The final operation in motion estimation involves comparison, a MSB-first favored algorithm. The comparison operation can finish earlier if it is processed MSB-first. As a result, for better throughput, we choose a MSB-first implementation for bit-serial motion estimation.

4.5.2 Algorithm mapping and dataflow design

When employing a bit-serial approach, the inherit bandwidth is reduced since in each cycle the bandwidth required is divided by n for n-bit pixels. As the bit-serial architecture is not deeply pipelined, the pipeline flushing due to the effect of data hazard introduced by data dependent algorithms such as TSS, DS is smaller than that in systolic and tree designs. As a result, bit-serial implementations are suitable for fast algorithms and thus for low to mid-end applications.

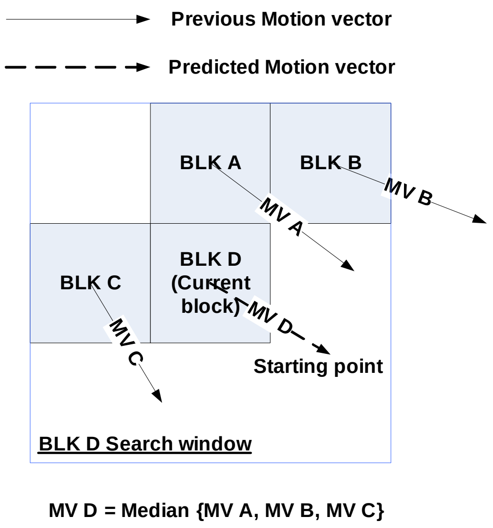

Figure 4.11: H.264/AVC motion vector prediction

4.5.3 Early termination scheme

There are two related advantages having a good initial value for the minimum SAD. The first is that early termination of comparisons to the current minimum SAD can be affected more frequently, and the second is that updates to the minimum SAD value take extra cycles, and better initialization can serve to reduce their occurrence. H.264/AVC uses motion vector prediction mode (figure 4.11) and we can initializes the search to the predicted location.

In the typical case, this serves to reduce the number of SAD updates as the search is started with a near-minimum value. Table 4.1 summarizes our simulation results in Matlab showing the number of clock cycles needed to complete the comparison operation for different video scenes with different motion vector initialization strategies. A non early termination implementation requires 16 cycles since the longest word size is 16-bit for the comparison. The News scene is an almost still motion video. Zero-assumed motion and predicted MV initialization performs

Video type	Sequential	Zero MV	Predicted MV
News	6.95	5.39	5.4
Flower	6.64	5.83	5.5
Stefan	7.26	6.54	6.46

Table 4.1: Number of cycles to complete comparison stage for different scenes using different starting strategy (16 cycles for no early termination scheme)

better than a standard sequential scheme. In fast motion scenes, such as flower and Stefan, which constitute a fast moving background and a sports scene, the H.264/AVC predicted MV initialization scheme performs the best and has an average of 5.79 cycles. On average our scheme offers a (16-5.79)/16=63.8% savings in comparison operations. For the entire motion estimation computation, in total (12+16)=28 cycles (figure 4.20) are required in the worst case, and on average our scheme offers a 36.5% improvement.

4.5.4 Top-level architecture

Motion estimation involves the calculation of SAD values between current block and reference block as shown in equation 2.4. By rewriting equation 2.4 in bit-serial fashion, we get equation 4.6 with a triple summation.

$$SAD_{P,Q}(m,n)=\sum_{i=0}^{P}\sum_{j=0}^{Q}|\sum_{k=0}^{7}2^k\times(c(i,j,k)-r(i+m,j+n,k))| \tag{4.6}$$

The double summation (P,Q) are mapped to the signed-digit adder tree and computed spatially while the innermost summation (0 to 7) of bit-serial part is computed iteratively. The remaining problem is how to generate signed-digit numbers from current and reference pixel values. Both current and reference pixels are positive 8-bit integers. The computation of their difference in signed-digit representation can be done easily by

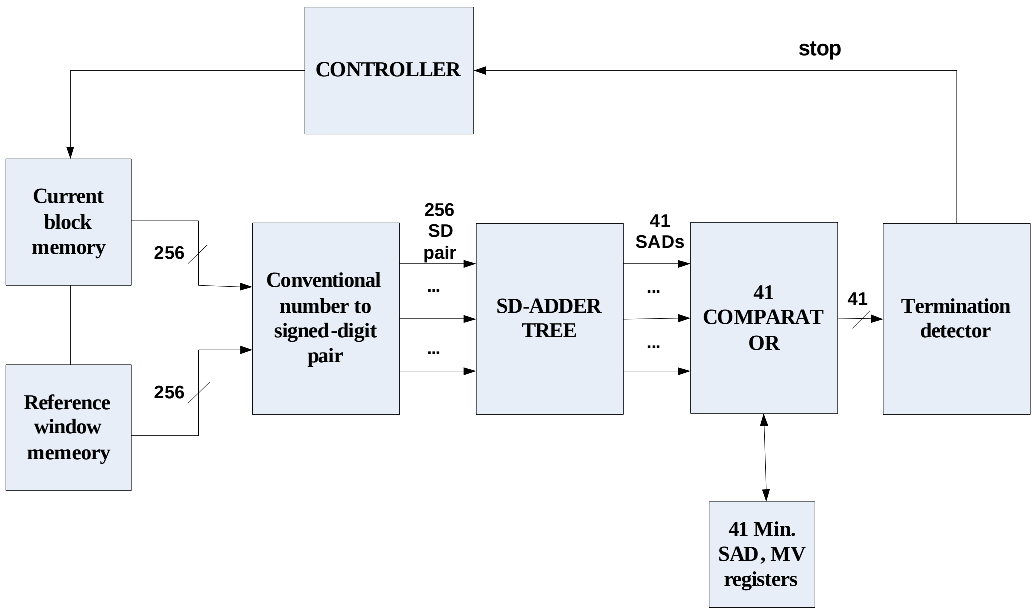

Figure 4.12: Top level architecture of bit-serial motion estimation unit

making the current pixel positively weighed and the reference pixel negatively weighed as discussed in section 3.5. The absolute value operation can be done by on-the-fly checking of the signed-digit number until 1 or -1 is detected for the first non-zero digit. The positive weighing is interchanged with the negative weighing part to complete the absolute value operation if -1 is encountered.

Then, we describe the entire bit serial motion estimation process in 4 stages: non-redundant positive number to signed-digit number conversion, summation, comparison and early termination. The top level system is shown in figure 4.12.

4.5.5 Non redundant positive number to signed digit conversion

As described in chapter 3, the $|ci - ri|$ operation, where ci and ri are 8-bit positive integers from the current and reference blocks, can be converted to a SD representation by setting ci and ri as

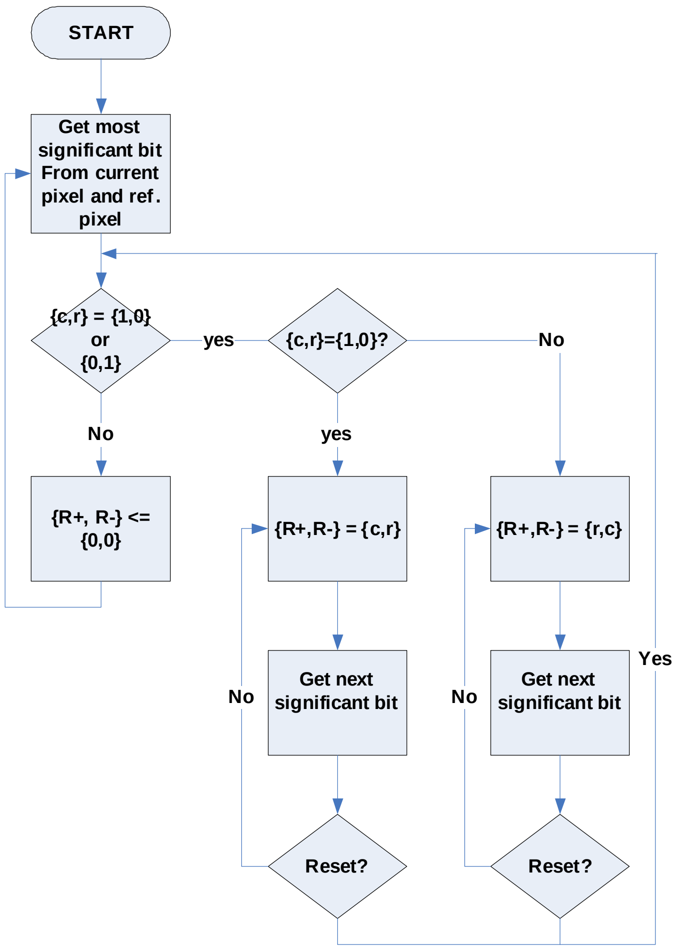

Figure 4.13: Flow chart of non-redundant to signed-digit number conversion

being positively and negatively weighed respectively and finally doing a sign-detection to check if changing the sign of result is necessary. The circuit that implements this functionality requires few hardware resources and little computation delay is introduced. A finite state machine which detects the first non-zero digit is required for the absolute value. Together with a pair of multiplexers for interchanging the signed-digit, $|ci - rj|$ in signed-digit form is produced.

Figure 4.13 shows the flow chart for sign-detection of the signed-digit number. In total there are $16 \times 16 = 256$ absolute difference stages in our motion estimation processor.

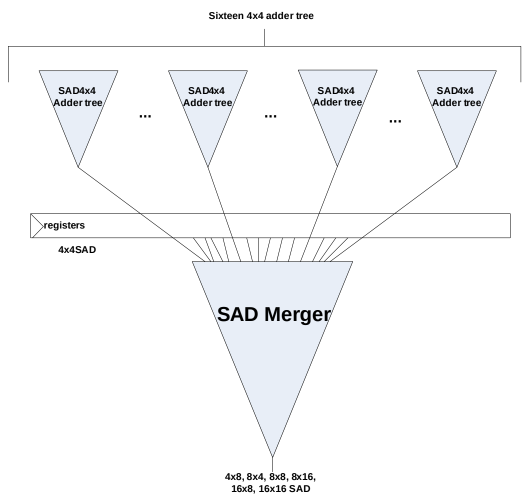

Figure 4.14: Signed-digit adder tree that generates 41 SADs

4.5.6 Signed-digit adder tree

The macroblock size of H264/AVC is 16 pixels by 16 pixels with a 4x4-block as its smallest sub-block. To find all the minimum motion vectors of a 16x16-block and its subblocks, we employ of a SAD-reuse strategy [7]. As a result, the 4x4-SAD computation becomes our primitive element and is reused to form other SADs. Since the different macroblock modes are overlapped in the spatial domain (Figure 2.5), the SAD can be calculated using 4x4 SADs and a sequence of merging steps to obtain other SADs. For example, an 8x4 and 4x8-SAD can be formed by combining corresponding values of 4x4-SADs (e.g. 4x4-SAD (Block 1, 2) in figure 2.5 to form 8x4-SAD(Block 17)). Similarly, an 8x8-SAD can be formed from 4x8-SADs, 16x8 and 8x16-SADs can be calculated from 8x8-SADs and finally a 16x16-SAD is combined from 16x8-SADs. The top level adder tree is shown in figure 4.14.

The SAD for a 4x4 subblock is produced by 16 pairs of operands summed in signed-digit format, implying we need to add 32 bit operands in our adder tree. According to [50], we could implement a 16-operand signed-digit adder tree based on

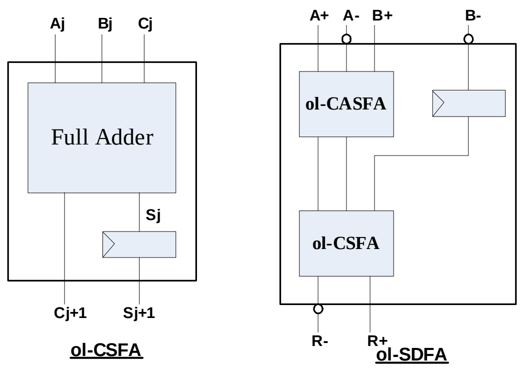

Figure 4.15: On-line carry save and signed digit adders

two ol-CSFA trees and an ol-SDFA. These primitives are shown in figure 4.15. Together, the hardware utilization in this adder tree is minimized [50]. This is illustrated in figure 4.16 and figure 4.17 and consists of 8 levels with 8 cycles of on-line delay. The total number of cycles to calculate the 12-bit summation including the on-line delay is $8 + 8 = 16$ cycles. The output of a SAD4x4 adder tree is the SAD value of a 4x4-subblock in signed-digit format. This value is passed to the SAD Merger unit to calculate other necessary SADs.

4.5.7 SAD merger

In our design we need sixteen SAD4x4 adder trees to compute the SAD of 16 subblocks in parallel. The sixteen SAD4x4 values computed are passed to the SAD merger as inputs (figure 4.18). The sixteen 4x4-SADs are fed to a series of ol-SDFAs, i.e. SAD merger, and combined to form 4x8, 8x4, 8x8, 16x8, 8x16 and 16x16 SADs. The number shown in figure 4.18 indicates which block's SAD is calculated at that node. The block index is shown in figure 2.5. In total, the number of ol-SDFAs in SAD merger is 8+8+4+2+2+1=25. Pipelining registers are added between SAD4x4 adder trees and the SAD merger to split the combinatorial path and boost the operating frequency. In our

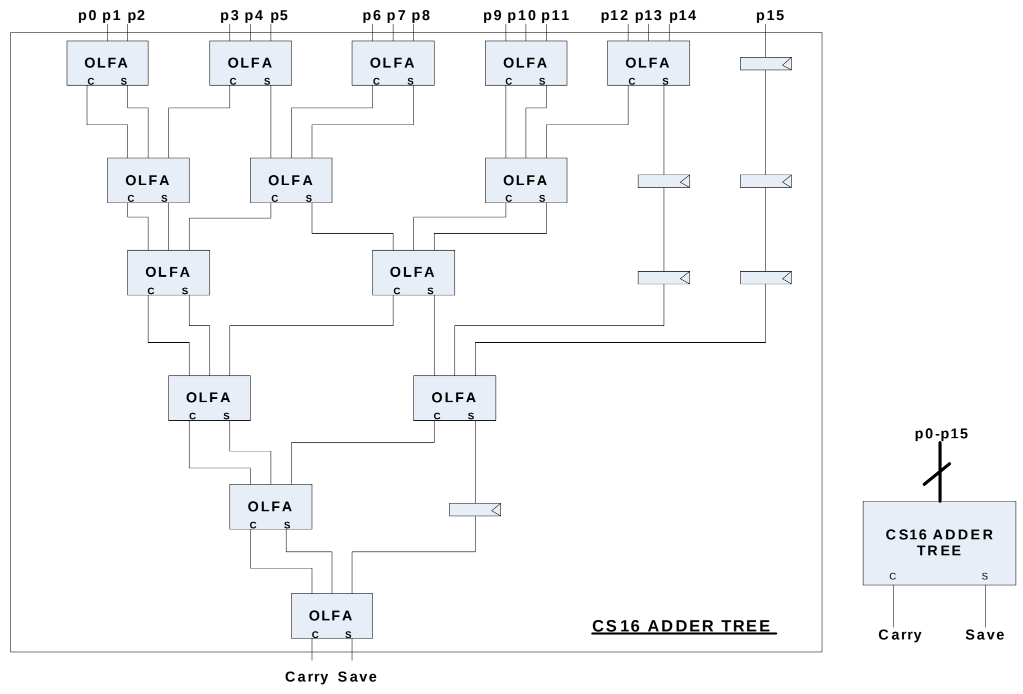

Figure 4.16: A 16-operand carry save adder tree

FPGA prototype, one pipeline register obtains a good balance between maximum frequency and latency.

Finally, the 41 SAD values are passed to an on-line comparator for final stage processing. Since the arrival times of different SAD results are different, the completion times to determine the minimum SAD vary. Table 4.2 shows the delay for each type of SAD.

4.5.8 Signed-digit comparator

In the comparison stage, we compare the current SAD to the current minimum SAD for each subblock type in a MSB-first manner. A signed-digit comparator is used for this purpose. The architecture of the comparator suggested in [45] is shown in figure 4.19. If the number being compared has a difference of two or more, we can determine which SD number is bigger.

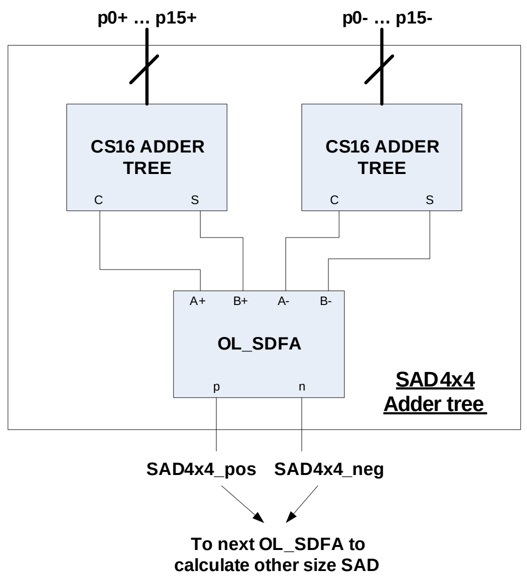

Figure 4.17: 16-operand signed-digit adder tree for 4x4 SADs

SAD type	Delay (cycles)
4x4	16
4x8,8x4	19
8x8	21
8x16,16x8	23
16x16	25

Table 4.2: On-line delay of different SAD types

The on-line comparator will stop when this situation arises. A proof for this algorithm is given in [45] and described in chapter 3.7.2. The on-line comparator can determine the result in 2 cycles minimum.

4.5.9 Early termination controller

Early termination of the SAD computation allows the avoidance of redundant calculations. In terms of processor throughput, 100% speed-up can be achieved when 50% of calculations can be eliminated. In our case, we have to deal with the variable

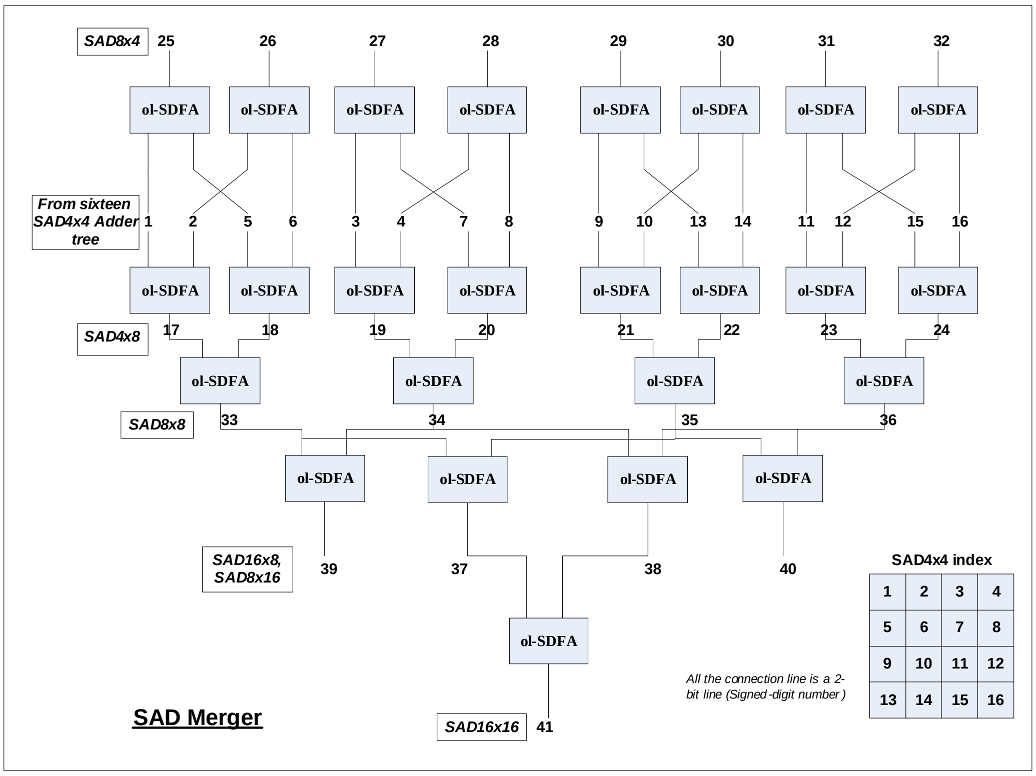

Figure 4.18: SAD merger

block size effect, which affects our early termination scheme. Since we have to compute 41 parallel comparisons, some can be terminated earlier than the others. There exists dependencies between successive types of SADs, e.g. 8x4 depends on 4x4, and we cannot terminate the 4x4 summation process even if we are sure the current 4x4 SAD must be rejected. For the sake of simplicity, we check for early termination on all SADs and when all have terminated, the current summation can be completed and begins next searching position. Termination can be detected by OR-ing all the comparator outputs.

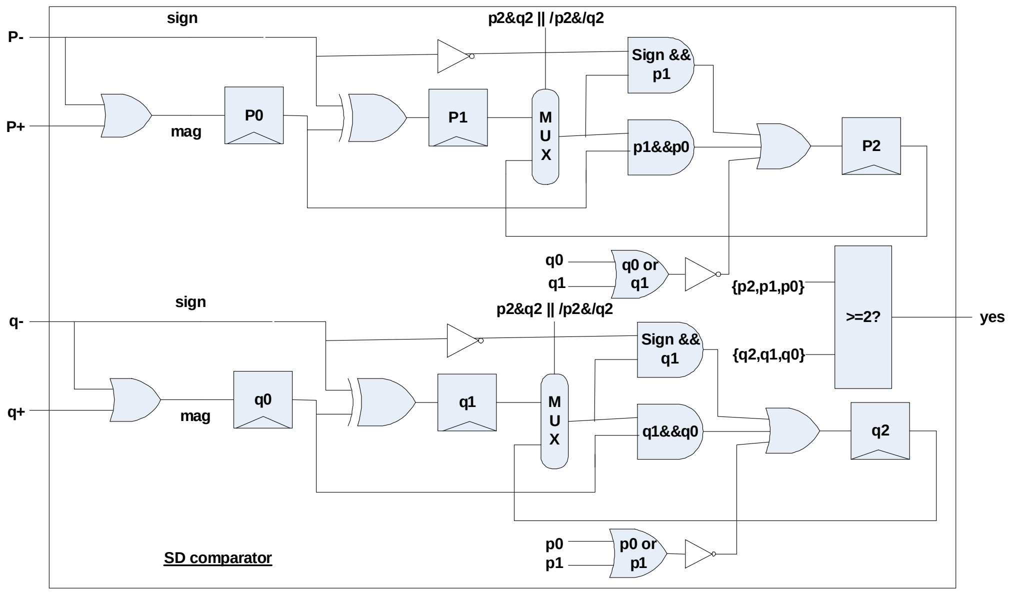

Figure 4.19: Architecture of on-line comparator

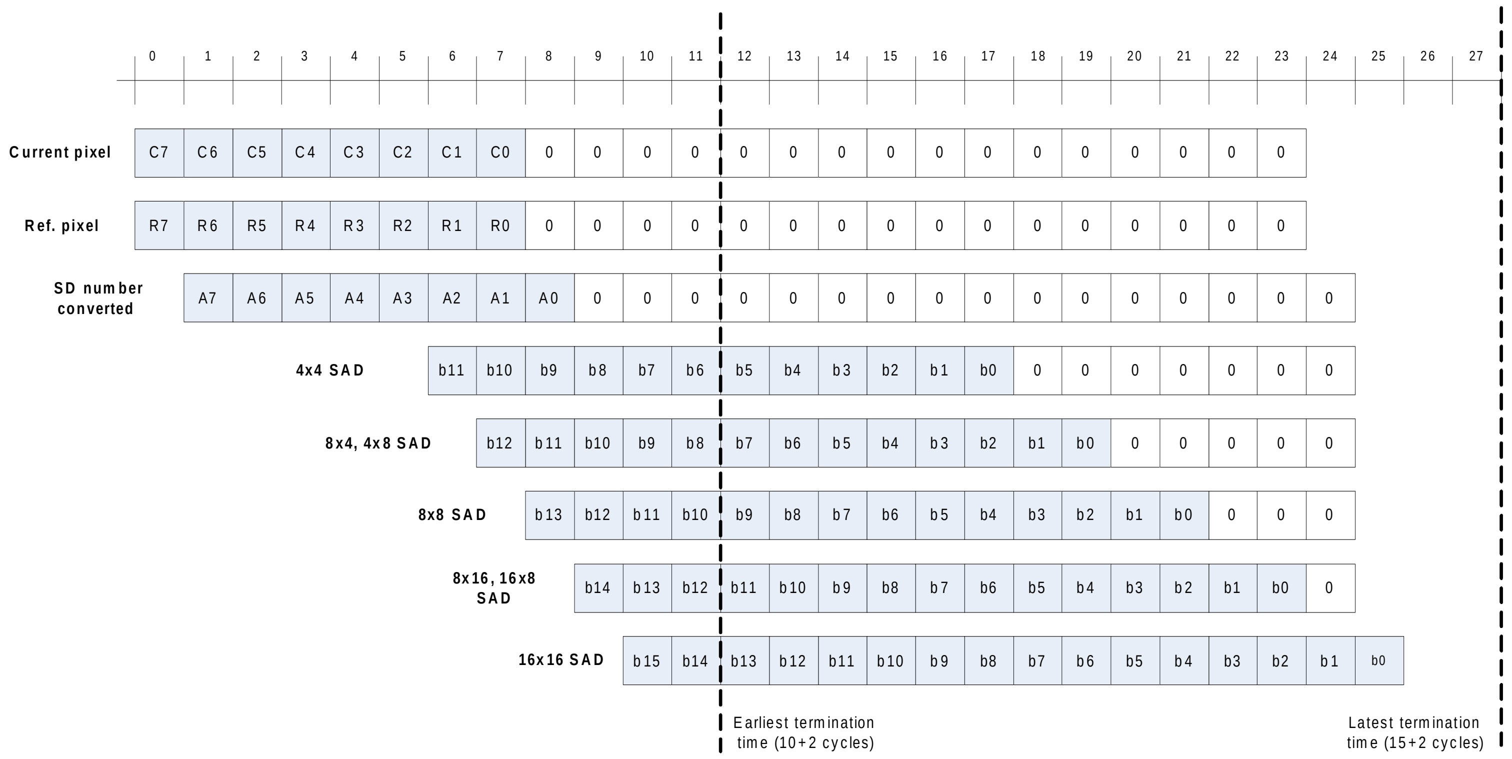
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
Current pixel
C7 C6 C5 C4 C3 C2 C1 C0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Ref. pixel
R7 R6 R5 R4 R3 R2 R1 R0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
SD number converted
A7 A6 A5 A4 A3 A2 A1 A0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4x4 SAD
b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 0 0 0 0 0 0 0
8x4, 4x8 SAD
b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 0 0 0 0 0
8x8 SAD
b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 0 0 0
8x16, 16x8 SAD
b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 0
16x16 SAD
b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
Earliest termination time (10+2 cycles)
Latest termination time (15+2 cycles)

Figure 4.20: Timeline of bit-serial design for whole motion estimation computation process

4.5.10 Data scheduling and timeline

In a bit-serial based architecture, we need to handle word-to-serial conversion which is unnecessary in a bit-parallel design. In addition, we have to handle extra scheduling brought upon by MSB-first arithmetic. For example, summation of 16 8-bit signed-digit numbers results in a 12-bit result, which involves 8 cycles of on-line delay. We have to generate 8 consecutive cycles of all-zero operands feeding into adder tree to compensate the online delay. Similarly, a 16x16 SAD requires 12 consecutive cycles of zeros as shown in figure 4.20. The 16-bit 16x16-SAD result is calculated in 28 cycles in the worse case.

4.6 Decision metric in different architectural types

In this section we analyze different bit-parallel architectures in terms of throughput, occupied silicon area, memory bandwidth requirement and power consumption. Their values are estimated theoretically. The analysis can give designers a first impression of how the characteristics of different design parameters are affected by different architectures. We make the following assumptions in the analysis that follows. The unit areas for primitive blocks are collected from the Xilinx ISE implementation tools.

1. The delay of a full adder constitutes 1 unit time.
2. A 1-bit full adder occupies 12 unit areas.
3. A multiplexor (MUX) occupies 6 unit areas.
4. A FF occupies 8 unit areas.
5. The maximum frequency of processors depends on the total unit time delay between pipeline registers.

Operation	Unit time required
N-bit addition	Nx1=N
N-bit abs(a-b)	$(N+1)\times 2+1=2N+3$
N-bit comparison	N+1+1=N+2

Table 4.3: Delays of primitive operations employed in bit-parallel motion estimation architectures

Component	Unit area occupied
PE(1D)	$12\times 12+2\times 9\times 12+6\times 8+8\times 12=504$
PE(2D)	$12\times 12+2\times 9\times 12+6\times 8+8\times 8+8\times 12=568$
PE(Tree)	$8\times 8+2\times 9\times 12+6\times 8=328$
ADD	$16\times 12+16\times 8=320$
COMP	$17\times 12+6\times 16+(12+16)\times 8=524$

Table 4.4: Areas of primitive component employed in bit-parallel motion estimation architectures

6. Block size of a macroblock is $N \times N$.

The delays and areas for primitives are deduced in table 4.3 and 4.4. The PE for 2D systolic arrays require the largest hardware demand and the PE for tree architectures require the least. Since the calculation method is different between bit-serial and bit-parallel approaches, the comparison shown here doesn't include bit-serial architectures. A comparison between two will be given in chapter 5 based on real data.

4.6.1 Throughput

We define throughput in terms of number of operations performed in a cycle time. For systolic array type architectures, a subtraction, an absolute value and an accumulation are performed per cycle time. For the tree type architectures, a subtraction and an absolute value are performed per cycle time. As a result, the delay in tree PEs is less than that of systolic PE

Architectures	Operations per unit systolic cycle time
1D systolic	$((2N+3)+(N+4))N+N+(N+2)$ $=3N^2+9N+2$
2D systolic	$((2N+3)+(N+4))N^2+N^2+N+2$ $=3N^3+8N^2+N+2$
1D tree	$1.63\times((2N+3)N+N^2+N+2)$ $=1.63\times(3N^2+4N+2)$
2D tree	$1.63\times((2N+3)N^2+(N^2-1)N+N+2)$ $=1.63\times(3N^3+3N^2+2)$

Table 4.5: Throughput of different architectural types (N: block size)

by an accumulation delay. Given that:

1. The delay of systolic PE per unit cycle is (9+1+9)+12= 31 unit.

2. The delay of tree PE per unit cycle is (9+1+9) = 19 unit.

A tree architecture can operate on average of a $\frac{31}{19}=1.63$ higher frequency than the systolic array ideally. The maximum throughput in table 4.5 can be deduced.

Table 4.5 shows the maximum throughput can be achieved in these architectures. With more hardware resources, 2D architectures can perform more operations than 1D architectures. Tree architectures perform better than others with its higher frequency.

Usually, data dependency prevents fully parallel operation of processing elements. The calculations assume a 100% efficiency. Among them, 2D architectures are more sensitive to data dependencies. The degradation of 2D architectures is more significant than 1D arrays when data hazards occur.

Architectures	Bandwidth (bytes per unit systolic cycle time)
1D systolic	$2N$
2D systolic	$N+1$
1D tree (No cache)	$3.26N$
1D tree (With cache)	$1.63N+1.63$
2D tree	$1.63N+1.63$

Table 4.6: Bandwidth requirement of different architectural types

4.6.2 Memory bandwidth

Bandwidth requirements come from reading the current and reference block pixels. A pixel size is 1 byte. In FS, the reference block of one search point is usually overlapped with that of previous reference blocks. The frequency of reading the same pixels within the FS determines the memory bandwidth. If there are no local caches for overlapped pixels, the resulting bandwidth can be very high. Assume that tree architectures operates 1.63 higher frequency than that of systolic array, the maximum bandwidth requirements of table 4.6 are drawn.

The operating frequencies required for motion estimations are adjustable depending on the motion estimation algorithm. Full search requires the highest frequency while fast algorithms are less demanding. As a result, the memory bandwidth can be lowered by employing a fast search algorithm.

4.6.3 Silicon area occupied and power consumption

Silicon areas approximately depend on operation count per unit cycle while power consumptions depend on the bandwidth requirement and throughput. Based on the assumption made above and table 4.4, the following table 4.7 and 4.8 was calculated.

α and β are weighting parameters combining the effects of throughput and bandwidth. Typically the bandwidth constant

Architectures	Silicon area
1D systolic	$504N + 320 + 524$
2D systolic	$568N^2 + 320N + 524$
1D tree (No cache)	$328N + 320N + 524$
1D tree (With cache)	$328N + 320N + 524 + 64N^2$
2D tree	$328N^2 + 320(N^2 - 1) + 524$

Table 4.7: Area estimation of different architectural types (Variable block sizes not supported)

Architectures	Power consumption
1D systolic	$\alpha(3N^2 + 9N + 2) + \beta(2N)$
2D systolic	$\alpha(3N^3 + 8N^2 + N + 2) + \beta(N + 1)$
1D tree (No cache)	$1.63\alpha(3N^2 + 4N + 2) + 1.63\beta(2N)$
1D tree (With cache)	$1.63\alpha(3N^2 + 4N + 2) + 1.63\beta(N + 1)$
2D tree	$1.63\alpha(3N^3 + 3N^2 + 2) + 1.63\beta(N + 1)$

Table 4.8: Power estimation of different architectural types (Variable block sizes not supported)

is larger since reading or writing data from and to memory bus demands larger power consumption.

4.7 Architecture selection for different applications

4.7.1 CIF and QCIF resolution

CIF and QCIF resolution are 352×288 and 176×144 respectively and 30 frames per second is a standard frame rate. The motion estimation hardware is thus require to process $\frac{352\times288\times30}{16\times16} =$ 11880 macroblocks per second, which translates to 170 Gops/sec. This resolution is commonly used in mobile video delivery or video conferencing and real time encoding at this resolution is desirable. Power consumption is also an issue in these applications. All architectures described are capable to handle the encoding in real time whatever algorithms are employed. 1D architectures require 200MHz and 2D architectures require 15MHz at real time processing. The architecture finally chosen depends on the other requirements such as area, power consumption or memory bandwidth.

4.7.2 SDTV resolution

In standard television, 704×480 and 640×480 at 30 fps have been commonly used for the last two decades. Careful design needs to be made in order to achieve real time encoding. The processing requires $\frac{704 \times 480 \times 30}{16 \times 16} = 39600$ macroblocks per second, which translates to 570 Gops/sec. Our 1D and bit-serial architectures are not able to process in real time if full search is employed. Operating frequency over 600MHz is required for 1D architectures and it is not applicable in modern FPGA devices. In contrast, only 50MHz is required for 2D architectures. Alternatively, fast algorithms, e.g. TSS, DS, can be used with a bit-serial design to achieve real time with some sort of picture quality loss. In addition, power consumption is usually not an issue as they are employed in a non-mobile device, e.g. set-top box, television. As a result, 2D architectures, bit-serial architectures at fast searches suit this application domain.

4.7.3 HDTV resolution

In HDTV application, the resolution involves 720p, 1080i and 1080p. They are used in high-quality video editing or cinema movie playback. The complexity is 4 times to 10 times higher than for in DVD video. Particularly when variable block size, multi-reference frames and sub-pixel motion estimation features are used, the complexity increases are far beyond the MPEG-2 standard. To encode real time video in these cases, a fully parallel architecture with the highest parallelism is required. For the highest profile 1080p, the motion estimation processor has to process $\frac{1920\times1080\times30}{16\times16} = 243000$ macroblocks per second (3499 Gops/sec) which is 20 times higher than encoding CIF video in real time. Typically, only 2D bit-parallel architectures at 300MHz with ASIC technology or high-end FPGA implementation can achieve this throughput.

4.8 Summary

In this chapter, several architecture alternatives are discussed together with their implementation strategies. Bit-parallel 1-D, 2-D and tree architectures are discussed and their variable block size support analyzed. For bit-serial designs, we suggested a MSB-first architecture supporting variable block sizes. We also presented an early termination scheme to save power and boost performance. Several applications are suggested for mapping of architectures. The ranges of supported applications are from low to high-end.

Chapter 5

Results and comparison

5.1 Introduction

In this chapter the result and comparison of different architectures are presented. Based of the results, the family of motion estimation processors supporting variable block size is built. Lastly, we compare our architectures to previous work in the literatures.

5.2 Implementation details

The proposed motion estimation architectures are written in VHDL hardware language, implemented, simulated and verified on a Xilinx Virtex-II Pro -6 speed grade device. The motion estimation processors are synthesized using Simplicity Pro 8.4. Place-and-route was done and the power consumption is estimated using XPower provided by Xilinx ISE tools. The area utilization, maximum frequency and estimated power consumption are shown in following sections. In order to make a fair comparison between different architectures, "performance per slice/gate" and "power consumption per slice/gate" are indicated to show our motion estimation processors efficiency on FPGA platform.

We compare both implementations of the architectures supporting fixed block size and variable block size. The search range is fixed to -16 to +15 so that the search area is 48 $\times$ 48 pixels in size. The macroblock size is 16 by 16. The full search algorithm is employed. To measure the efficiency of different architectures, we ignore the area occupied by data storage for current block pixels, reference block pixels, motion vectors and minimum SADs since they are required in other parts of a H.264/AVC coder. The amount of area that can be ignored depends on architectures. For 2D architectures the data storage for the current block cannot be ignored since it is implemented inside the systolic array. In contrast, for bit-serial architectures, the minimum SADs and motion vectors can be stored in external memories and this area is not counted. The throughput of architectures is given by $\frac{\text{cycles per macroblock}}{\text{max frequency}}$ where “cycles per macroblock” depends on the particular motion estimation algorithm.

5.2.1 Bit-parallel 1-D systolic array

Column 1 in Table 5.1 summarizes the 1D systolic array implementation without variable block size support. It consists of 16 processing elements, 1 ADD and 1 COMP component. Including the pipeline speedups, it takes 16 cycles on average to calculate 1 SAD. The total number of cycles need is $32 \times 32 \times 16 + 17 = 16401$ cycles to process one 16x16 macroblock.

Column 2 in Table 5.1 shows an implementation of a 1D systolic array with variable block size support. Variable block size support is done by the conversion of one 16-PEs array into four 4-PEs arrays. An adder tree is connected to output of these four arrays to produce SADs for larger block sizes. In this design, we have to add 3 ADD and 3 COMP components for 4x4 SAD comparisons. It takes $32 \times 32 \times 16 + 20 = 16404$ cycles

	Fixed block size	variable block size
Design Strategy	Bit parallel	Bit parallel
Max frequency(MHz)	239	230
Area (Slices)	836	1457
Area (Gate)	16788	25158
Throughput (MB/s)	14424	14021
Max bandwidth required (MB/s)	7615	6228
Performance/Slice	17.3	9.6
Performance/Gate	0.859	0.557
Total power (mW)	1344	1794
Power/Slice (mW/Slice)	1.61	1.23
Power/gate (mW/gate)	0.080	0.071

Table 5.1: Results of 1D systolic array processor

to process a 16x16 macroblock.

5.2.2 Bit-parallel 2-D systolic array

Column 1 in Table 5.2 shows the implementation results for a 2D systolic array without variable block size support. It consists of $16 \times 16 = 256$ processing elements (PE), 16 ADD and 1 COMP component. Because of the pipeline stages, it takes 16 cycles of latency to calculate 1 SAD. When employing data dependency-free algorithms, e.g. full search, on average only 1 cycle is required for calculation of 1 SAD because of pipelining. As a result, the total number of cycles needed is $32 \times 32 + 17 = 1041$ cycles for one 16x16 macroblock.

Column 2 in Table 5.2 shows the implementation results for variable block size via the conversion of a 16x16-PEs array into sixteen 4x4-PEs arrays. An adder tree is connected to the output of these 16 arrays to form SADs of larger block size. In this design, we need to add 48 ADD, 15 COMP units and an adder tree to support variable block size. Similar to fixed block size

	Fixed block size	variable block size
Design Strategy	Bit parallel	Bit parallel
Max frequency(MHz)	232	227
Area (Slices)	9478	10794
Area (Gate)	193345	215315
Throughput (MB/s)	222862	217433
Max bandwidth required (MB/s)	7424	29056
Performance/Slice	23.5	20.1
Performance/Gate	1.15	1.01
Total power (mW)	26755	29875
Power/Slice (mW/Slice)	2.82	2.77
Power/gate (mW/gate)	0.138	0.139

Table 5.2: Results of 2D systolic array processor

architectures, it takes $32 \times 32 + 20 = 1044$ cycles to process a 16x16 macroblock.

5.2.3 Bit-parallel Tree architecture

Table 5.3 shows the implementation results of 1D tree architectures that includes variable block size support and pipelining. Its performance is the most efficient among all 1D architectures. The 1D tree architecture consists of 16 PEs, each handling an absolute difference operation. Since accumulation is eliminated, it operating frequency appears higher. In total, it takes 16390 cycles to perform a full search of 1 macroblock. For variable block size support, four 4x1 trees are needed with 3 additional comparison units added. It takes 16393 cycles to perform a full search of 1 macroblock.

Table 5.4 shows the implementation results for 2D tree architectures. It has the highest throughput among all architectures and stores the current block pixels in each of its PEs. Thus its area may appear larger. This is a tradeoff as memory bandwidth

	Fixed block size	variable block size
Design Strategy	Bit parallel	Bit parallel
Max frequency(MHz)	240	240
Area (Slices)	350	925
Area (Gates)	8281	20614
Throughput (MB/s)	14643	14641
Max. bandwidth required (MB/s)	7680	7680
Performance/Slice	41.8	15.8
Performance/Gate	1.768	0.71
Total power (mW)	1160	2834
Power/Slice (mW/Slice)	3.31	3.06
Power/gate (mW/gate)	0.140	0.137

Table 5.3: Results of 1D tree-based motion estimation processor

can be greatly reduced. Variable block size is supported by attaching comparison element at the correct position of the adder tree to produce SADs of larger block sizes. Both architectures take 1049 cycles to process 1 16x16 macroblock.

5.2.4 MSB-first bit-serial design

Bit-serial design fills the gap between 1D and 2D architectures and balances throughput, bandwidth, area and power consumption. It takes 18432 cycles to process one macroblock for a full search algorithm. With variable-block-size support, table 5.5 gives its implementation results and capability.

	Fixed block size	variable block size
Design Strategy	Bit parallel	Bit parallel
Max frequency(MHz)	240	239
Area (Slices)	5789	8513
Area (Gates)	142234	184329
Throughput (MB/s)	230547	228924
Max. bandwidth required (MB/s)	3920	3920
Performance/Slice	39.8	26.9
Performance/Gate	1.62	1.24
Total power (mW)	19324	32337
Power per Slice (mW)	3.34	3.79
Power per gate (mW/gate)	0.136	0.175

Table 5.4: Results of 2D tree-based motion estimation processor

	variable block size
Design Strategy	Bit serial
Max frequency(MHz)	420
Area (Slices)	2133
Area (Gate)	55301
Throughput (MB/s)	23068
Max. bandwidth required (MB/s)	13440
Performance/Slice	10.8
Performance/Gate	0.417
Total power (mW)	13919
Power/Slice (mW/slice)	6.5
Power/Gate (mW/gate)	0.252

Table 5.5: Results of MB-first bit-serial processor

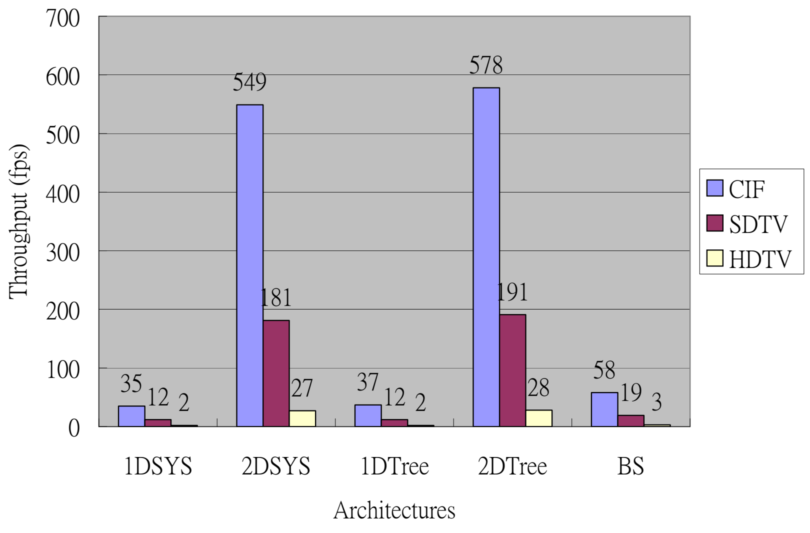

Figure 5.1: Throughput of different motion estimation architectures at different resolutions

5.3 Comparison between motion estimation architectures

5.3.1 Throughput and latency

The throughputs of different architectures at different resolutions are shown in figure 5.1. The throughput of 2D bit-parallel designs is the highest among all alternatives since its pipeline and parallel characteristics. For example, 2D tree architectures have ten times the throughput of a bit serial design. It also provides ten times less latency than a bit serial design.

Comparing 1D architectures to the bit-serial design, the bit-serial architecture has better throughput when they are working at their maximum frequency. Latency is similar between 1D systolic and bit serial architectures.

Two dimensional architectures are suitable for dealing with high throughput applications like 1080p encoding, cinema quality video creation, etc. Low throughput applications, like video

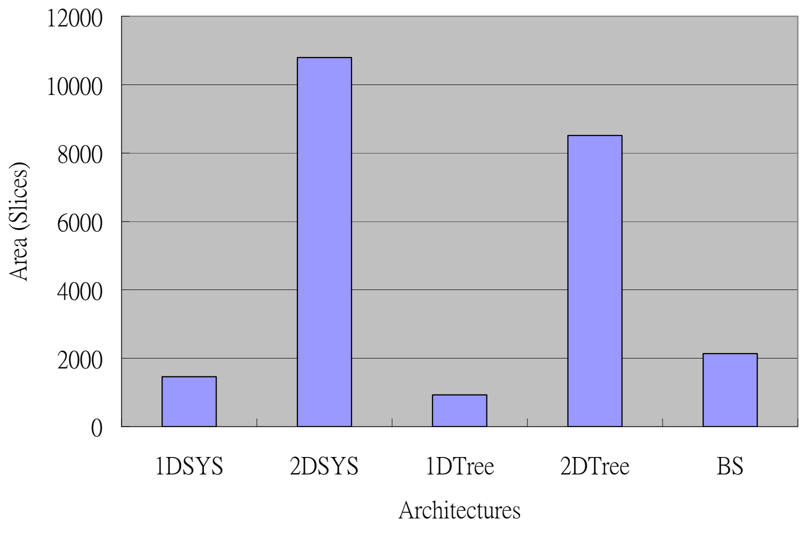

Figure 5.2: Occupied slices of different motion estimation architectures

conferencing, are suitable for 1D systolic and bit serial architectures.

5.3.2 Occupied resources

The occupied areas for different architectures are shown in figure 5.2. 2D architectures occupy the most resources, the second being the bit-serial architecture. The 1D systolic array occupies the least amount of resources. In general, the area occupied is proportional to its performance.

As area occupation directly affects the price of hardware device, suitable architecture should be selected for minimizing production cost. In modern technology, implementation of 2D architectures on FPGA is still expensive since it requires over 10k slices which is usually provided only in high-end FPGA devices. 1D and bit-serial architectures are a moderate choice for cost-constrained applications.

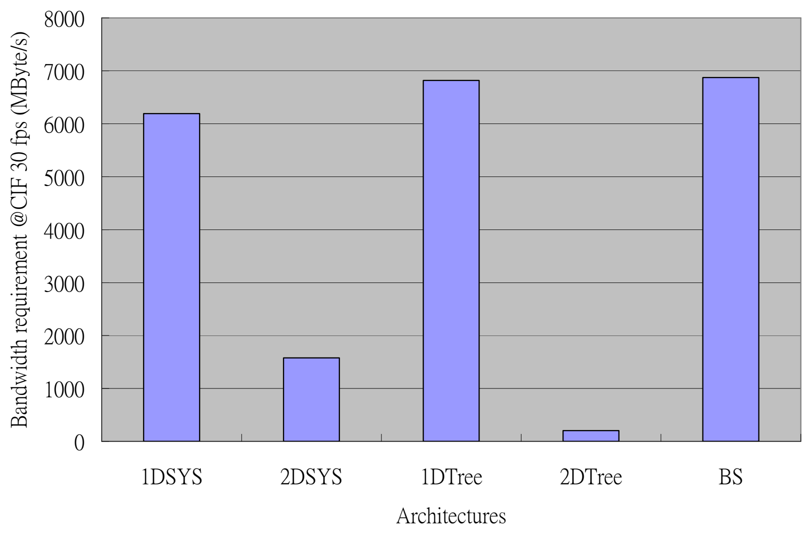

Figure 5.3: Bandwidth requirements of different motion estimation architectures at CIF 30 fps

5.3.3 Memory bandwidth

The required bandwidth for CIF 30 fps of different architectures are shown in figure 5.3. Bit-serial design has the largest bandwidth requirement, 64 bytes/cycle, inherited by its non-systolic architecture. The local communication in 1D, 2D systolic and tree architectures significantly reduce bandwidth requirements. Among 1D and 2D architectures, 1D architectures require more bandwidth.

Memory bandwidth significantly affects the power consumption. In battery-powered applications, high bandwidth architectures should be avoided. On the other hand, memory is a slow device compared to computation logic. Smaller bandwidth requirement means we could process data at a higher throughput.

5.3.4 Motion estimation algorithm

Pipelining impedes efficient processing of data dependent algorithms like fast motion estimation algorithms. The pipeline

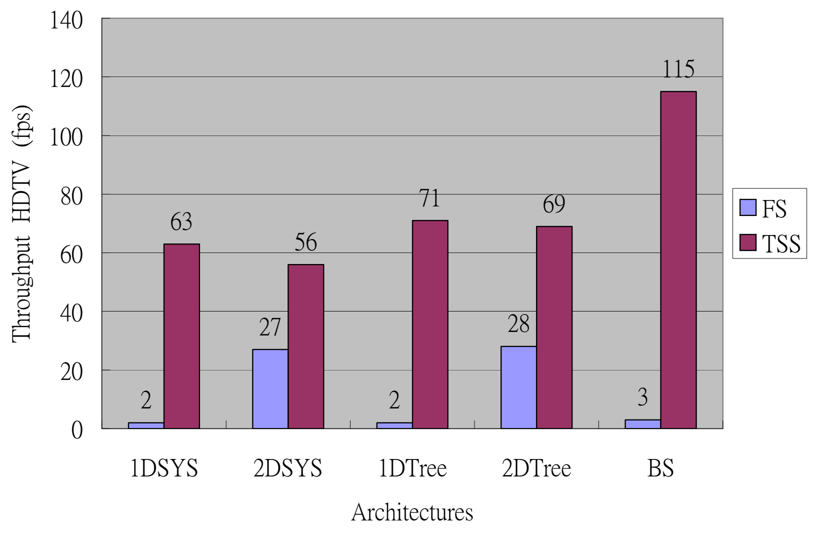

Figure 5.4: Throughput of different architectures at different motion estimation algorithms

must be flushed in the decision making process, which is a kind of data hazard. Flushing a pipeline leads to wastage of resources and execution time. In 1D or 2D systolic arrays, 16 and 32 cycles are flushed respectively when a data hazard occurs. For example, TSS in a 2D systolic array requires 448 cycles to calculate SADs for 25 search points. The number of cycles per search point is decreased from 1.017 to 17.92 cycles/search point.

In our bit-serial architecture, since it is not systolic based, the efficiency for fast algorithms can be much higher. It is able to process TSS in around 450 cycles compared to 18432 in FS. The number of cycles per search points is almost kept constant.

Figure 5.4 shows the effect of different architectures by TSS. Obviously, bit-serial design performs the best among the architectures. Figure 5.5 concludes the efficiency of different architectures per slice. FS and TSS are given for comparisons.

As a result, the algorithmic flexibility of a bit-serial design is the highest among all architectures. It gives a larger design space for algorithm designers to design any algorithms they like

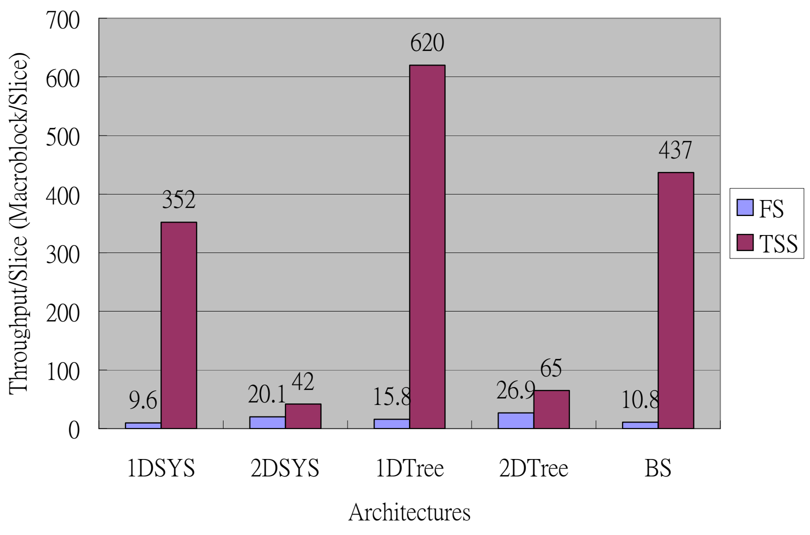

Figure 5.5: Maximum throughput per slice of different motion estimation architectures

and gets rid of hardware concern.

5.3.5 Power consumption

Power consumption is due to four main factors. The area occupied, operating frequency, bandwidth requirement, and algorithm involved. Bandwidth can be reduced by introducing more local memories, but the architecture gate count is increased. The power consumption utilizations are shown in figure 5.6. The high power consumption is due to the high frequency required by bit-serial architectures. Employing fast algorithms other than full search can greatly reduce its power consumption. Although some quality may be lost, it is acceptable in many low-end applications. As a result, bit-serial design is still energy-efficient in fast algorithms.

Since pipeline based architectures favor full search, those architectures are not a good choice for minimizing power although they require less bandwidth and operating frequency. Bit-serial

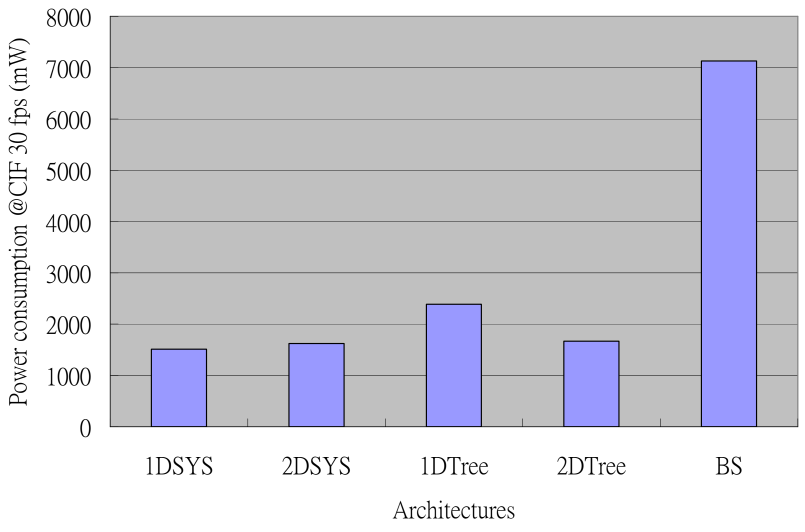

Figure 5.6: Power consumptions of different architectures

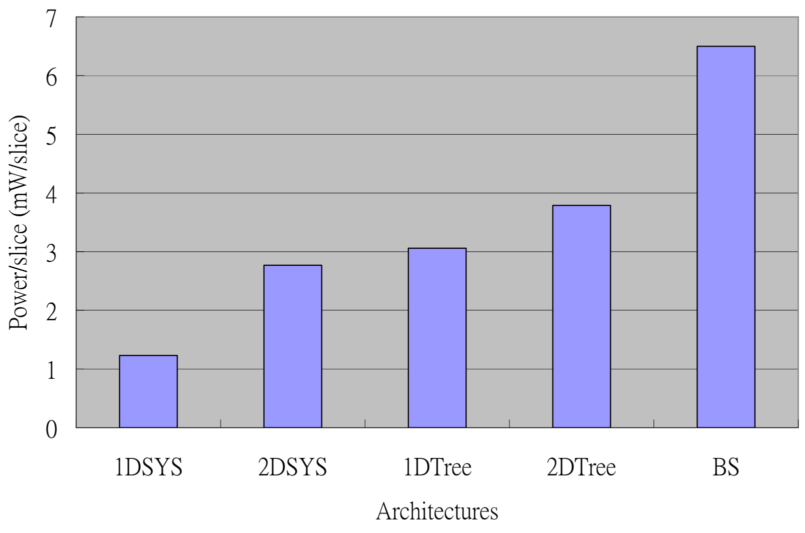

Figure 5.7: Power efficiencies of different motion estimation architectures

is a good solution as it does not discard any calculations in fast algorithms. Figure 5.7 concludes the power efficiencies of different architectures per slice.

5.4 Comparison to ASIC and FPGA architectures in past literature

In this section we compare our first-reported bit-serial architecture, the MSB-first bit-serial architecture, to previously reported FPGA or ASIC designs. The bit-serial architecture is chosen for comparisons as it is the first reported bit-serial motion estimation processor for H.264/AVC.

The processors include bit-parallel and bit-serial architectures with or without variable block size support. Since we can obtain an equivalent gate count from Xilinx ISE tools, we are able compare our architectures to ASIC implementations. Notice that the gate count collected from ISE tools likely to be is overestimated. Readers should have a sense that the equivalent gate counts collected from ASIC tools are usually smaller. Table 5.6 and table 5.7 show the comparisons.

In FPGA implementations, since variable block size is not supported in some cases, their area utilizations are underestimated. Thus, for those which don't support variable block size, their performance per slice are not used for comparison. For the remaining, our bit-serial processor outperforms other variable-block-size supported processors in performance per slice. Our architecture is operating at the highest frequency among all architectures as well because of the bit-serial design as well as the 0.13nm technology.

	[31]	[34]	[52]	[53]	[46]	[38]	[51]	[30]	Our Bit-serial
Design strategy	BP[a]	BP	BP	BP	BS	BS	BP	BP	BS[b]
Max frequency(MHz)	103.84	191	380.7	197	352	425	120	51.49	420
Area (Slices)	1654	1876	31060	1699	1510	1945	7381	9788	2133
Throughput (MB/s)	18519	4752	371513	7125	5078	17456	29296	3036	23068
Performance/Slice	11.2	2.53	12	4.19	3.36	8.97	3.97	0.31	10.8

[a]BP is an abbreviation of bit-parallel.
[b]BS is an abbreviation of bit-serial.

Table 5.6: Results and comparison of motion estimation processors on FPGA devices

	[39]	[22]	[54]	[55]	Our Bit-serial
Design strategy	BP	BP	BP	BP	BS
Num. PEs	256	256	16	16	N/A
Max frequency(MHz)	200	100	100	294	420
Area (Gate)	597k	154k	108k	61k	55k
Throughput (MB/s)	195313	97560	5560	17820	23068
Performance/gate	0.327	0.634	0.051	0.292	0.417

Table 5.7: Results and comparison of motion estimation processors on ASIC devices

In ASIC comparison, we select only H.264/AVC supported architectures for a fair comparison. The number of PEs in table 5.7 indicates what class of motion estimation processor belongs. Typically, 16-PE architectures belong to 1D class. 256-PE architectures belong to 2D class. In typical 1D implementation, performance per slice is lowest while full parallel 2D architectures obtain the highest score. Our bit-serial design scores in between and sometimes better than 2D architectures even the gate count is overestimated.

5.5 Summary

In this chapter a comparison of different architectures are presented and analyzed. We created a family of hardware supporting a range of throughput, bandwidth, area, power, and flexibility. For any application with well defined requirements, a suitable architecture can be selected. This chapter also gives an overview for designers to pick up an appropriate architecture through these results.

Chapter 6

Conclusion

6.1 Summary

In this work, we studied and analyzed the hardware architectures for motion estimation in the latest video codec standard H.264/AVC. Through algorithmic, architectural and arithmetic optimizations, we suggested and implemented a family of motion estimation processors on a FPGA platform. Modifications were made to architectures proposed in previous literature to support variable block sizes. We proposed a family of architectures with different throughputs, area utilizations, memory bandwidths, power consumptions and algorithm flexibilities. As a result, designers can select the appropriate one when all these metrics are known.

6.1.1 Algorithmic optimizations

We studied several motion estimation algorithms in the past literatures. The algorithms can be classified into two categories: exhaustive search and fast search. The former is commonly known as full search. The latter is comprised of a number of different approaches to perform motion estimation via heuristic techniques. Those studied were three step search, two-dimensional logarithmic search and diamond search which all have slight differences in their search qualities and complexities. We studied their computational requirements, searching qualities and ease of implementation in hardware. The available algorithms enable tradeoffs between the throughput and picture quality in our motion estimation processors.

Our family of motion estimation processors is able to process the motion estimation algorithms presented at near to 100% processing element utilizations. High utilizations of the systolic and tree architectures can be achieved by employing a full search. For data dependent algorithms such as TSS and DS, we employ our bit-serial architectures to achieve a high utilization ratio. As a result, our family of architectures can implement any standard of algorithm with a high utilization rate, in terms of maximizing the logic performance. This flexibility is important in many areas. Within the given quality, developers always want to achieve the highest performance via algorithm optimizations.

6.1.2 Architecture and arithmetic optimizations

It is possible to efficiently map motion estimation algorithms into systolic arrays. Through systolic arrays, we can fully parallelize the computations and reduce the required bandwidth. 1D and 2D systolic and tree architectures were presented. We also made modifications to systolic-based motion estimation hardware to support variable block size motion estimation by employing adder trees to enable the reuse of partial SADs.

At the computer arithmetic level, we study both bit-parallel and bit-serial approaches. In bit-parallel architectures, we employ conventional number systems to perform mathematical calculations. 2D systolic array and 2D tree architectures are developed for high-end applications. Small area architectures such as the 1D systolic array and 1D tree architectures are also developed to support low-end applications. We also proposed a bit-serial motion estimation processor for mid-end applications. In the bit-serial design, we redefined the SAD operations present and employed redundant number and signed digit number systems. After analyzing the properties of the comparison operation, we employ a MSB-first approach to solve motion estimation jointly with the early termination scheme. We further optimized the early termination scheme by a more accurate starting point by employing the H.264/AVC motion vector prediction technique, which can reduce the updates of minimum SAD during comparisons.

Hardware developers often search for the best tradeoffs between performance, bandwidth, area and power. Our motion estimation processors provide different characteristics in which some are performance maximized, some of area optimized, etc. With predefined constraints on hand, hardware developers are able to make tradeoffs within a short time, thus shorten the development cycle. Without these measurements, designers can only estimate the performance, area, etc based on experiences.

6.1.3 Implementation on a FPGA platform

Different architectures are synthesized, implemented and place-and-routed on Xilinx Virtex-II Pro device using Xilinx ISE and Synplicity as synthesis and implementation tools. The maximum frequency, slices occupied and power consumption are reported. In bit-parallel architectures, encoding of HDTV at 28 fps can be achieved in 2D architectures. Our bit-serial architecture can perform encoding of CIF at 58 fps. In our FPGA platform, the performance of our architectures is able to perform real time encoding up to 1080p. An area-throughput chart for architectures is shown in figure 6.1 showing different architectures mapped to different applications. The area is in terms of Xilinx Virtex-II slices and the throughput calculations assume

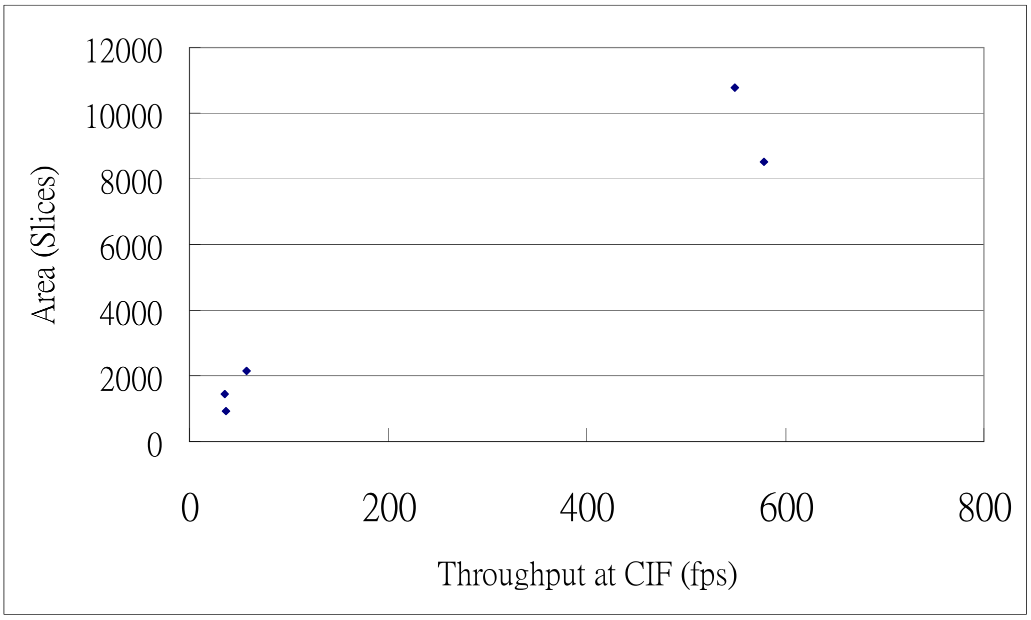

Figure 6.1: Area vs throughput in different motion estimation architectures

adequate memory bandwidth for I/O to the video coder core.

When employing general purpose processors, previous work [20] shows that a Pentium 4 2.8E GHz processor can only perform encoding of CIF video at 0.28 fps when no algorithmic optimizations are done. Motion estimation occupies 65% of the total encoding time. As a result, our designs are at least 81 times faster than motion estimation implementations on general purpose processors. The systolic, tree and bit-serial architectures proposed in this work show that an FPGA design can have much higher performance than general purpose processors. Moreover, the power consumption and memory bandwidth are reduced at the same time as the power needed for microprocessors is in range of 70 to 100 Watts (http://www.intel.com and http://www.amd.com).

In this work, we proposed the first MSB-first bit-serial variable block size motion estimation architecture for H.264/AVC. The architecture, with careful choosing of algorithms, is able to perform motion estimations efficiently on a low cost FPGA device. As an example, our bit-serial architecture is able to fit in a low cost Spartan-3 device such as XC3S200.

6.2 Future work

Video coding systems can be accelerated by hardware because of inherit opportunities for parallelism. Optimizations of video coding through software are limited since modern general purpose processors are not able to compute at several giga operations per second. Algorithms such as motion estimation can not be implemented efficiently as in general purpose processors. ASIC or FPGA technologies, making them necessary for high performance solutions.

Besides motion estimation, many algorithms in video codec can be accelerated:

1. Interpolation for fractional motion estimation that involves a large amount of pixel filtering.

2. Integer transform from residue values to transformed coefficients in the transform stage.

3. Deblocking filtering of pixels between blocks in the deblocking stage.

Furthermore, higher radix (e.g. radix-4) bit-serial implementations of motion estimation processors may have performance advantages (although at the cost of increased area) and may better exploit the 6-input LUTs in the recently announced Xilinx Virtex-5 device.

A family of hardware cores for video codecs can be built in a similar fashion to this work. Although the complexities of these stages appear smaller than that of the motion estimation, they tend to be complicated in modern and future codecs. The effort to optimize these stages will be an important topic for the future.

Appendix A

VHDL Sources

A.1 Online Full Adder

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity olFA is
    Port ( clk : in     STD_LOGIC;
           x : in    STD_LOGIC;
           y : in    STD_LOGIC;
           z : in    STD_LOGIC;
           s : out    STD_LOGIC;
           c : out    STD_LOGIC);
end olFA;

architecture Behavioral of olFA is

signal s2: std_logic;

begin

   s2 <= x xor y xor z;
        c <= (x and y) or (x and z) or (y and z);

        process(clk)
        begin
                if(clk'event and clk='1') then
```

```
                        s <= s2;
                end if;
        end process;

end Behavioral;
```

A.2 Online Signed Digit Full Adder

```
library IEEE;
use IEEE.STD˙LOGIC˙1164.ALL;
use IEEE.STD˙LOGIC˙ARITH.ALL;
use IEEE.STD˙LOGIC˙UNSIGNED.ALL;

entity olSDA is
    Port ( clk : in    STD˙LOGIC;
           a : in   STD˙LOGIC;
           b : in   STD˙LOGIC;
           c : in   STD˙LOGIC;
           d : in   STD˙LOGIC;
           neg : out   STD˙LOGIC;
           pos : out   STD˙LOGIC);
end olSDA;

architecture Behavioral of olSDA is
component olFA
Port ( clk : in    STD˙LOGIC;
                  x : in   STD˙LOGIC;
                  y : in   STD˙LOGIC;
                  z : in   STD˙LOGIC;
                  s : out   STD˙LOGIC;
                  c : out   STD˙LOGIC);
end component;

signal c1,s1: std˙logic;
signal c2,s2: std˙logic;
signal d˙p1: std˙logic;
signal b˙n: std˙logic;

begin

b˙n <= not(b);
```

```
neg <= not(c2);
pos <= s2;

u1: olFA port map (clk,a,b˙n,c,s1,c1);
u2: olFA port map (clk,c1,s1,d˙p1,s2,c2);

process (clk)
begin
    if(clk'event and clk = '1') then
             d_p1 <= not(d);

         end if;
end process;

end Behavioral;
```

A.3 Online Full Adder Tree

```
library IEEE;
use IEEE.STD˙LOGIC˙1164.ALL;
use IEEE.STD˙LOGIC˙ARITH.ALL;
use IEEE.STD˙LOGIC˙UNSIGNED.ALL;



entity olFA˙tree˙16op is
    Port ( clk : in   STD˙LOGIC;
           x1 : in  STD˙LOGIC;
           x2 : in  STD˙LOGIC;
           x3 : in  STD˙LOGIC;
           x4 : in  STD˙LOGIC;
           x5 : in  STD˙LOGIC;
           x6 : in  STD˙LOGIC;
           x7 : in  STD˙LOGIC;
           x8 : in  STD˙LOGIC;
           x9 : in  STD˙LOGIC;
           x10 : in  STD˙LOGIC;
           x11 : in  STD˙LOGIC;
           x12 : in  STD˙LOGIC;
           x13 : in  STD˙LOGIC;
```

```
            x14 : in   STD˙LOGIC;
            x15 : in   STD˙LOGIC;
            x16 : in   STD˙LOGIC;
            sum : out  STD˙LOGIC;
            carry : out   STD˙LOGIC);
end olFA˙tree˙16op;

architecture Behavioral of olFA˙tree˙16op is

signal s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14: std˙logic;
signal c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14: std˙logic;

signal s5˙p1,s5˙p2: std˙logic;
signal s12˙p1: std˙logic;
signal x16˙p1,x16˙p2,x16˙p3: std˙logic;
signal s11˙sp,s12˙sp,c11˙sp,c12˙sp: std˙logic;
signal x16˙p2˙sp: std˙logic;
signal s6˙sp,s7˙sp,s8˙sp,c6˙sp,c7˙sp,c8˙sp: std˙logic;
signal s5˙p1˙sp: std˙logic;

component olFA
Port ( clk : in    STD˙LOGIC;
                    x : in   STD˙LOGIC;
                    y : in   STD˙LOGIC;
                    z : in   STD˙LOGIC;
                    s : out  STD˙LOGIC;
                    c : out  STD˙LOGIC);
end component;


begin

-- level 1
u1: olFA port map (clk,x1,x2,x3,s1,c1);
u2: olFA port map (clk,x4,x5,x6,s2,c2);
u3: olFA port map (clk,x7,x8,x9,s3,c3);
u4: olFA port map (clk,x10,x11,x12,s4,c4);
u5: olFA port map (clk,x13,x14,x15,s5,c5);

-- level 2

-- No pipeline -------------------------------
u6: olFA port map (clk,c1,s1,c2,s6,c6);
u7: olFA port map (clk,s2,c3,s3,s7,c7);
```

```
u8: olFA port map (clk,c4,s4,c5,s8,c8);
-----------------------------------------------

-- level 3
u9: olFA port map (clk,c6,s6,c7,s9,c9);
u10: olFA port map (clk,s7,c8,s8,s10,c10);

-- level 4

-- No pipeline --------------------------------
u11: olFA port map (clk,c9,s9,c10,s11,c11);
u12: olFA port map (clk,s10,s5_p2,x16_p3,s12,c12);
-----------------------------------------------

-- level 5
u13: olFA port map (clk,c11,s11,c12,s13,c13);
-- level 6
u14: olFA port map (clk,c13,s13,s12_p1,s14,c14);

process(clk)
begin
    if(clk'event and clk = '1') then
             x16_p1 <= x16;
                  x16_p2 <= x16_p1;
                  x16_p3 <= x16_p2;
                  s12_p1 <= s12;
                  s5_p1 <= s5;
                  s5_p2 <= s5_p1;

         end if;
end process;

sum <= s14;
carry <= c14;

end Behavioral;
```

A.4 SAD merger

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD˙LOGIC˙ARITH.ALL;
use IEEE.STD˙LOGIC˙UNSIGNED.ALL;

entity olSDFA˙tree is
    Port ( clk : in   STD˙LOGIC;
           SAD˙4x4˙p : in   STD˙LOGIC˙VECTOR (15 downto 0);
           SAD˙4x4˙n : in   STD˙LOGIC˙VECTOR (15 downto 0);
           oSAD˙4x8˙p : out   STD˙LOGIC˙VECTOR (7 downto 0);
           oSAD˙4x8˙n : out   STD˙LOGIC˙VECTOR (7 downto 0);
           oSAD˙8x4˙p : out   STD˙LOGIC˙VECTOR (7 downto 0);
           oSAD˙8x4˙n : out   STD˙LOGIC˙VECTOR (7 downto 0);
           oSAD˙8x8˙p : out   STD˙LOGIC˙VECTOR (3 downto 0);
           oSAD˙8x8˙n : out   STD˙LOGIC˙VECTOR (3 downto 0);
           oSAD˙8x16˙p : out   STD˙LOGIC˙VECTOR (1 downto 0);
           oSAD˙8x16˙n : out   STD˙LOGIC˙VECTOR (1 downto 0);
           oSAD˙16x8˙p : out   STD˙LOGIC˙VECTOR (1 downto 0);
           oSAD˙16x8˙n : out   STD˙LOGIC˙VECTOR (1 downto 0);
           oSAD˙16x16˙p : out   STD˙LOGIC;
           oSAD˙16x16˙n : out   STD˙LOGIC);
end olSDFA˙tree;

architecture Behavioral of olSDFA˙tree is

component olSDA
    Port ( clk : in   STD˙LOGIC;
           a : in   STD˙LOGIC;
           b : in   STD˙LOGIC;
           c : in   STD˙LOGIC;
           d : in   STD˙LOGIC;
           neg : out   STD˙LOGIC;
           pos : out   STD˙LOGIC);
end component;

signal SAD˙8x4˙p, SAD˙8x4˙n : std˙logic˙vector(7 downto 0);
signal SAD˙4x8˙p, SAD˙4x8˙n : std˙logic˙vector(7 downto 0);
signal SAD˙8x8˙p, SAD˙8x8˙n : std˙logic˙vector(3 downto 0);
signal SAD˙8x16˙p, SAD˙8x16˙n : std˙logic˙vector(1 downto 0);
signal SAD˙16x8˙p, SAD˙16x8˙n : std˙logic˙vector(1 downto 0);

signal SAD˙4x8˙pipe˙n, SAD˙4x8˙pipe˙p: std˙logic˙vector(7 downto 0);
signal SAD˙8x8˙pipe˙n, SAD˙8x8˙pipe˙p: std˙logic˙vector(3 downto 0);
signal SAD˙8x16˙pipe˙n, SAD˙8x16˙pipe˙p: std˙logic˙vector(1 downto 0);
signal SAD˙16x8˙pipe˙n, SAD˙16x8˙pipe˙p: std˙logic˙vector(1 downto 0);
```

```
begin

-- 4x8 SAD
SAD˙4x8˙1˙2: olSDA port map (clk, SAD˙4x4˙p(0), SAD˙4x4˙n(0),
          SAD˙4x4˙p(1), SAD˙4x4˙n(1),
          SAD˙4x8˙pipe˙n(0), SAD˙4x8˙pipe˙p(0));
SAD˙4x8˙3˙4: olSDA port map (clk, SAD˙4x4˙p(2), SAD˙4x4˙n(2),
          SAD˙4x4˙p(3), SAD˙4x4˙n(3),
          SAD˙4x8˙pipe˙n(1), SAD˙4x8˙pipe˙p(1));
SAD˙4x8˙5˙6: olSDA port map (clk, SAD˙4x4˙p(4), SAD˙4x4˙n(4),
          SAD˙4x4˙p(5), SAD˙4x4˙n(5),
          SAD˙4x8˙pipe˙n(2), SAD˙4x8˙pipe˙p(2));
SAD˙4x8˙7˙8: olSDA port map (clk, SAD˙4x4˙p(6), SAD˙4x4˙n(6),
          SAD˙4x4˙p(7), SAD˙4x4˙n(7),
          SAD˙4x8˙pipe˙n(3), SAD˙4x8˙pipe˙p(3));
SAD˙4x8˙9˙10: olSDA port map (clk, SAD˙4x4˙p(8), SAD˙4x4˙n(8),
          SAD˙4x4˙p(9), SAD˙4x4˙n(9),
          SAD˙4x8˙pipe˙n(4), SAD˙4x8˙pipe˙p(4));
SAD˙4x8˙11˙12: olSDA port map (clk, SAD˙4x4˙p(10), SAD˙4x4˙n(10),
          SAD˙4x4˙p(11), SAD˙4x4˙n(11),
          SAD˙4x8˙pipe˙n(5), SAD˙4x8˙pipe˙p(5));
SAD˙4x8˙13˙14: olSDA port map (clk, SAD˙4x4˙p(12), SAD˙4x4˙n(12),
          SAD˙4x4˙p(13), SAD˙4x4˙n(13),
          SAD˙4x8˙pipe˙n(6), SAD˙4x8˙pipe˙p(6));
SAD˙4x8˙15˙16: olSDA port map (clk, SAD˙4x4˙p(14), SAD˙4x4˙n(14),
          SAD˙4x4˙p(15), SAD˙4x4˙n(15),
          SAD˙4x8˙pipe˙n(7), SAD˙4x8˙pipe˙p(7));

-- 8x4 SAD
SAD˙8x4˙1˙5: olSDA port map (clk, SAD˙4x4˙p(0), SAD˙4x4˙n(0),
          SAD˙4x4˙p(4), SAD˙4x4˙n(4),
          SAD˙8x4˙n(0), SAD˙8x4˙p(0));
SAD˙8x4˙2˙6: olSDA port map (clk, SAD˙4x4˙p(1), SAD˙4x4˙n(1),
          SAD˙4x4˙p(5), SAD˙4x4˙n(5),
          SAD˙8x4˙n(1), SAD˙8x4˙p(1));
SAD˙8x4˙3˙7: olSDA port map (clk, SAD˙4x4˙p(2), SAD˙4x4˙n(2),
          SAD˙4x4˙p(6), SAD˙4x4˙n(6),
          SAD˙8x4˙n(2), SAD˙8x4˙p(2));
SAD˙8x4˙4˙8: olSDA port map (clk, SAD˙4x4˙p(3), SAD˙4x4˙n(3),
          SAD˙4x4˙p(7), SAD˙4x4˙n(7),
          SAD˙8x4˙n(3), SAD˙8x4˙p(3));
SAD˙8x4˙9˙13: olSDA port map (clk, SAD˙4x4˙p(8), SAD˙4x4˙n(8),
          SAD˙4x4˙p(12), SAD˙4x4˙n(12),
          SAD˙8x4˙n(4), SAD˙8x4˙p(4));
```

```
SAD˙8x4˙10˙14: olSDA port map (clk, SAD˙4x4˙p(9), SAD˙4x4˙n(9),
          SAD˙4x4˙p(13), SAD˙4x4˙n(13),
          SAD˙8x4˙n(5), SAD˙8x4˙p(5));
SAD˙8x4˙11˙15: olSDA port map (clk, SAD˙4x4˙p(10), SAD˙4x4˙n(10),
          SAD˙4x4˙p(14), SAD˙4x4˙n(14),
          SAD˙8x4˙n(6), SAD˙8x4˙p(6));
SAD˙8x4˙12˙16: olSDA port map (clk, SAD˙4x4˙p(11), SAD˙4x4˙n(11),
          SAD˙4x4˙p(15), SAD˙4x4˙n(15),
          SAD˙8x4˙n(7), SAD˙8x4˙p(7));

-- 8x8 SAD
SAD˙8x8˙0: olSDA port map (clk, SAD˙4x8˙p(0), SAD˙4x8˙n(0),
          SAD˙4x8˙p(2), SAD˙4x8˙n(2),
          SAD˙8x8˙pipe˙n(0), SAD˙8x8˙pipe˙p(0));
SAD˙8x8˙1: olSDA port map (clk, SAD˙4x8˙p(1), SAD˙4x8˙n(1),
          SAD˙4x8˙p(3), SAD˙4x8˙n(3),
          SAD˙8x8˙pipe˙n(1), SAD˙8x8˙pipe˙p(1));
SAD˙8x8˙2: olSDA port map (clk, SAD˙4x8˙p(4), SAD˙4x8˙n(4),
          SAD˙4x8˙p(6), SAD˙4x8˙n(6),
          SAD˙8x8˙pipe˙n(2), SAD˙8x8˙pipe˙p(2));
SAD˙8x8˙3: olSDA port map (clk, SAD˙4x8˙p(5), SAD˙4x8˙n(5),
          SAD˙4x8˙p(7), SAD˙4x8˙n(7),
          SAD˙8x8˙pipe˙n(3), SAD˙8x8˙pipe˙p(3));

-- 8x16 SAD
SAD˙8x16˙0: olSDA port map (clk, SAD˙8x8˙p(0), SAD˙8x8˙n(0),
          SAD˙8x8˙p(1), SAD˙8x8˙n(1),
          SAD˙8x16˙pipe˙n(0),SAD˙8x16˙pipe˙p(0));
SAD˙8x16˙1: olSDA port map (clk, SAD˙8x8˙p(2), SAD˙8x8˙n(2),
          SAD˙8x8˙p(3), SAD˙8x8˙n(3),
          SAD˙8x16˙pipe˙n(1),SAD˙8x16˙pipe˙p(1));

-- 16x8 SAD
SAD˙16x8˙0: olSDA port map (clk, SAD˙8x8˙p(0), SAD˙8x8˙n(0),
          SAD˙8x8˙p(2), SAD˙8x8˙n(2),
          SAD˙16x8˙pipe˙n(0),SAD˙16x8˙pipe˙p(0));
SAD˙16x8˙1: olSDA port map (clk, SAD˙8x8˙p(1), SAD˙8x8˙n(1),
          SAD˙8x8˙p(3), SAD˙8x8˙n(3),
          SAD˙16x8˙pipe˙n(1),SAD˙16x8˙pipe˙p(1));

-- 16x16 SAD
SAD˙16x16: olSDA port map (clk, SAD˙8x16˙p(0), SAD˙8x16˙n(0),
          SAD˙8x16˙p(1), SAD˙8x16˙n(1),
          oSAD˙16x16˙n, oSAD˙16x16˙p);
```

```
process (clk)
begin
    if(clk'event and clk = '1') then
                oSAD_4x8_p <= SAD_4x8_p;
                oSAD_4x8_n <= SAD_4x8_n;
                oSAD_8x4_p <= SAD_8x4_p;
                oSAD_8x4_n <= SAD_8x4_n;
                oSAD_8x8_p <= SAD_8x8_p;
                oSAD_8x8_n <= SAD_8x8_n;
                oSAD_8x16_p <= SAD_8x16_p;
                oSAD_8x16_n <= SAD_8x16_n;
                oSAD_16x8_p <= SAD_16x8_p;
                oSAD_16x8_n <= SAD_16x8_n;

                SAD_4x8_p <= SAD_4x8_pipe_p;
                SAD_4x8_n <= SAD_4x8_pipe_n;
                SAD_8x8_p <= SAD_8x8_pipe_p;
                SAD_8x8_n <= SAD_8x8_pipe_n;
                SAD_8x16_p <= SAD_8x16_pipe_p;
                SAD_8x16_n <= SAD_8x16_pipe_n;
                SAD_16x8_p <= SAD_16x8_pipe_p;
                SAD_16x8_n <= SAD_16x8_pipe_n;
        end if;
end process;
end Behavioral;
```

A.5 Signed digit adder tree stage (top)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sd_16op_tree is
    Port ( clk : in    STD_LOGIC;
            --     rst: in STD_LOGIC;
           p : in   STD_LOGIC_VECTOR (15 downto 0);
           n : in   STD_LOGIC_VECTOR (15 downto 0);
           neg : out   STD_LOGIC;
           pos : out   STD_LOGIC);
```

```
end sd˙16op˙tree;

architecture Behavioral of sd˙16op˙tree is

component olFA˙tree˙16op
    Port ( clk : in    STD˙LOGIC;
           x1 : in   STD˙LOGIC;
           x2 : in   STD˙LOGIC;
           x3 : in   STD˙LOGIC;
           x4 : in   STD˙LOGIC;
           x5 : in   STD˙LOGIC;
           x6 : in   STD˙LOGIC;
           x7 : in   STD˙LOGIC;
           x8 : in   STD˙LOGIC;
           x9 : in   STD˙LOGIC;
           x10 : in   STD˙LOGIC;
           x11 : in   STD˙LOGIC;
           x12 : in   STD˙LOGIC;
           x13 : in   STD˙LOGIC;
           x14 : in   STD˙LOGIC;
           x15 : in   STD˙LOGIC;
           x16 : in   STD˙LOGIC;
           sum : out  STD˙LOGIC;
           carry : out   STD˙LOGIC);
end component;

component olSDA
    Port ( clk : in    STD˙LOGIC;
           a : in   STD˙LOGIC;
           b : in   STD˙LOGIC;
           c : in   STD˙LOGIC;
           d : in   STD˙LOGIC;
           neg : out  STD˙LOGIC;
           pos : out  STD˙LOGIC);
end component;

signal p˙p, n˙p: std˙logic˙vector(15 downto 0);
signal neg˙p, pos˙p: std˙logic;
signal s1,s2,c1,c2: std˙logic;

begin

tree1: olFA˙tree˙16op port map (clk,p˙p(0),p˙p(1),
            p˙p(2),p˙p(3),p˙p(4),p˙p(5),p˙p(6),p˙p(7),
             p˙p(8),p˙p(9),p˙p(10),p˙p(11),p˙p(12)
            ,p˙p(13),p˙p(14),p˙p(15),s1,c1);
tree2: olFA˙tree˙16op port map (clk,n˙p(0),n˙p(1),
             n˙p(2),n˙p(3),n˙p(4),n˙p(5),n˙p(6),n˙p(7),
             n˙p(8),n˙p(9),n˙p(10),n˙p(11),n˙p(12),
              n˙p(13),n˙p(14),n˙p(15),s2,c2);

olSDA1: olSDA port map (clk, c1,c2,s1,s2,neg˙p,pos˙p);


process(clk)
begin
    if(clk'event and clk='1') then
             p_p <= p;
                  n_p <= n;

                  neg <= neg_p;
                  pos <= pos_p;

         end if;
end process;

end Behavioral;
```

A.6 Absolute element

```
library IEEE;
use IEEE.STD˙LOGIC˙1164.ALL;
use IEEE.STD˙LOGIC˙ARITH.ALL;
use IEEE.STD˙LOGIC˙UNSIGNED.ALL;

entity abs˙stage is
    Port ( clk : in     STD˙LOGIC;
           msd : in   STD˙LOGIC;
           ref˙pixel : in       STD˙LOGIC;
           curr˙pixel : in       STD˙LOGIC;
           abs˙pos : out     STD˙LOGIC;
           abs˙neg : out     STD˙LOGIC);
end abs˙stage;

architecture Behavioral of abs˙stage is
```

```
signal msd˙pos, msd˙neg : std˙logic;
signal exchange: std˙logic;

begin

exchange <= not(msd˙pos) and msd˙neg;

process(clk)
begin
    if(clk'event and clk = '1') then
             if(msd = '1') then
                msd_pos <= curr_pixel;
                msd_neg <= ref_pixel;
                if((not(curr_pixel) and ref_pixel) = '1') then
                    abs_pos <= ref_pixel;
                    abs_neg <= curr_pixel;
                 else
                     abs_pos <= curr_pixel;
                     abs_neg <= ref_pixel;
                end if;
            else
            if(exchange = '1') then
                      abs_pos <= ref_pixel;
                      abs_neg <= curr_pixel;
           else
                      abs_pos <= curr_pixel;
                      abs_neg <= ref_pixel;
           end if;
          end if;
    end if;
end process;

end Behavioral;
```

A.7 Absolute stage (top)

```
library IEEE;
use IEEE.STD˙LOGIC˙1164.ALL;
use IEEE.STD˙LOGIC˙ARITH.ALL;
use IEEE.STD˙LOGIC˙UNSIGNED.ALL;
```

```
entity abs_stage_top is
    Port ( clk : in    STD_LOGIC;
           msd : in   STD_LOGIC;
           ref_pixel : in      STD_LOGIC_VECTOR (255 downto 0);
           curr_pixel : in      STD_LOGIC_VECTOR (255 downto 0);
           sd_num_pos : out    STD_LOGIC_VECTOR (255 downto 0);
           sd_num_neg : out    STD_LOGIC_VECTOR (255 downto 0));
end abs_stage_top;

architecture Behavioral of abs_stage_top is

component abs_stage
    Port ( clk : in    STD_LOGIC;
           msd : in   STD_LOGIC;
           ref_pixel : in      STD_LOGIC;
           curr_pixel : in      STD_LOGIC;
           abs_pos : out    STD_LOGIC;
           abs_neg : out    STD_LOGIC);
end component;


begin

abs_stage_generate: for i in 0 to 255 generate
    abs_stage: abs_stage port map (clk, msd, ref_pixel(i),
                                   curr_pixel(i), sd_num_pos(i), sd_num_neg(i));
end generate;


end Behavioral;
```

A.8 Online comparator element

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity ol_comp is
```

```
    Port ( clk : in    STD˙LOGIC;
           rst : in    STD˙LOGIC;
           p˙pos : in    STD˙LOGIC;
           p˙neg : in    STD˙LOGIC;
           q˙pos : in    STD˙LOGIC;
           q˙neg : in    STD˙LOGIC;
           result : out    STD˙LOGIC);
end ol˙comp;

architecture Behavioral of ol˙comp is

signal sign˙p, sign˙q: std˙logic;
signal mag˙p, mag˙q: std˙logic;
signal s˙p, s˙q: std˙logic˙vector(2 downto 0);

signal s˙p0, s˙p1, s˙q0, s˙q1: std˙logic;
signal s˙p2, s˙q2: std˙logic;
--signal set˙p, set˙q, reset˙p, reset˙q: std˙logic;

begin

-- initialization
sign˙p <= p˙neg;
sign˙q <= q˙neg;

mag˙p <= p˙pos or p˙neg;
mag˙q <= q˙pos or q˙neg;

-- contatentation of bit 2, bit 1 and bit 0
s˙p <= s˙p2 & s˙p1 & s˙p0;
s˙q <= s˙q2 & s˙q1 & s˙q0;

process(clk,rst)
begin
     if(rst = '1') then
              s_p0 <= '0';
                        s_q0 <= '0';
                   s_p1 <= '0';
                        s_q1 <= '0';
                        s_p2 <= '0';
                        s_q2 <= '0';
                        result <= '0';
          else
    if(clk'event and clk = '1') then
-- bit 0 is always equal to mag of p and q
        s_p0 <= mag_p;
        s_q0 <= mag_q;

-- bit 1 is always depend to bit 0 and sign
         s_p1 <= s_p0 xor sign_p;
        s_q1 <= s_q0 xor sign_q;

-- bit 1 depends on bit 0 and bit 2, bit 2 has higher priority
      if((s_p2 = '0' and s_q2 = '0') or
         (s_p2 = '1' and s_q2 = '1')) then
        s_p2 <= (not(sign_p) and s_p1) or
        (s_p1 and s_p0) or (not(s_q1 or s_q0) and sign_q);
        s_q2 <= (not(sign_q) and s_q1) or
        (s_q1 and s_q0) or (not(s_p1 or s_p0) and sign_p);
      else
        s_p2 <= (not(sign_p) and s_p2) or
        (s_p2 and s_p0) or (not(s_q2 or s_q0) and sign_q);
        s_q2 <= (not(sign_q) and s_q2) or
        (s_q2 and s_q0) or (not(s_p2 or s_p0) and sign_p);

      end if;

   if(((s_p>s_q)and(s_p-s_q >= 2)) or
      ((s_q>s_p)and(s_q-s_p>=2))) then
         result <= '1';
   else
         result <= '0';
   end if;

end if;
end if;
end process;

end Behavioral;
```

A.9 Comparator stage (top)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD˙LOGIC˙ARITH.ALL;
use IEEE.STD˙LOGIC˙UNSIGNED.ALL;

entity comp˙stage˙top is
    Port ( clk : in    STD˙LOGIC;
           rst : in    STD˙LOGIC;
           SAD˙4x4˙pos : in   STD˙LOGIC˙VECTOR (15 downto 0);
           SAD˙4x4˙neg : in   STD˙LOGIC˙VECTOR (15 downto 0);
           SAD˙4x8˙pos : in   STD˙LOGIC˙VECTOR (7 downto 0);
           SAD˙4x8˙neg : in   STD˙LOGIC˙VECTOR (7 downto 0);
           SAD˙8x4˙pos : in   STD˙LOGIC˙VECTOR (7 downto 0);
           SAD˙8x4˙neg : in   STD˙LOGIC˙VECTOR (7 downto 0);
           SAD˙8x8˙pos : in   STD˙LOGIC˙VECTOR (3 downto 0);
           SAD˙8x8˙neg : in   STD˙LOGIC˙VECTOR (3 downto 0);
           SAD˙8x16˙pos : in   STD˙LOGIC˙VECTOR (1 downto 0);
           SAD˙8x16˙neg : in   STD˙LOGIC˙VECTOR (1 downto 0);
           SAD˙16x8˙pos : in   STD˙LOGIC˙VECTOR (1 downto 0);
           SAD˙16x8˙neg : in   STD˙LOGIC˙VECTOR (1 downto 0);
           SAD˙16x16˙pos : in   STD˙LOGIC;
           SAD˙16x16˙neg : in   STD˙LOGIC;
           mv˙41˙p : out std˙logic˙vector(40 downto 0);
           mv˙41˙n : out std˙logic˙vector(40 downto 0);
           stop: out STD˙LOGIC);
end comp˙stage˙top;

architecture Behavioral of comp˙stage˙top is

component ol˙comp
    Port ( clk : in    STD˙LOGIC;
           rst : in    STD˙LOGIC;
           p˙pos : in   STD˙LOGIC;
           p˙neg : in   STD˙LOGIC;
           q˙pos : in   STD˙LOGIC;
           q˙neg : in   STD˙LOGIC;
           result : out    STD˙LOGIC);
end component;


-- = Minimum Motion vectors and temp motion vectors=

signal min˙sad˙4x4˙p, min˙sad˙4x4˙n: std˙logic˙vector(191 downto 0);
signal temp˙sad˙4x4˙p, temp˙sad˙4x4˙n: std˙logic˙vector(191 downto 0);

signal min˙sad˙4x8˙p, min˙sad˙4x8˙n: std˙logic˙vector(103 downto 0);
signal temp_sad_4x8_p, temp_sad_4x8_n: std_logic_vector(103 downto 0);
signal min_sad_8x4_p, min_sad_8x4_n: std_logic_vector(103 downto 0);
signal temp_sad_8x4_p, temp_sad_8x4_n: std_logic_vector(103 downto 0);

signal min_sad_8x8_p, min_sad_8x8_n: std_logic_vector(55 downto 0);
signal temp_sad_8x8_p, temp_sad_8x8_n: std_logic_vector(55 downto 0);

signal min_sad_8x16_p, min_sad_8x16_n: std_logic_vector(29 downto 0);
signal temp_sad_8x16_p, temp_sad_8x16_n: std_logic_vector(29 downto 0);
signal min_sad_16x8_p, min_sad_16x8_n: std_logic_vector(29 downto 0);
signal temp_sad_16x8_p, temp_sad_16x8_n: std_logic_vector(29 downto 0);

signal min_sad_16x16_p, min_sad_16x16_n: std_logic_vector(15 downto 0);
signal temp_sad_16x16_p, temp_sad_16x16_n: std_logic_vector(15 downto 0);



signal comp_result_4x4: std_logic_vector(15 downto 0);
signal comp_result_4x8: std_logic_vector(7 downto 0);
signal comp_result_8x4: std_logic_vector(7 downto 0);
signal comp_result_8x8: std_logic_vector(3 downto 0);
signal comp_result_8x16: std_logic_vector(1 downto 0);
signal comp_result_16x8: std_logic_vector(1 downto 0);
signal comp_result_16x16: std_logic;



signal sad_4x4_msd_p, sad_4x4_msd_n: std_logic_vector(15 downto 0);
signal sad_4x8_msd_p, sad_4x8_msd_n: std_logic_vector(7 downto 0);
signal sad_8x4_msd_p, sad_8x4_msd_n: std_logic_vector(7 downto 0);
signal sad_8x8_msd_p, sad_8x8_msd_n: std_logic_vector(3 downto 0);
signal sad_8x16_msd_p, sad_8x16_msd_n: std_logic_vector(1 downto 0);
signal sad_16x8_msd_p, sad_16x8_msd_n: std_logic_vector(1 downto 0);
signal sad_16x16_msd_p, sad_16x16_msd_n: std_logic;

signal cnt: std_logic_vector(3 downto 0);

signal min4x4: std_logic_vector(15 downto 0);
signal min4x8,min8x4 : std_logic_vector(7 downto 0);
signal min8x8: std_logic_vector(3 downto 0);
signal min16x8,min8x16 : std_logic_vector(1 downto 0);
signal min16x16: std_logic;

begin
```

```
mv˙41˙p <= sad˙4x4˙msd˙p & sad˙4x8˙msd˙p &
                              sad˙8x4˙msd˙p & sad˙8x8˙msd˙p &
                            sad˙8x16˙msd˙p & sad˙16x8˙msd˙p &
                          sad˙16x16˙msd˙p;
mv˙41˙n <= sad˙4x4˙msd˙n & sad˙4x8˙msd˙n &
                            sad˙8x4˙msd˙n & sad˙8x8˙msd˙n &
                          sad˙8x16˙msd˙n & sad˙16x8˙msd˙n &
                          sad˙16x16˙msd˙n;

sad4x4: for i in 0 to 15 generate
        sad˙4x4˙msd˙p(i) <= min˙sad˙4x4˙p(11+12*i)
                    when min4x4(i)='0' else
                    temp_sad_4x4_p(11+12*i);
        sad_4x4_msd_n(i) <= min_sad_4x4_n(11+12*i)
                    when min4x4(i)='0'
                    else temp_sad_4x4_n(11+12*i);
end generate;

sad4x8: for i in 0 to 7 generate
        sad_4x8_msd_p(i) <= min_sad_4x8_p(12+13*i)
                   when min4x8(i)='0'
                   else temp_sad_4x8_p(12+13*i);
        sad_4x8_msd_n(i) <= min_sad_4x8_n(12+13*i)
                   when min4x8(i)='0'
                   else temp_sad_4x8_n(12+13*i);
        sad_8x4_msd_p(i) <= min_sad_8x4_p(12+13*i)
                   when min8x4(i)='0'
                   else temp_sad_8x4_p(12+13*i);
        sad_8x4_msd_n(i) <= min_sad_8x4_n(12+13*i)
                   when min8x4(i)='0'
                   else temp_sad_8x4_n(12+13*i);
end generate;

sad8x8: for i in 0 to 3 generate
        sad_8x8_msd_p(i) <= min_sad_8x8_p(13+14*i)
                   when min8x8(i)='0'
                   else temp_sad_8x8_p(13+14*i);
        sad_8x8_msd_n(i) <= min_sad_8x8_n(13+14*i)
                   when min8x8(i)='0'
                   else temp_sad_8x8_n(13+14*i);
end generate;

sad8x16: for i in 0 to 1 generate
```

```
    sad_8x16_msd_p(i) <= min_sad_8x16_p(14+15*i)
                    when min8x16(i)='0'
                    else temp_sad_8x16_p(14+15*i);
    sad_8x16_msd_n(i) <= min_sad_8x16_n(14+15*i)
                    when min8x16(i)='0'
                    else temp_sad_8x16_n(14+15*i);

    sad_16x8_msd_p(i) <= min_sad_16x8_p(14+15*i)
                    when min16x8(i)='0'
                    else temp_sad_16x8_p(14+15*i);
    sad_16x8_msd_n(i) <= min_sad_16x8_n(14+15*i)
                    when min16x8(i)='0'
                    else temp_sad_16x8_n(14+15*i);

end generate;

sad_16x16_msd_p <= min_sad_16x16_p(15)
                    when min16x16='0'
                    else temp_sad_16x16_p(15);
sad_16x16_msd_n <= min_sad_16x16_n(15)
                    when min16x16='0'
                    else temp_sad_16x16_n(15);

ol_comp_4x4_generate:
for i in 0 to 15 generate
    ol_comp_4x4: ol_comp port map (clk,rst,SAD_4x4_pos(i),
                    SAD_4x4_neg(i),sad_4x4_msd_p(i),
                    sad_4x4_msd_n(i),comp_result_4x4(i));
end generate;


ol_comp_4x8_8x4_generate:
for i in 0 to 7 generate
    ol_comp_4x8: ol_comp port map (clk,rst, SAD_4x8_pos(i),
          SAD_4x8_neg(i),sad_4x8_msd_p(i),
          sad_4x8_msd_n(i), comp_result_4x8(i));
    ol_comp_8x4: ol_comp port map (clk,rst, SAD_8x4_pos(i),
         SAD_8x4_neg(i),sad_8x4_msd_p(i),
         sad_8x4_msd_n(i), comp_result_8x4(i));
end generate;

ol_comp_8x8_generate:
for i in 0 to 3 generate
    ol_comp_8x8: ol_comp port map (clk,rst, SAD_8x8_pos(i),
        SAD_8x8_neg(i),sad_8x8_msd_p(i),
        sad_8x8_msd_n(i), comp_result_8x8(i));
end generate;

ol_comp_16x8_8x16_generate:
for i in 0 to 1 generate
    ol_comp_16x8: ol_comp port map (clk,rst, SAD_16x8_pos(i),
        SAD_16x8_neg(i),sad_16x8_msd_p(i),
        sad_16x8_msd_n(i), comp_result_16x8(i));
    ol_comp_8x16: ol_comp port map (clk,rst, SAD_8x16_pos(i),
        SAD_8x16_neg(i),sad_8x16_msd_p(i),
         sad_8x16_msd_n(i), comp_result_8x16(i));
end generate;

ol_comp_16x16: ol_comp port map (clk, rst, SAD_16x16_pos,
        SAD_16x16_neg,sad_16x16_msd_p,
        sad_16x16_msd_n,comp_result_16x16);

stop <= comp_result_4x4(0) and comp_result_4x4(1)
         and comp_result_4x4(2) and comp_result_4x4(3)
         and comp_result_4x4(4) and comp_result_4x4(5)
        and comp_result_4x4(6) and comp_result_4x4(7)
        and comp_result_4x4(8) and comp_result_4x4(9)
        and comp_result_4x4(10) and comp_result_4x4(11)
        and comp_result_4x4(12) and comp_result_4x4(13)
         and comp_result_4x4(14) and comp_result_4x4(15)
        and comp_result_4x8(0) and comp_result_4x8(1)
         and comp_result_4x8(2) and comp_result_4x8(3)
        and comp_result_4x8(4) and comp_result_4x8(5)
         and comp_result_4x8(6) and comp_result_4x8(7)
        and comp_result_8x4(0) and comp_result_8x4(1)
         and comp_result_8x4(2) and comp_result_8x4(3)
        and comp_result_8x4(4) and comp_result_8x4(5)
         and comp_result_8x4(6) and comp_result_8x4(7)
        and comp_result_8x8(0) and comp_result_8x8(1)
        and comp_result_8x8(2) and comp_result_8x8(3)
        and comp_result_8x16(0) and comp_result_8x16(1)
         and comp_result_16x8(0) and comp_result_16x8(1)
        and comp_result_16x16;


process (clk,rst)
begin
if(rst = '1') then
```

```
          cnt <= "0000";
          min4x4 <= (others => '0');
          min4x8 <= (others => '0');
          min8x4 <= (others => '0');
          min8x8 <= (others => '0');
          min8x16 <= (others => '0');
          min16x8 <= (others => '0');
          min16x16 <= '0';

          temp_sad_4x4_p <= (others => '0');
          temp_sad_4x4_n <= (others => '0');
          temp_sad_4x8_p <= (others => '0');
          temp_sad_4x8_n <= (others => '0');
          temp_sad_8x4_p <= (others => '0');
          temp_sad_8x4_n <= (others => '0');
          temp_sad_8x8_p <= (others => '0');
          temp_sad_8x8_n <= (others => '0');
          temp_sad_8x16_p <= (others => '0');
          temp_sad_8x16_n <= (others => '0');
          temp_sad_16x8_p <= (others => '0');
          temp_sad_16x8_n <= (others => '0');
          temp_sad_16x16_p <= (others => '0');
          temp_sad_16x16_n <= (others => '0');
          min_sad_4x4_p <= (others => '1');
          min_sad_4x4_n <= (others => '1');
          min_sad_4x8_p <= (others => '1');
          min_sad_4x8_n <= (others => '1');
          min_sad_8x4_p <= (others => '1');
          min_sad_8x4_n <= (others => '1');
          min_sad_8x8_p <= (others => '1');
          min_sad_8x8_n <= (others => '1');
          min_sad_8x16_p <= (others => '1');
          min_sad_8x16_n <= (others => '1');
          min_sad_16x8_p <= (others => '1');
          min_sad_16x8_n <= (others => '1');
          min_sad_16x16_p <= (others => '1');
          min_sad_16x16_n <= (others => '1');

else
if(clk'event and clk='1') then
        cnt <= cnt + 1;

        if(cnt = "0001") then
          for i in 0 to 15 loop
```

```
            if(comp_result_4x4(i) = '1') then
                 min4x4(i) <= not(min4x4(i));
            end if;
          end loop;
        end if;

        if(cnt = "0011") then
          for i in 0 to 7 loop
            if(comp_result_4x8(i) = '1') then
              min4x8(i) <= not(min4x8(i));
            end if;
            if(comp_result_8x4(i) = '1') then
              min8x4(i) <= not(min8x4(i));
            end if;
        end loop;
end if;

if(cnt="0101") then
  for i in 0 to 3 loop
     if(comp_result_8x8(i) = '1') then
         min8x8(i) <= not(min8x8(i));
    end if;
  end loop;
end if;

if(cnt="0111") then
  for i in 0 to 1 loop
    if(comp_result_16x8(i) = '1') then
      min16x8(i) <= not(min16x8(i));
    end if;
    if(comp_result_8x16(i) = '1') then
      min8x16(i) <= not(min8x16(i));
    end if;
  end loop;
end if;

if(cnt="1001") then
    if(comp_result_16x16 = '1') then
      min16x16 <= not(min16x16);
    end if;
end if;


for i in 0 to 15 loop
```

```
   if(min4x4(i) = '0') then
       temp_sad_4x4_p(191-12*i downto 180-12*i) <=
        temp_sad_4x4_p(190-12*i downto 180-12*i) & sad_4x4_pos(i);
       temp_sad_4x4_n(191-12*i downto 180-12*i) <=
        temp_sad_4x4_n(190-12*i downto 180-12*i) & sad_4x4_neg(i);
       min_sad_4x4_p(191-12*i downto 180-12*i) <=
        min_sad_4x4_p(190-12*i downto 180-12*i)
        & temp_sad_4x4_p(191-12*i);
       min_sad_4x4_n(191-12*i downto 180-12*i) <=
        min_sad_4x4_n(190-12*i downto 180-12*i)
        & temp_sad_4x4_n(191-12*i);
  else
       min_sad_4x4_p(191-12*i downto 180-12*i) <=
        min_sad_4x4_p(190-12*i downto 180-12*i) & sad_4x4_pos(i);
       min_sad_4x4_n(191-12*i downto 180-12*i) <=
        min_sad_4x4_n(190-12*i downto 180-12*i) & sad_4x4_neg(i);
       temp_sad_4x4_p(191-12*i downto 180-12*i) <=
         temp_sad_4x4_p(190-12*i downto 180-12*i)
        & min_sad_4x4_p(191-12*i);
       temp_sad_4x4_n(191-12*i downto 180-12*i) <=
        temp_sad_4x4_n(190-12*i downto 180-12*i)
        & min_sad_4x4_n(191-12*i);
  end if;
end loop;

for i in 0 to 7 loop
  if(min4x8(i) = '0') then
     temp_sad_4x8_p(103-13*i downto 91-13*i) <=
        temp_sad_4x8_p(102-13*i downto 91-13*i) & sad_4x8_pos(i);
     temp_sad_4x8_n(103-13*i downto 91-13*i) <=
        temp_sad_4x8_n(102-13*i downto 91-13*i) & sad_4x8_neg(i);
     min_sad_4x8_p(103-13*i downto 91-13*i) <=
        min_sad_4x8_p(102-13*i downto 91-13*i)
        & temp_sad_4x8_p(103-13*i);
     min_sad_4x8_n(103-13*i downto 91-13*i) <=
        min_sad_4x8_n(102-13*i downto 91-13*i)
        & temp_sad_4x8_n(103-13*i);
  else
     min_sad_4x8_p(103-13*i downto 91-13*i) <=
        min_sad_4x8_p(102-13*i downto 91-13*i) & sad_4x8_pos(i);
     min_sad_4x8_n(103-13*i downto 91-13*i) <=
        min_sad_4x8_n(102-13*i downto 91-13*i) & sad_4x8_neg(i);
     temp_sad_4x8_p(103-13*i downto 91-13*i) <=
        temp_sad_4x8_p(102-13*i downto 91-13*i)
```

```
         & min_sad_4x8_p(103-13*i);
      temp_sad_4x8_n(103-13*i downto 91-13*i) <=
         temp_sad_4x8_n(102-13*i downto 91-13*i)
         & min_sad_4x8_n(103-13*i);
   end if;
   if(min8x4(i) = '0') then
      temp_sad_8x4_p(103-13*i downto 91-13*i) <=
         temp_sad_8x4_p(102-13*i downto 91-13*i) & sad_8x4_pos(i);
      temp_sad_8x4_n(103-13*i downto 91-13*i) <=
         temp_sad_8x4_n(102-13*i downto 91-13*i) & sad_8x4_neg(i);
      min_sad_8x4_p(103-13*i downto 91-13*i) <=
         min_sad_8x4_p(102-13*i downto 91-13*i)
         & temp_sad_8x4_p(103-13*i);
      min_sad_8x4_n(103-13*i downto 91-13*i) <=
         min_sad_8x4_n(102-13*i downto 91-13*i)
         & temp_sad_8x4_n(103-13*i);
   else
      min_sad_8x4_p(103-13*i downto 91-13*i) <=
         min_sad_8x4_p(102-13*i downto 91-13*i) & sad_8x4_pos(i);
      min_sad_8x4_n(103-13*i downto 91-13*i) <=
         min_sad_8x4_n(102-13*i downto 91-13*i) & sad_8x4_neg(i);
      temp_sad_8x4_p(103-13*i downto 91-13*i) <=
         temp_sad_8x4_p(102-13*i downto 91-13*i)
         & min_sad_8x4_p(103-13*i);
      temp_sad_8x4_n(103-13*i downto 91-13*i) <=
         temp_sad_8x4_n(102-13*i downto 91-13*i)
         & min_sad_8x4_n(103-13*i);
   end if;
end loop;

for i in 0 to 3 loop
   if(min8x8(i) = '0') then
      temp_sad_8x8_p(55-14*i downto 42-14*i) <=
        temp_sad_8x8_p(54-14*i downto 42-14*i) & sad_8x8_pos(i);
      temp_sad_8x8_n(55-14*i downto 42-14*i) <=
        temp_sad_8x8_n(54-14*i downto 42-14*i) & sad_8x8_neg(i);
      min_sad_8x8_p(55-14*i downto 42-14*i) <=
        min_sad_8x8_p(54-14*i downto 42-14*i)
        & temp_sad_8x8_p(55-14*i);
      min_sad_8x8_n(55-14*i downto 42-14*i) <=
        min_sad_8x8_n(54-14*i downto 42-14*i)
        & temp_sad_8x8_n(55-14*i);
   else
      min_sad_8x8_p(55-14*i downto 42-14*i) <=
        min_sad_8x8_p(54-14*i downto 42-14*i) & sad_8x8_pos(i);
      min_sad_8x8_n(55-14*i downto 42-14*i) <=
        min_sad_8x8_n(54-14*i downto 42-14*i) & sad_8x8_neg(i);
      temp_sad_8x8_p(55-14*i downto 42-14*i) <=
        temp_sad_8x8_p(54-14*i downto 42-14*i)
        & min_sad_8x8_p(55-14*i);
      temp_sad_8x8_n(55-14*i downto 42-14*i) <=
        temp_sad_8x8_n(54-14*i downto 42-14*i)
        & min_sad_8x8_n(55-14*i);
   end if;
end loop;

for i in 0 to 1 loop
  if(min8x16(i) = '0') then
       temp_sad_8x16_p(29-15*i downto 15-15*i) <=
        temp_sad_8x16_p(28-15*i downto 15-15*i) & sad_8x16_pos(i);
       temp_sad_8x16_n(29-15*i downto 15-15*i) <=
        temp_sad_8x16_n(28-15*i downto 15-15*i) & sad_8x16_neg(i);
       min_sad_8x16_p(29-15*i downto 15-15*i) <=
        min_sad_8x16_p(28-15*i downto 15-15*i)
        & temp_sad_8x16_p(29-15*i);
       min_sad_8x16_n(29-15*i downto 15-15*i) <=
        min_sad_8x16_n(28-15*i downto 15-15*i)
        & temp_sad_8x16_n(29-15*i);
  else
       min_sad_8x16_p(29-15*i downto 15-15*i) <=
        min_sad_8x16_p(28-15*i downto 15-15*i) & sad_8x16_pos(i);
       min_sad_8x16_n(29-15*i downto 15-15*i) <=
        min_sad_8x16_n(28-15*i downto 15-15*i) & sad_8x16_neg(i);
       temp_sad_8x16_p(29-15*i downto 15-15*i) <=
        temp_sad_8x16_p(28-15*i downto 15-15*i)
        & min_sad_8x16_p(29-15*i);
       temp_sad_8x16_n(29-15*i downto 15-15*i) <=
        temp_sad_8x16_n(28-15*i downto 15-15*i)
        & min_sad_8x16_n(29-15*i);
  end if;

  if(min16x8(i) = '0') then
       temp_sad_16x8_p(29-15*i downto 15-15*i) <=
        temp_sad_16x8_p(28-15*i downto 15-15*i) & sad_16x8_pos(i);
       temp_sad_16x8_n(29-15*i downto 15-15*i) <=
        temp_sad_16x8_n(28-15*i downto 15-15*i) & sad_16x8_neg(i);
       min_sad_16x8_p(29-15*i downto 15-15*i) <=
        min_sad_16x8_p(28-15*i downto 15-15*i)
        & temp_sad_16x8_p(29-15*i);
       min_sad_16x8_n(29-15*i downto 15-15*i) <=
        min_sad_16x8_n(28-15*i downto 15-15*i)
        & temp_sad_16x8_n(29-15*i);
  else
       min_sad_16x8_p(29-15*i downto 15-15*i) <=
        min_sad_16x8_p(28-15*i downto 15-15*i) & sad_16x8_pos(i);
       min_sad_16x8_n(29-15*i downto 15-15*i) <=
        min_sad_16x8_n(28-15*i downto 15-15*i) & sad_16x8_neg(i);
       temp_sad_16x8_p(29-15*i downto 15-15*i) <=
        temp_sad_16x8_p(28-15*i downto 15-15*i)
        & min_sad_16x8_p(29-15*i);
       temp_sad_16x8_n(29-15*i downto 15-15*i) <=
        temp_sad_16x8_n(28-15*i downto 15-15*i)
        & min_sad_16x8_n(29-15*i);
  end if;
end loop;

if(min16x16 = '0') then
       temp_sad_16x16_p(15 downto 0) <=
        temp_sad_16x16_p(14 downto 0) & sad_16x16_pos;
       temp_sad_16x16_n(15 downto 0) <=
        temp_sad_16x16_n(14 downto 0) & sad_16x16_neg;
       min_sad_16x16_p(15 downto 0) <=
        min_sad_16x16_p(14 downto 0) & temp_sad_16x16_p(15);
       min_sad_16x16_n(15 downto 0) <=
        min_sad_16x16_n(14 downto 0) & temp_sad_16x16_n(15);
else
       min_sad_16x16_p(15 downto 0) <=
         min_sad_16x16_p(14 downto 0) & sad_16x16_pos;
       min_sad_16x16_n(15 downto 0) <=
         min_sad_16x16_n(14 downto 0) & sad_16x16_neg;
       temp_sad_16x16_p(15 downto 0) <=
        temp_sad_16x16_p(14 downto 0) & min_sad_16x16_p(15);
       temp_sad_16x16_n(15 downto 0) <=
         temp_sad_16x16_n(14 downto 0) & min_sad_16x16_n(15);
end if;
end if;
end if;
end process;

end Behavioral;
```

A.10 MSB-first motion estimation processor

```
library IEEE;
use IEEE.STD˙LOGIC˙1164.ALL;
use IEEE.STD˙LOGIC˙ARITH.ALL;
use IEEE.STD˙LOGIC˙UNSIGNED.ALL;

entity MSD˙SAD˙UNIT is
    Port ( clk : in    STD˙LOGIC;
           rst : in    STD˙LOGIC;
           curr˙pixel : in      STD˙LOGIC˙VECTOR (255 downto 0);
           ref˙pixel : in      STD˙LOGIC˙VECTOR (255 downto 0);
                mv˙addr : in STD˙LOGIC˙VECTOR (5 downto 0);
                mv˙p: out STD˙LOGIC˙VECTOR (40 downto 0);
                mv˙n: out STD˙LOGIC˙VECTOR (40 downto 0);
                stop: out STD˙LOGIC);
end MSD˙SAD˙UNIT;

architecture Behavioral of MSD˙SAD˙UNIT is

component abs˙stage˙top
    Port ( clk : in    STD˙LOGIC;
           msd : in   STD˙LOGIC;
           ref˙pixel : in      STD˙LOGIC˙VECTOR (255 downto 0);
           curr˙pixel : in      STD˙LOGIC˙VECTOR (255 downto 0);
           sd˙num˙pos : out    STD˙LOGIC˙VECTOR (255 downto 0);
           sd˙num˙neg : out    STD˙LOGIC˙VECTOR (255 downto 0));
end component;

component comp˙stage˙top
    Port ( clk : in    STD˙LOGIC;
           rst : in    STD˙LOGIC;
           SAD˙4x4˙pos : in    STD˙LOGIC˙VECTOR (15 downto 0);
           SAD˙4x4˙neg : in    STD˙LOGIC˙VECTOR (15 downto 0);
           SAD˙4x8˙pos : in    STD˙LOGIC˙VECTOR (7 downto 0);
           SAD˙4x8˙neg : in    STD˙LOGIC˙VECTOR (7 downto 0);
           SAD˙8x4˙pos : in    STD˙LOGIC˙VECTOR (7 downto 0);
           SAD˙8x4˙neg : in    STD˙LOGIC˙VECTOR (7 downto 0);
           SAD˙8x8˙pos : in    STD˙LOGIC˙VECTOR (3 downto 0);
           SAD˙8x8˙neg : in    STD˙LOGIC˙VECTOR (3 downto 0);
           SAD˙8x16˙pos : in    STD˙LOGIC˙VECTOR (1 downto 0);
           SAD˙8x16˙neg : in    STD˙LOGIC˙VECTOR (1 downto 0);
           SAD˙16x8˙pos : in    STD˙LOGIC˙VECTOR (1 downto 0);
           SAD˙16x8˙neg : in    STD˙LOGIC˙VECTOR (1 downto 0);
           SAD˙16x16˙pos : in    STD˙LOGIC;
           SAD˙16x16˙neg : in    STD˙LOGIC;
           mv˙41˙p : out std˙logic˙vector(40 downto 0);
           mv˙41˙n : out std˙logic˙vector(40 downto 0);
           stop: out STD˙LOGIC);
end component;


component sd˙16op˙tree
    Port ( clk : in    STD˙LOGIC;
           p : in   STD˙LOGIC˙VECTOR (15 downto 0);
           n : in   STD˙LOGIC˙VECTOR (15 downto 0);
           neg : out   STD˙LOGIC;
           pos : out   STD˙LOGIC);
end component;

component olSDFA˙tree
    Port ( clk : in    STD˙LOGIC;
           SAD˙4x4˙p : in   STD˙LOGIC˙VECTOR (15 downto 0);
           SAD˙4x4˙n : in   STD˙LOGIC˙VECTOR (15 downto 0);
           oSAD˙4x8˙p : out   STD˙LOGIC˙VECTOR (7 downto 0);
           oSAD˙4x8˙n : out   STD˙LOGIC˙VECTOR (7 downto 0);
           oSAD˙8x4˙p : out   STD˙LOGIC˙VECTOR (7 downto 0);
           oSAD˙8x4˙n : out   STD˙LOGIC˙VECTOR (7 downto 0);
           oSAD˙8x8˙p : out   STD˙LOGIC˙VECTOR (3 downto 0);
           oSAD˙8x8˙n : out   STD˙LOGIC˙VECTOR (3 downto 0);
           oSAD˙8x16˙p : out   STD˙LOGIC˙VECTOR (1 downto 0);
           oSAD˙8x16˙n : out   STD˙LOGIC˙VECTOR (1 downto 0);
           oSAD˙16x8˙p : out   STD˙LOGIC˙VECTOR (1 downto 0);
           oSAD˙16x8˙n : out   STD˙LOGIC˙VECTOR (1 downto 0);
           oSAD˙16x16˙p : out   STD˙LOGIC;
           oSAD˙16x16˙n : out   STD˙LOGIC);
end component;

signal sd˙num˙pos, sd˙num˙neg: std˙logic˙vector(255 downto 0);
signal SAD˙4x4˙p, SAD˙4x4˙n: std˙logic˙vector(15 downto 0);
signal SAD˙8x4˙p, SAD˙8x4˙n : std˙logic˙vector(7 downto 0);
signal SAD˙4x8˙p, SAD˙4x8˙n : std˙logic˙vector(7 downto 0);
signal SAD˙8x8˙p, SAD˙8x8˙n : std˙logic˙vector(3 downto 0);
signal SAD˙8x16˙p, SAD˙8x16˙n : std˙logic˙vector(1 downto 0);
signal SAD˙16x8˙p, SAD˙16x8˙n : std˙logic˙vector(1 downto 0);
signal SAD˙16x16˙p, SAD˙16x16˙n: std˙logic;

signal msd: std˙logic;
```

```
begin

stage1˙absolute: abs˙stage˙top port map (clk, msd,
         ref˙pixel, curr˙pixel, sd˙num˙pos, sd˙num˙neg);

tree˙generate:
for i in 0 to 15 generate
    adder˙tree: sd˙16op˙tree port map (clk,
        sd˙num˙pos(16*(i+1)-1 downto 16*i),
        sd˙num˙neg(16*(i+1)-1 downto 16*i),
        SAD˙4x4˙p(i),SAD˙4x4˙n(i));
end generate;
SAD˙merger: olSDFA˙tree port map (clk,SAD˙4x4˙p,
        SAD˙4x4˙n, SAD˙4x8˙p, SAD˙4x8˙n,
        SAD˙8x4˙p, SAD˙8x4˙n, SAD˙8x8˙p,
        SAD˙8x8˙n,SAD˙8x16˙p, SAD˙8x16˙n,
        SAD˙16x8˙p, SAD˙16x8˙n,SAD˙16x16˙p,
        SAD˙16x16˙n);

stage3˙comparator: comp˙stage˙top port map (clk, rst,
        SAD˙4x4˙p, SAD˙4x4˙n, SAD˙4x8˙p, SAD˙4x8˙n,
                SAD˙8x4˙p, SAD˙8x4˙n, SAD˙8x8˙p, SAD˙8x8˙n,
        SAD˙8x16˙p,SAD˙8x16˙n, SAD˙16x8˙p,
        SAD˙16x8˙n, SAD˙16x16˙p, SAD˙16x16˙n,
         mv˙p, mv˙n, stop);

process(clk)
begin
    if(clk'event and clk = '1') then
         if(rst = '0') then
             msd <= '0';
        else
            msd <= '1';
        end if;
    end if;
end process;

end Behavioral;
```

Bibliography

[1] *Virtex-2 User Guide.* http://direct.xilinx.com/bvdocs/userguides/ug012.pdf.

[2] *Virtex-5 XtremeDSP Design Consideration.* http://direct.xilinx.com/bvdocs/userguides/ug193.pdf.

[3] Draft ITU-T Recommendation H.263, Video coding for low bit rate communication, Version 2. 1998.

[4] G. Bjontegaard and K. Lillevold. Context-adaptive VLC Coding of coefficients. In *JVT document JVT-C028*, Fairfax, May 2002.

[5] S. Bouchoux and E. Bourennane. Application based on Dynamic Reconfiguration of Field-programmable Gate Arrays: JPEG 2000 Arithmetic Decoder. *SPIE Journal in Optical Engineering*, 44(10):107001–107006, Oct. 2005.

[6] C. Y. Chen, S. Y. Chien, Y. W. Huang, T. C. Chen, T. C. Wang, and L. G. Chen. Analysis and Architecture Design of Variable Block Size Motion Estimation for H.264/AVC. *IEEE Trans. on Circuits and Systems*, 53(2):578–593, Feb. 2006.

[7] C. Y. Cho, S. Y. Huang, and J. S. Wong. An Embedded Merging Scheme for H.264/AVC Motion Estimation. In *IEEE Int. Conf. on Image Processing*, volume 3, pages 1016–1019, Sept. 2005.

[8] W. C. Chung. Implementing the H.264/AVC Video Coding Standard on FPGAs. In *Xcell publication*, pages 18–21, Sept. 2005. www.xilinx.com/publications/solguides/be_01/xc_pdf/p18-21_be1-dsp4.pdf.

[9] K. Compton and S. Hauck. Reconfigurable computing: survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[10] M. D. Ercegovac and T. Lang. *Digital Arithmetic.* Morgan Kaufmann Publishers, 2004.

[11] M. D. Erecgovac and T. Lang. On-Line Arithmetic: A Design Methodology and Applications. In *Proc. IEEE workshop. on VLSI Signal Processing*, pages 252–263, 1988.

[12] E. M. Fakhouri. Variable block-size motion estimation. citeseer.ist.psu.edu/fakhouri97variable.html.

[13] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression.* Kluwer Academic Publishers, 1992.

[14] B. Hedayati. Fpgas expand their roles as best asic replacement. http://www.xilinx-china.com/company/success/asic.htm.

[15] W. Hsu and H. Derin. Three-dimensional subband coding of video. *Proc. Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, pages 1100–1103, Apr. 1988.

[16] ISO/IEC11172. *Information technology - coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s.* (MPEG-1), 1993.

[17] ISO/IEC113818. *Information technology - generic coding of moving pictures and associated audio information.* (MPEG-2), 1995.

[18] ISO/IEC14496-2. *Amendment 1, Information technology - coding of audio-visual objects - Part 2: Visual.* 2001.

[19] ISO/IEC15444. *Information technology - JPEG2000 image coding system.* 2000.

[20] V. Iverson, J. MacVeigh, and B. Reese. Real-time H.24-AVC codec on Intel architectures. In *Image Processing, 2004. ICIP '04. 2004 International Conference on*, volume 2, pages 757–760, Oct. 2004.

[21] J. R. Jain and A. K. Jain. Displacement Measurement and its Application in Interframe Image Coding. *IEEE Trans. on Communication*, 29(12):1799–1808, Dec. 1981.

[22] M. Kim, I. Hwang, and S. I. Chae. A fast VLSI Architecture for Full-search Variable Block Size Motion Estimation in MPEG-4 AVC/H.264. In *Proc. of the ASP-DAC*, volume 1, pages 631–634, Jan. 2005.

[23] T. Koga, K. Iinuna, A. Hirano, Y. Iijima, and T. Ishiguro. Motion Compensated Interframe Coding for Video Conferencing. In *Proc. of National Telecomm. Conf*, pages G5.3.1–G5.3.5, New Orleans, Nov. 1981.

[24] T. Kormarek and P. Pirsch. Array Architectures for Block Matching Algorithms. *IEEE Trans. on Circuits and Systems*, 36(10):1301–1308, 1989.

[25] P. M. Kuhn, G. Diebel, S. Herrmann, A. Keil, H. Mooshofer, A. Kaup, R. M. Mayer, and W. Stechele. Complexity and PSNR comparison of several fast motion estimation algorithms for MPEG-4. *Proc. SPIE*, 3460:486–489, 1998.

[26] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). *Sparse Matrix Proceedings*, 1979.

[27] W. Lee, Y. Kim, R. J. Gove, and C. J. Read. Media Station 5000: Integrating Video and Audio. *IEEE Trans. Multimedia*, 1(2):50–61, 1994.

[28] M. Li. *Arithmetic and Logic in Computer Systems.* Wiley Interscience, 2004.

[29] W. Li and E. Salari. Successive Elimination Algorithm for Motion Estimation. *IEEE Trans. Image Processing*, 4(1):105–107, Jan. 1995.

[30] S. Lopez, F. Tobajas, A. Villar, V. de Armas, J. Lopez, and R. Sarmiento. Low Cost Efficient Architecture for H.264 Motion Estimation. In *Proc. of IEEE Int. Symp. on Circuits and Systems*, volume 1, pages 412–415, 2005.

[31] H. Loukil, F. Ghozzi, and A. Samet. Hardware implementation of Block Matching Algorithm with FPGA technology. In *IEEE Int. Conf. on Microelectronics*, volume 16, pages 542–546, 2004.

[32] H. S. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerosfsky. Low-complexity Transform and Quantization in H.264/AVC. *IEEE Trans. on Video Technology*, 13(7):620–644, July 2003.

[33] D. Marpe, H. Schwarz, and T. Wiegand. Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard. *IEEE Trans. on Circuits and Systems for Video Tech.*, 13(7):620–636, July 2003.

[34] M. Mohammadzadeh, M. Eshghi, and M. Azadfar. An Optimized Systolic Array Architecture for Full Search Block Matching Algorithm and its Implementation on FPGA chips. In *IEEE Int. Conf. NEWCAS*, volume 3, pages 327–330, 2005.

[35] G. E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), Apr. 1965.

[36] H. Nozaki, M. Motoyama, A. Shimbo, and S. Kawamura. Implementation of RSA Algorithm Based on RNS Montgomery Multiplication. *Lecture Notes in Computer Science*, 2162:364–376, May 2001.

[37] T. M. Oh, Y. R. Kim, W. G. Hong, and S. J. Ko. Partial Norm Based Search Algorithm for Fast Motion Estimation. *Electron. Lett.*, 36(14):1195–1196, 2000.

[38] J. Olivares and J. Hormigo. Minimum Sum of Absolute Differences Implementation in a Single FPGA Device. In *IEEE Int. Conf. on Field Programmable Logic*, pages 986–990, 2004.

[39] C. Ou, C. F. Le, and W. J. Hwang. An efficient VLSI architecture for H.264 Variable block size motion estimation. *IEEE Trans. on Signal Processing*, 51(4):1291–1299, Nov. 2005.

[40] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs.* Oxford University Press, 2000.

[41] K. R. Rao and P. Yip. *Discrete Cosine Transform.* Academic Press, 1990.

[42] I. E. G. Richardson. *H.264 and MPEG-4 Video Compression.* John Wiley Publisher, 2003.

[43] D. Salomon. *Data Compression: The Complete Reference.* Springer, 2004.

[44] M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, and F. J. Taylor, editors. *Residue number system arithmetic: modern applications in digital signal processing.* IEEE Press, Piscataway, NJ, USA, 1986.

[45] C. L. Su and C. W. Jen. Motion Estimation using On-line Arithmetic. In *Proc. of IEEE Intl. Symp. on Circuits and System*, volume 1, pages 683–686, 2000.

[46] C. L. Su and C. W. Jen. Motion Estimation using MSD-first Processing. In *Proc. of IEEE circuits, device and and systems*, volume 150, pages 124–133, Apr. 2003.

[47] P. D. Symes. *Digital Video Compression.* McGraw-Hill, 2004.

[48] T. Wieg and Ed. Pattaya. Draft ITU-T Recommendation H.264 and Draft ISO/IEC 14496-10 AVC. In *JVC of ISO/IEC and ITU-T SG16/Q.6 Doc. JVT-G050*, Mar. 2003.

[49] J. Y. Tham, S. Ranganath, M. Ranganath, and A. A. Kassim. A Novel Unrestricted Center-Biased Diamond Search Algorithm for Block Motion Estimation. *IEEE Trans. on Circuits and Systems*, 8(4):369–377, Aug. 1998.

[50] J. Villalba, J. Hormigo, J. M. prades, and E. L. Zapata. On-line Multioperand Addition Based on On-line Full Adders. In *IEEE Intl. Conf. on App. Specific systems*, pages 322–327, July 2005.

[51] C. Wei and M. Z. Gang. A novel SAD Computing Hardware Architecture for Variable-size Block Matching Estimation and Its Implementation with FPGA. In *Proc. of IEEE Int. Symp. on Circuits and Systems*, volume 1, pages 683–686, 2000.

[52] S. Wong, B. Stougie, and S. Cotofana. Alternatives in FPGA-based SAD Implementations. In *IEEE Int. Conf. on Field Programmable Logic*, pages 449–452, Dec. 2002.

[53] S. Wong, S. Vassiliadis, and S. Cotofana. A Sum of Absolute Differences Implementation in FPGA Hardware. In *Proc. of* 28th *Euromico Conf.*, pages 183–188, Sept. 2002.

[54] S. Y. Yap and J. V. McCanny. A VLSI Architecture for Advanced Video Coding Motion Estimation. In *Proc. IEEE Intl. Conf. on application-specific systems, arch., and processors*, pages 293–301, June 2003.

[55] S. Y. Yap and J. V. McCanny. A VLSI Architecture for Variable Block Size Video Motion Estimation. *IEEE Trans. on Circuits and Systems*, 51(7):384–389, July 2004.

[56] S. Zhu and K. K. Ma. A New Diamond Search Algorithm for Fast Block Matching Motion Estimation. In *Proc. of Intl. Conf. on Information Communication and Signal Processing (ICICS)*, pages 292–296, Sept. 1997.

[57] S. Zhu and K. K. Ma. A New Diamond Search Algorithm for Fast Block Matching Motion Estimation. *IEEE Trans. Image Processing*, 9(2):287–290, Feb. 2000.