A CONFIGURABLE MOTION ESTIMATION ACCELERATOR FOR VIDEO COMPRESSION

By

GOURI LANDGE
B.E. (University of Pune, India) August, 1994

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Chair, Dr. Bevan M. Baas

Member, Dr. Venkatesh Akella

Member, Dr. Rajeevan Amirtharajah

Committee in charge
2009

Abstract

The design of motion estimation accelerator (ME_ACC), a dedicated-purpose processing element, for AsAP2 platform is presented. AsAP2 platform consists of a 2-dimensional array of processing elements with a small amount of data storage. By its inherent nature, motion estimation is one of the most computationally intensive tasks in video encoding, and it also requires a significant amount of memory resources. The ME_ACC provides 4 KB dual ported SRAM to support the memory resources needed. It accelerates the motion estimation process by computing up to 16 absolute differences in one clock cycle and minimizing memory access overhead. The highly configurable ME_ACC supports programmable motion estimation parameters and search algorithms. The independently-clocked ME_ACC is the fastest programmable architecture ever fabricated (that we know of) for motion estimation, that operates at a maximum frequency of 938 MHz and occupies $0.67\,\mathrm{mm}^2$ in 65 nm CMOS technology. Multiple search algorithms are run on five benchmark video sequences using the ME_ACC. It is shown that the ME_ACC allows user to adapt search algorithm and other motion estimation parameters depending on the video characteristics in order to achieve the best trade off between the motion estimation quality and the computational complexity.

Acknowledgments

I would like to thank my adviser, Professor Bevan Baas. In addition to guiding me through this research, he has taught me lessons which I know have made me a better engineer. His willingness to spend time with me is sincerely appreciated.

I thank Professor Venkatesh Akella and Professor Rajeevan Amirtharajah for their valuable insight and their consultation. Their different perspectives helped me produce a better thesis.

University of California, Davis has provided an enriching environment to study in. They also provided me ACCEL Fellowship. I am grateful to University of California, Davis.

I would like to thank Cisco Systems for allowing me to take one year Educational Leave to complete the course work on campus.

To Dean Truong, Tinoosh Mohsenin, Christine Watnik, Prof. Zhiyi Yu and Stephen Le, thank you for all the support you have provided me when I was working off-campus on my research and thesis. You made my work and life a lot easier.

My husband, Avinash, motivated me to pursue Master's degree and provided me all the support through out the journey. Avi, thank you for always being there with me.

My and my sister's education was the top priority for my father Suresh and mother Sunanda. That has enabled me to reach so far. Words would never be sufficient to thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Video applications in various form factors ranging from (but not limited to) life-size TV to cell phone and other mobile devices have become an integral part of the modern life. These video applications have varying power budgets and video quality requirements. Plugged-in devices such as TVs can support high quality, large form factor video experiences. On the other hand, mobile devices have limited power budgets and at the same time have smaller form factor displays. The future trend of video applications is to support higher quality video experiences with reduced power budgets for longer battery life of the mobile devices. Plugged-in devices are also experiencing tighter power budgets with larger form factors and higher quality video requirements.

Digital video is a series of still digital images called *frames* which, when played back at a rate of 15 to 60 frames per second, give an illusion of a motion picture. Thus, thousands of frames are needed even for a short video clip. Video data in raw format needs an enormous amount of storage. Hence, the video data is stored and delivered in a compressed form and is decompressed at the time of playback. Neighboring frames are very similar to each other. They typically differ only by small movements of objects observed in a fraction of a second. Hence, a very good compression can be achieved by eliminating redundancy between the neighboring frames. The *motion estimation* process detects the movements of small regions of each frame and calculates motion vectors from one frame to another and uses that information to eliminate redundancy.

During the process of motion estimation, a frame to be compressed and a reference frame are divided into many non-overlapping sections called *micro-blocks*. Each micro-block from the frame to be compressed is then compared against micro-blocks in the reference frame to find the best matching micro-block in the reference frame. The computing process for motion estimation involves performing several simple arithmetic operations, such as addition, subtraction, and absolute value calculation on integer data, to compute sum of the absolute differences (SAD) between the pixel data of a micro-block from the current frame and that of a micro-block from the reference frame.

Various types of hardware platforms have been built to perform the motion estimation calculations. Application Specific Integrated Circuit (ASIC) implementations are customized to yield high performance with a specific set of motion estimation parameters and motion vector (MV) search algorithms. They provide little flexibility to adjust the motion estimation parameters according to the video characteristics for increased power efficiency. Fully programmable implementations like micro-processors provide the flexibility of MV search algorithms and other motion estimation parameters, however they can not meet the high performance requirements without high power consumption. Multi-core array platforms can support the high performance and can also achieve the high energy efficiency by adapting the motion estimation parameters according to the video characteristics. However, individual processing elements of the multi-core platforms typically have very limited local memory resources. The motion estimation process needs memory resources on the order of a few kilo-bytes. A dedicated-purpose processing element on a multi-core platform can address the diverse needs of the motion estimation process very efficiently. It can provide large memory resources, support the flexibility of programmable processors, achieve the high throughput comparable to that provided by an ASIC, and also meet a low power budget.

1.1 Project Goals

This work explores the design of a dedicated-purpose processing element for motion estimation, motion estimation accelerator (ME_ACC), in an asynchronous array of simple processors, AsAP2. The AsAP2 is a multi-core platform with 164 homogeneous processing elements [1]. Some of the key features of the AsAP2 platform are fully programmable processing elements with independent dynamic voltage and frequency scaling [2], high throughput, low area, energy efficient inter-processing element network [3], and three 16 KB on-chip shared memories [4].

The main requirements for the implementation of the ME_ACC are the following:

- support throughput for 1080p video compression for up to 30 frames per second with a micro-block size of 16x16 pixels by performing more than 13×10^6 SAD computations per second,
- support trade-offs between low power budget and extensive MV search,
- support for any MV search algorithm,
- programmable parameters such as micro-block size and MV search window size,
- seamless interfacing with neighboring homogeneous processing elements on AsAP2, and
- a low overhead Input/Output (IO) protocol to access the memory and the configuration registers in the motion estimation accelerator.

1.2 Contributions

The main contributions of this work are the following:

- A hybrid architecture involving an array of simple processors and a motion estimation accelerator, ME_ACC, is developed to perform motion vector search with high performance and high energy efficiency. The ME_ACC is a highly programmable dedicated-purpose processing element of an array of globally asynchronous locally synchronous (GALS) simple processors. It has the same IO interface as that of the other simple processing elements of the array of processors.

- A complete micro-architecture of the ME_ACC, including a simple but highly efficient IO protocol, a high performance multi-bank memory unit, and a fully pipelined SAD computation unit is designed. The micro-architecture is implemented in Register Transfer Level (RTL) using Verilog HDL. Numerous tests are developed to validate the RTL implementation.

- The ME_ACC RTL implementation is synthesized to 65 nm CMOS technology at 938 MHz frequency. The ME_ACC in the AsAP2 chip is successfully debugged and brought up in silicon.

- Multiple benchmark MV search algorithms are implemented on the ME_ACC. These algorithms are run on multiple benchmark video sequences to evaluate the performance and the merits of the programmable architecture of the ME_ACC. With micro-block size of 16x16 pixels, full search algorithm achieves significantly higher MV search quality as compared to that achieved by diamond search. However, with micro-block size of 16x16 pixels, diamond search algorithm runs ten times faster as compared to full search algorithm. The programmable architecture of the ME_ACC allows user to achieve trade off between computational efforts required and the MV search quality. It is shown that using diamond search algorithm with micro-block size of 8x8 pixels, the ME_ACC can perform MV search five times faster as compared to that using full search algorithm with micro-block size of 16x16 pixels and achieve equal or better MV search quality.

1.3 Overview

Chapter 2 introduces theory of digital video compression. Chapter 3 discusses the details of the motion estimation process. Chapter 4 presents previous work done on efficient implementations of motion estimation for H.264. In Chapter 5, the ME_ACC architecture is presented. Chapter 5 also discusses the performance analysis of the ME_ACC. Chapter 6 presents the physical data of the ME_ACC. Chapter 7 describes MATLAB model of the ME_ACC, me_acc_model. In Chapter 8, simulation results with various motion vector search algorithms on multiple benchmark video sequences are presented. Chapter 9 summarizes the contributions of this work and outlines ideas for future work.

Chapter 2

Digital Video Compression

This chapter presents an introduction to digital video compression. The overview assumes familiarity with signal processing and video processing basics. Basic concepts and methodologies of image processing can be found in any textbook on image processing, such as Gonzalez and Woods [13]. For in-depth knowledge on digital video processing, several textbooks, such as those by Tekalp [14] or Bovik [15], can be consulted. Textbooks on advanced video coding standards, such as that by Richardson and Richardson [16], present the latest standards such as MPEG-4 Visual [17] and H.264 [18].

2.1 Digital Video

The motion picture experience works on the phenomenon of persistence of vision of the human eye. A sequence of still photographs, when shown to a viewer at a sufficiently high frequency (15 to 60 frames per second), gives an illusion of watching a real movement.

Digital video is a sequence of images, each of which is a two-dimensional frame of picture elements called *pixels*. Associated with each pixel are two values: luminance (luma) and chrominance (chroma). Luminance is a value proportional to the pixel's intensity. Chrominance is a value that represents the color of the pixel. In the YCbCr format and its variations (sometimes called as YUV), luminance is given by the Y component and chrominance of the pixel is represented by the color difference components: Cb and Cr [16]. The pixel value can also be represented in other formats such as RGB, where three numerical values give the mixture of red, green and blue components of the color.

When an analog signal is digitized, it is quantized. Quantization is a process by which a continuous range of values from an input signal is divided into non-overlapping discrete ranges and each range is assigned a unique symbol. In a digital video signal, the number of bits used to represent the unique symbols is called the *pixel depth*. Thus, digital video can be characterized by a few variables:

- Frame rate: The number of frames displayed per second. The illusion of motion can be experienced at a frame rate as low as 12 frames per second. Modern cinema uses 24 frames per second, Phase Alternating Line (PAL) television uses 25 frames per second, and high-end high-definition television (HDTV) systems use 50 or 60 frames per second. Digital cameras capture video data at 30 frames per second.
- Frame dimensions: The width and height of the image expressed in the number of pixels. Digital video comparable to television requires dimensions of around 640x480 pixels, National Television System Committee (NTSC) standard-definition television (SDTV) requires dimensions of 720x480 pixels, and HDTV 1080p requires dimensions of 1920x1080 pixels.
- Pixel depth: The number of bits per pixel. Typically 16 or 24 bits are used per pixel.

2.2 Overview of Video Compression

Table 2.1 shows the number of bytes required to store raw video data for a video playback at 30 frames per second with pixel depth of 3 bytes. A huge amount of memory space will be required to store the raw video data even for a normal TV format playback for an hour. And it will need equally large bandwidth for the transmission.

A solution to this problem is to store and transmit digital video data in compressed format and decompress it at the time of playback. Fortunately, digital video data contains a lot of redundancy. A very good compression ratio can be achieved by eliminating redundancy in the raw digital video data. Redundancy exists within a frame as well as between successive frames. For example, in a picture with blue sky, many neighboring pixels have the

Table 2.1: Size of uncompressed video at 30 fps with pixel depth of 24 bits in gigabytes

Length	1080p (1920x1080 pixels)	720p (1280x720 pixels)	DVD (720x480 pixels)	Common Intermediate Format (CIF) (352x288 pixels)
1 sec	0.19	0.08	0.03	0.01
1 min	11.20	4.98	1.87	0.82
1 hour	671.85	298.60	111.97	49.27
1000 hours	671,846.40	298,598.40	111,974.40	49,268.74

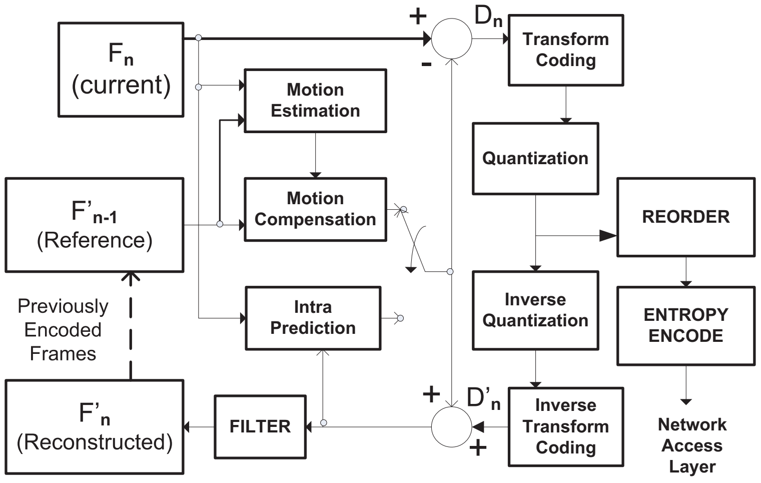


Figure 2.1: H.264 Video Encoder

same blue color (*i.e.,* same or very similar pixel values). Between two consecutive frames of a motion picture, few objects or parts of the objects move, but most of the objects and the back-ground remain the same, and this data is repeated in the two frames. The compression mechanism exploits such redundancies to minimize the size of memory space required to store digital video while minimally impacting the perceived quality of the decompressed video.

Figure 2.1 obtained from the textbook by Richardson and Richardson [16] depicts the major steps in the latest video compression standard, H.264. Description of the terms used in the Figure 2.1 is as follows.

- $\mathrm{F_n}$: Frame number n. It denotes the *current frame* to be encoded.
- $\mathrm{F'_{n-1}}$: Previously encoded and then decoded frame number $n-1$. It denotes the *reference frame*.
- $\mathrm{F'_n}$: *Reconstructed frame* number n.
- $\mathrm{D_n}$: *Difference frame*.
- $\mathrm{D'_n}$: Previously encoded and then decoded difference frame.

The following paragraphs briefly describe the major steps in the H.264 standard. Please refer to the JVT-G050 standard specification [18] for details.

A video picture frame that is being encoded is called the *current frame*. A region of a frame that is coded as a unit is called a *micro-block*.

In the first step, the current frame undergoes either *intra-frame prediction* or *inter-frame prediction*. In intra-frame prediction, micro-blocks of the current frame are predicted based on previously encoded micro-blocks of the current frame. In inter-frame prediction, micro-blocks of the current frame are predicted from previously encoded frame(s), called *reference frame(s)*, using block-based *inter-frame motion estimation* and *motion compensation*. In the process of block-based inter-frame motion estimation, a micro-block of the current frame is compared against many micro-blocks in the reference frame to find the best match. Displacement of a micro-block in the current frame with respect to the best matching block in the reference frame is called a *motion vector* (MV). Inter-frame motion estimation is followed by motion compensation, where the motion vectors (MVs) are applied to the micro-blocks of the reference frame to construct a motion compensated frame. The motion compensated frame is subtracted from the current frame to get a *residual frame*. The process of inter-frame motion estimation is described in more details in Chapter 3.

Transform coding is applied to the micro-blocks of the residual frame to reduce its entropy. The H.264 standard allows use of Hadamard transform or a Digital Cosine Transform (DCT)-based transform. The transform coding relies on a premise that pixels in an image exhibit a certain level of correlation with their neighboring pixels. These correlations can be exploited to predict the value of a pixel from its respective neighbors.

The transform coding converts the micro-block pixel information into the frequency domain where pixel correlation information is captured in a DC coefficient and pixel difference information is captured in AC coefficients. Because of the high correlation between the pixels in a micro-block, the AC coefficients normally have very small values [19].

Transform coding is followed by *quantization*. Quantization works mainly on a phenomenon that human eyes are more sensitive to lower frequencies compared to higher frequencies. Each element in a transform coded image matrix is divided by a corresponding element in a quantization matrix to throw away the least significant bits. The divisor value from the quantization matrix is called the *quantization step*. The process of quantization is irreversible and lossy. If a higher quantization step is used, fewer number of bits are required for entropy encoding. However, image quality loss is higher with a higher quantization step.

The final step in the video compression process is entropy coding. In this step, input symbols are encoded into a compressed bit stream. The input symbols may include quantized transform coefficients, MVs, and other information. The H.264 standard specifies Context Adaptive Variable Length Coding (CAVLC) and Context-based Adaptive Binary Arithmetic Coding (CABAC).

Context Adaptive Variable Length Coding is a form of Variable Length Coding (VLC). In variable length coding (VLC), the input symbols are mapped onto a series of codewords with variable length. Each symbol maps to one unique codeword. More frequently appearing (high probability) symbols are represented with short codewords and less frequently appearing symbols are represented with longer codewords, thus achieving data compression.

A disadvantage of the VLC is that, each input symbol representation needs an integral number of bits. In arithmetic coding, the probability of a symbol is represented in a fractional range over the total range of 0 to 1. Each time a symbol is encoded, the range becomes progressively smaller. At the end of a finite sequence of input symbols, we get the final fractional range. The input symbol sequence can be represented by any fractional number within the final fractional range in the form of a fixed point number. Thus, a single number is used to represent the sequence of input symbols, without having the constraint of using an integral number of bits to represent each symbol. The arithmetic coding achieves higher compression efficiency but at a much higher computational complexity.

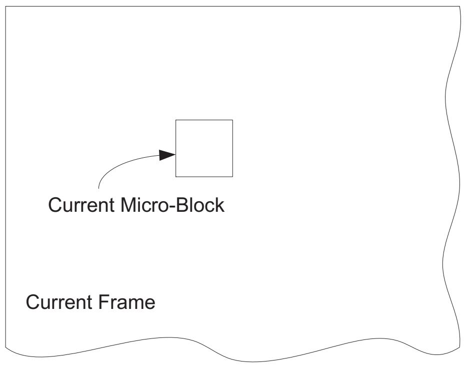
Chapter 3

Inter-frame Motion Estimation

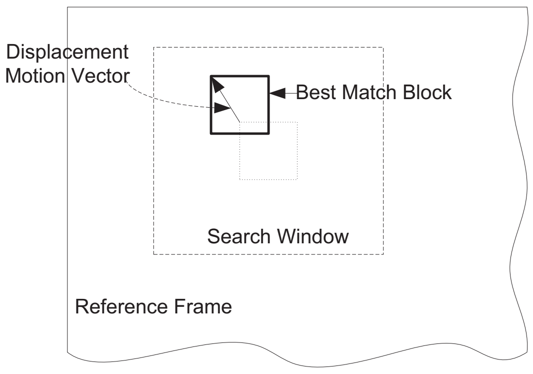
As mentioned in Chapter 2, during the process of inter-frame motion estimation, micro-blocks of the current frame are compared against the micro-blocks in the search window in the reference frame to find the best match. Figure 3.1(a) depicts a conceptual current frame with a current micro-block for motion estimation. Figure 3.1(b) shows the best matching position in the reference frame for the current micro-block in the current frame. Various block matching criteria, such as Sum of Absolute Difference (SAD), Mean Square Difference (MSD), and Pel (pixel) Difference Classification (PDC) etc., can be used to determine the best matching block. The best matching micro-block in the reference frame is subtracted from the corresponding current micro-block to produce a residual micro-block.

The better the motion estimation, the lesser the entropy of the residual micro-block and hence, better the compression efficiency. The quality of motion estimation, in terms of the entropy of the residual micro-block, depends on various factors such as search pattern used in the MV search algorithm, search window size, and micro-block size.

Of all the MV search algorithms, the full search algorithm is the most extensive one and hence, it achieves the highest quality motion estimation in the given search window. In the full search algorithm, a current micro-block from a current frame is compared against every possible position in the search window in the corresponding reference frame. Hence, the full search algorithm has the highest computational cost. Much research has been done and is still going on regarding the optimal MV search algorithm that achieves higher quality mo-

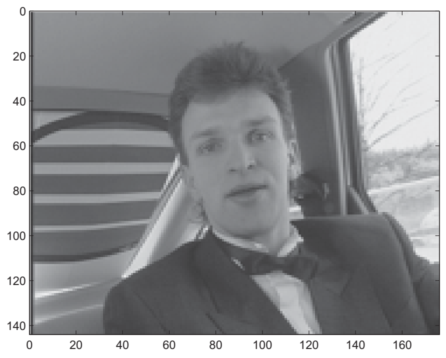


(a) Current frame with micro-block

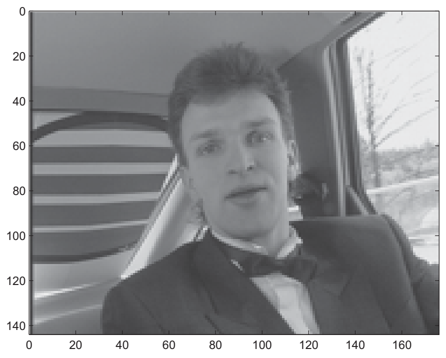


(b) Reference frame with micro-block best match and motion vector

Figure 3.1: Current and reference frames with micro-block motion vector



(a) Carphone Frame 2 (Current Frame)

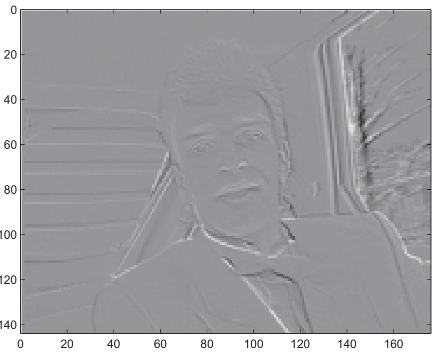


(b) Carphone Frame 1 (Reference Frame)

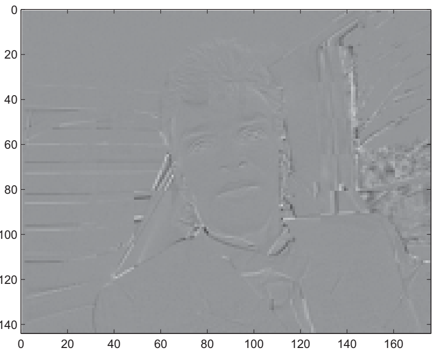
Figure 3.2: Carphone Current and reference frames

A small displacement of a tree can be seen between the two frames tion estimation with less computational complexity. The “four step search” algorithm [20] and the “diamond search” algorithm [21] are examples of some of the relatively simple, very efficient, and very widely used MV search algorithms. Zhibo Chen *et al.* proposed the “hybrid Unsymmetrical-cross Multi-Hexagon-grid Search” algorithm that efficiently performs MV search on a large search window without getting trapped into a local minima [22].

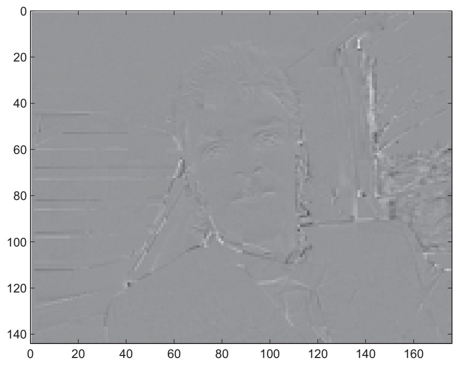
“Foreman”, “News”, and “Carphone” are some of the well known video sequences



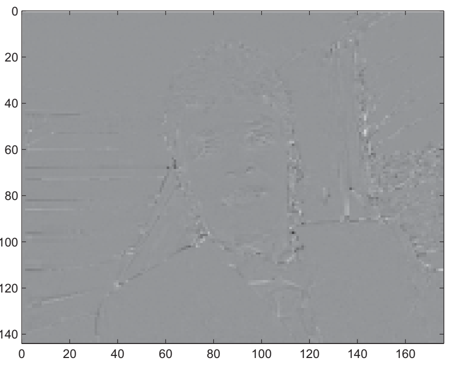
(a) *Current frame − reference frame*



(b) Residual frame with micro-block size 16x16



(c) Residual frame with micro-block size 8x8



(d) Residual frame with micro-block size 4x4

Figure 3.3: Carphone residual frames

More image details can be seen in the residual frame with micro-block size of 16x16 pixels as compared to that in the residual frames with micro-block sizes of 8x8 pixels and 4x4 pixels.

which are widely used in research on video compression. In a fast motion video sequence such as “Foreman”, positional displacements of micro-blocks from the reference frame to the current frame are larger than those in a slow motion video sequence such as “News”. Hence, a larger search window is needed to find the high quality “best” matching block for a fast motion video sequence as compared to that needed for a slow motion video sequence [23, 24]. S. Saponara and L. Fanucci have proposed a data-adaptive motion estimation algorithm, where the search window size is varied according to the video characteristics [25].

Better motion estimation is achieved with smaller micro-block sizes. The H.264 standard supports seven different sizes of micro-blocks: 16x16, 16x8, 8x16, 8x8, 8x4, 4x8, and 4x4 pixels. Figure 3.2 shows two frames from “Carphone” video sequence. Figure 3.2(a) is the reference frame used for motion estimation and Figure 3.2(b) is the current frame being encoded. The residual frames obtained by performing motion estimation using full search algorithm for the frame in Figure 3.2(b) are depicted in Figure 3.3. Figure 3.3(a) shows the difference between the reference frame in Figure 3.2(a) and the current frame in Figure 3.2(b). As can be seen from Figure 3.3(b), (c), and (d), the residual frames obtained by motion estimation using micro-block size of 16x16 pixels have higher energy than those obtained using micro-block size of 8x8 pixels and the residual frames obtained using micro-block size of 4x4 pixels have the least energy.

Chapter 4

Related Work

Recent implementations of motion estimators for modern video compression standards not only vary in terms of how much hardware and software are used but also in terms of the peak performance supported in terms of frame size, frame rate, power consumption, and Peak Signal to Noise Ratio (PSNR) quality of the decoded bit-stream. At one end of the spectrum are the ASICs designed to perform full search block motion (FSBM) estimation for few or all of the 41 micro-block partitions supported by the H.264 standard. The hardware architecture proposed by Pyen, S.M. [26] and the parameterizable hardware architecture by Nuno Roma and Leonel Sousa [27] are the examples of ASICs for FSBM estimation. These architectures exploit the highly regular data access pattern in the full search algorithm and employ large processing power to achieve real time encoding.

At the other end of the spectrum, processors with special instruction sets, Application Specific Instruction-set Processors (ASIP), have been investigated. Momcilovic, Dias, Roma, and Sousa have designed an ASIP [28] that uses specialized instructions such as *SAD16* and on-chip memory. Nikos Bellas and Malcolm Dwyer have designed a programmable vector array processor [29] with an array of Processing Elements (PEs), an on-chip memory, and a specialized instruction set.

Reconfigurable architectures also have been investigated for motion estimation. The reconfigurable Very Large Scale Integration (VLSI) architecture proposed by Cao Wei *et al.* [30] can switch between three levels of computing complexity to achieve trade off between motion estimation quality and power consumption. Liang Lu *et al.* proposed an architecture that implements the full search algorithm on a reconfigurable array of PEs. The PEs can be switched on or off as per the configuration in order to save the power and achieve performance versus power trade-off. Miguel Ribeiro and Leonel Sousa implemented a run-time reconfigurable architecture for motion estimation on a Xilinx Field-programmable Gate Array (FPGA), that supports programmable MV search algorithms [31].

Another paradigm for motion estimator implementation is a programmable architecture which, unlike targeted ASIC implementation, supports more than one search algorithms and other motion estimation parameters. Such an architecture generally uses much less computing power than a FSBM ASIC implementation, achieves similar or slightly inferior PSNR performance than a FSBM ASIC, and is much faster than a fully programmable ASIP implementation. Examples of programmable architectures are Horng-Dar *et al.* proposed architecture for hierarchical algorithms [32], generic motion estimation architecture proposed by Zhong *et al.* [33], and a patent claimed by Bellas and Dwyer on programmable motion estimation module with vector array unit [34]. Horng-Dar *et al.* implement sub-sampled scan and cluster scan MV search algorithms as tree search algorithms [32]. One dimensional (1-D) array of PEs is used in the generic motion estimator by Zhong *et al.*, to compute pixel Mean Absolute Difference (MAD) values at the search position specified by the *checking vector*. Multiple 1-D arrays of PEs can be cascaded together to increase the throughput of the motion estimator. The architecture proposed by Bellas and Dwyer [34] uses an array of PEs, a cross-bar switch, an on-chip memory, and a micro-controller that executes motion estimation specific instruction set. Mike Butts from Ambric Inc. proposed a H.264 motion estimation implementation on a Massively Parallel Processor Array (MPPA) using 322 processing objects and 188 memory objects [35].

Although this work, Motion Estimation Accelerator (ME_ACC), is also a programmable architecture, it is designed with generic motion estimation in mind rather than any specific video compression standard and it has inherent properties that distinguish itself from the mentioned programmable architectures. The details of the ME_ACC architecture are discussed in the next several chapters.

The performance of motion estimation in terms of final PSNR of the decoded bit-stream depends on many other parameters in addition to the absolute processing power (ability to perform N number of arithmetic/logic operations per second) of the motion estimator implementations. Examples of such parameters are

- optimized MV search algorithms versus full search algorithm,
- number of reference frames used, and
- micro-block sizes supported.

The efficiency of a motion estimator implementation depends on its ability to adapt one or more of these parameters according to the input video content. So, it is not a very straight forward task to compare one implementation against the other. Table 4.1 presents a comparison of the implementations discussed in terms of physical data, motion estimation parameters supported, and the peak performance achieved.

This work, ME_ACC, supports the peak performance of 210 CIF frames per second. It consumes 195 mW at the peak performance. The AsAP platform supports per-processor dynamic voltage and frequency scaling [2]. The power consumption of the ME_ACC and the energy required per workload will reduce substantially at lower supply voltages.

	Horng [32]	Zhong [33]	Roma [27]	Bellas [29]	Pyen [26]	Momcilovic [28]	This Work
Processor type	Programmable architecture	Programmable architecture	ASIC	ASIP	ASIC	ASIP	Programmable architecture
Technology	0.5 μm	N/A	0.25 μm	0.18 μm	0.18 μm	0.13 μm	65 nm
Clock speed	75 MHz	N/A	36.5 MHz	100 MHz	51 MHz	307 MHz	938 MHz
Area	8.36 mm^2	N/A	16.07 mm^2	4.21 mm^2	128K Gates 58K bit mem	0.065 mm^2	0.672 mm^2
Power at peak performance**	340* mW	N/A	N/A	N/A	N/A	8 mW	195 mW
Micro-block sizes supported	8x8, 16x16 pixels	4x4, 8x8 16x16 pixels	16x16 pixels	4x4, 4x8, 8x4, 8x8, 8x16, 16x8, 16x16 pixels	4x4, 4x8, 8x4, 8x8, 8x16, 16x8, 16x16 pixels	4x4, 4x8, 8x4, 8x8, 8x16, 16x8, 16x16 pixels	4x4, 4x8, 8x4, 8x8, 8x16, 16x8, 16x16 pixels
MV search algorithms supported	sub-sampled scan & cluster search	Programmable	Full search	Programmable	Full search	Programmable	Programmable
Peak performance**	30	30	60	30	120	45	**210**
Energy per micro-block of size 16x16 pixels	28.62 μJ	N/A	N/A	N/A	N/A	0.45 μJ	2.3 μJ

Table 4.1: Motion estimation implementations (* indicates normalized value from 300 mW at 66 MHz to 340 mW at 75 MHz. N/A indicates that data is not available. Peak performance** is normalized in terms of number of CIF frames processed per second using full search algorithm.)

Chapter 5

ME_ACC Architecture

The ME_ACC is a programmable motion estimation accelerator module that supports a multitude of MV search algorithms. The ME_ACC interfaces with neighboring processing element, called AsAP, on the AsAP2 chip. Figure 5.1 shows the interface between the ME_ACC and the neighboring AsAP elements.

Figure 5.2 illustrates constituent tasks in the MV search process. In the AsAP2 system, the task of SAD computation is performed by the ME_ACC and the task of best MV decision is handled by the controlling AsAP. Figure 5.3 indicates the distribution of the tasks involved in the MV search process between the ME_ACC and the controlling AsAP.

The ME_ACC consists of six sub-modules:

1. Configuration Registers Unit,
2. Memory Unit to hold the pixel data,
3. IO Interface Unit that communicates with the neighboring AsAPs,
4. SAD computation unit,
5. Address Generation Unit, and
6. Finite-State-Machine (FSM) that acts as the central control unit.

The Memory Unit instantiates a Reference Frame Pixel Memory that holds search window pixels from the Reference frame and a Current Frame Pixel Memory that holds

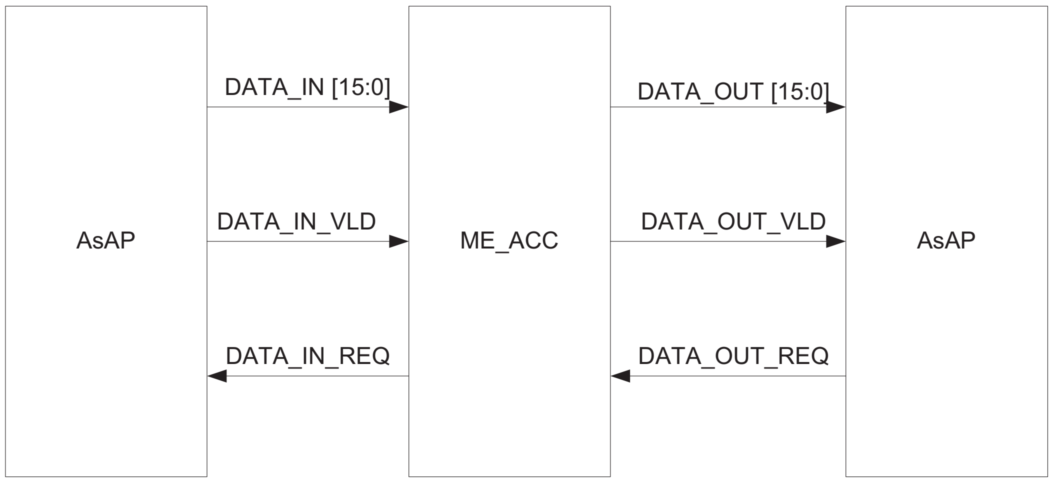


Figure 5.1: Motion estimation accelerator interface with AsAP

a pixel micro-block from the current frame. The micro-block held in the Current Frame Pixel Memory is referred to as Current Micro-block in the further discussion. The micro-block in the Reference Frame Pixel Memory, against which the Current Micro-block is being compared, is referred to as Reference Micro-block. Figure 5.4 shows the block diagram of the ME_ACC module.

Following sections describe the ME_ACC sub-units in detail.

5.1 Configuration Registers

The Configuration Registers are defined in the Verilog file me_registers.v

Table 5.1 through Table 5.11 describe the configuration registers of ME_ACC in detail. Table 5.12 lists all the configuration registers, the Reference Frame Pixel Memory, and the Current Frame Pixel Memory, and their addresses. The configuration registers are located at address from 0x8000 through 0x8556. The size of the Reference Frame Pixel Memory is 4 KB. To keep the address decoding logic simple, the Reference Frame Pixel Memory needs to be located at an offset that is a multiple of 4 K. Hence it is located at the offset that is the next multiple of 4 K, *i.e.,* 0x9000. The Reference Frame Pixel Memory address ranges from 0x9000 to 0x9FFF. The Current Frame Pixel Memory size is 256 bytes.

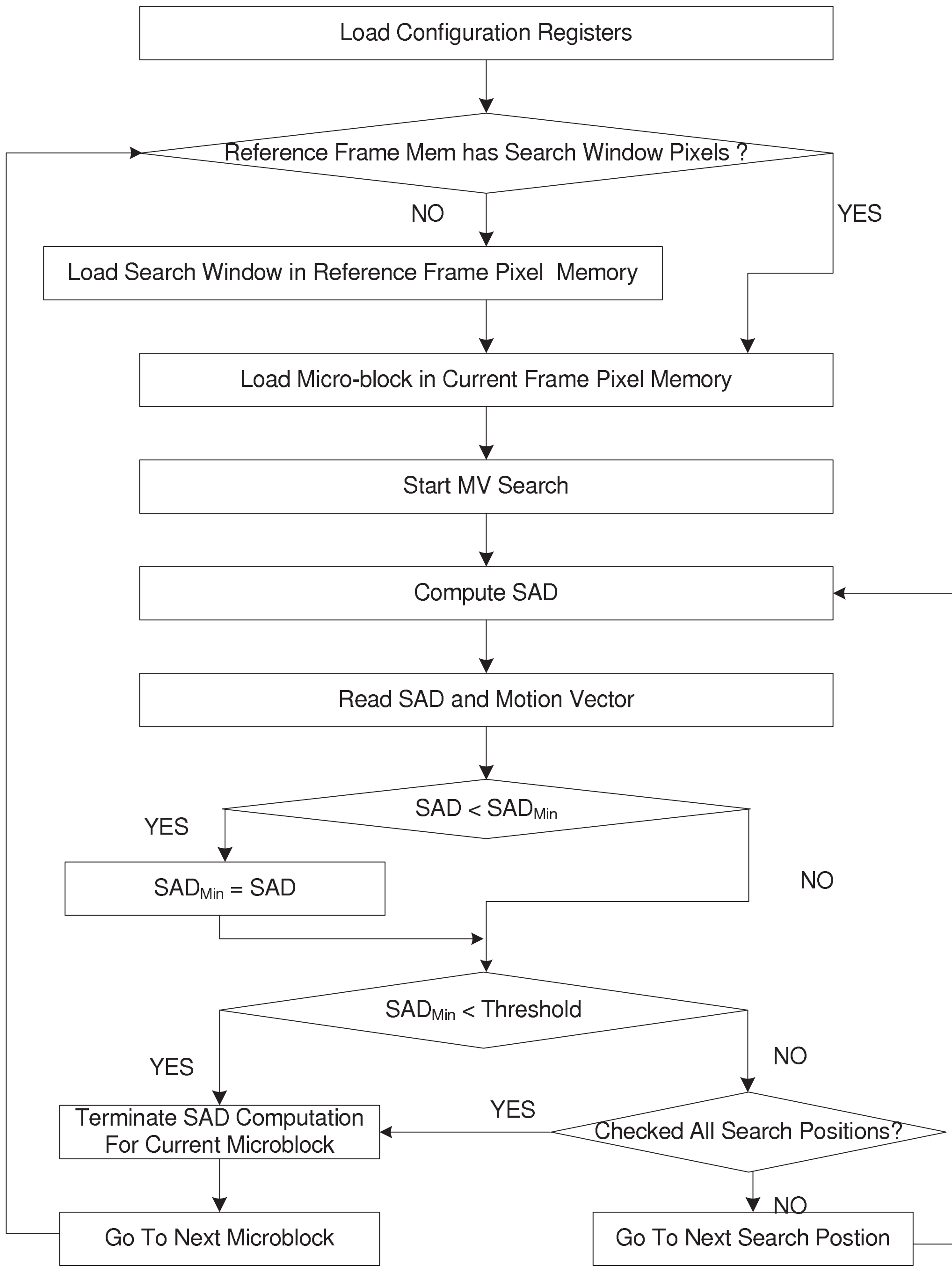

Load Configuration Registers
Reference Frame Mem has Search Window Pixels ?
NO
YES
Load Search Window in Reference Frame Pixel Memory
Load Micro-block in Current Frame Pixel Memory
Start MV Search
Compute SAD
Read SAD and Motion Vector
SAD < SAD_Min
YES
NO
SAD_Min = SAD
SAD_Min < Threshold
YES
NO
Terminate SAD Computation For Current Microblock
YES
Checked All Search Positions?
NO
Go To Next Microblock
Go To Next Search Postion

Figure 5.2: Motion vector search flow diagram

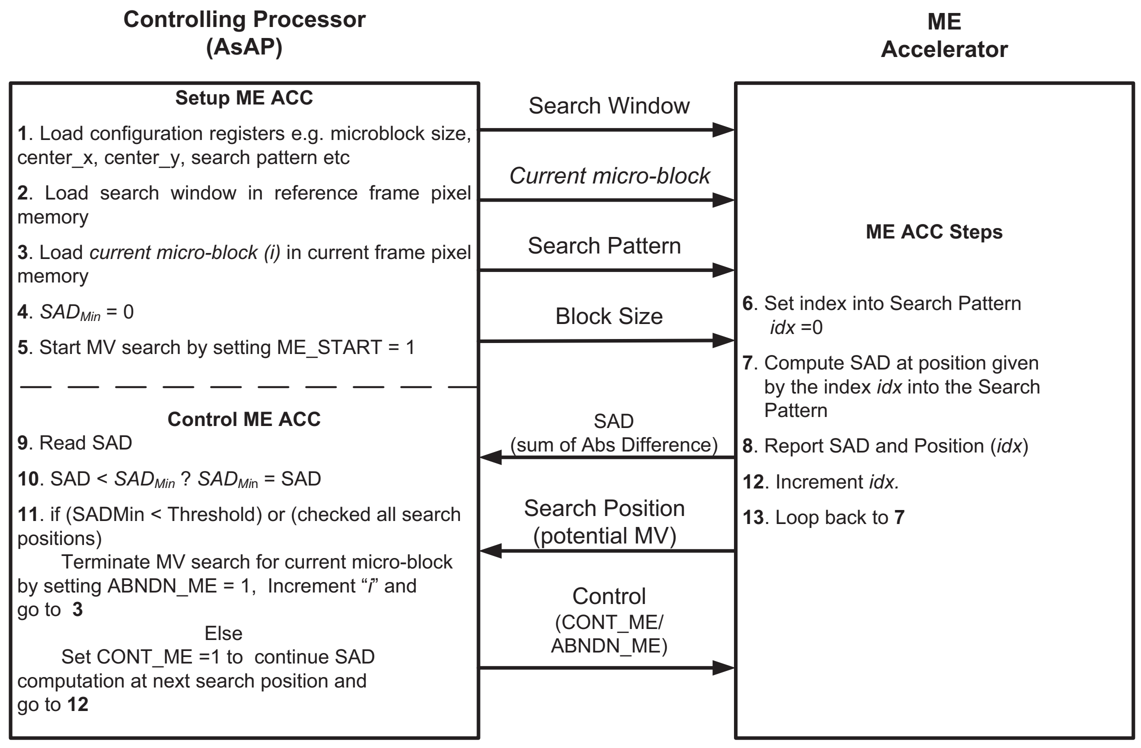

Controlling Processor (AsAP)
ME Accelerator
Setup ME ACC
1. Load configuration registers e.g. microblock size, center_x, center_y, search pattern etc
2. Load search window in reference frame pixel memory
3. Load current micro-block (i) in current frame pixel memory
4. SAD_{Min} = 0
5. Start MV search by setting ME_START = 1
Control ME ACC
9. Read SAD
10. SAD < SAD_{Min} ? SAD_{Min} = SAD
11. if (SADMin < Threshold) or (checked all search positions) Terminate MV search for current micro-block by setting ABNDN_ME = 1, Increment "i" and go to 3 Else Set CONT_ME =1 to continue SAD computation at next search position and go to 12
Search Window
Current micro-block
Search Pattern
Block Size
SAD (sum of Abs Difference)
Search Position (potential MV)
Control (CONT_ME/ ABNDN_ME)
ME ACC Steps
6. Set index into Search Pattern idx =0
7. Compute SAD at position given by the index idx into the Search Pattern
8. Report SAD and Position (idx)
12. Increment idx.
13. Loop back to 7

Figure 5.3: ME_ACC setup and best MV decision

Controlling processor sets up the ME_ACC to perform SAD computations by programming various registers and controls the ME_ACC to start, continue or stop the MV search by making the best MV decision based on the SAD value reported by the ME_ACC. The ME_ACC performs SAD computation at the given search position and increments the search position index.

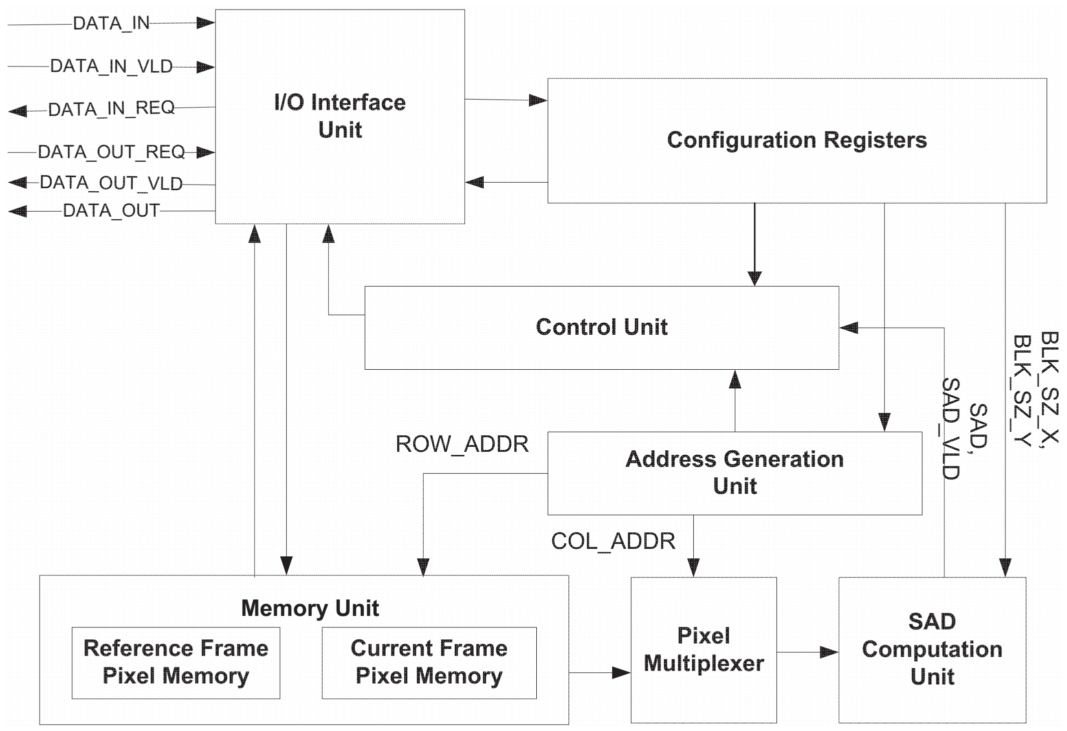


Figure 5.4: Motion estimation accelerator block diagram

Table 5.1: Address: 0x8002, Register: *BLK_SZ_X*

Bit field	Field name	Description	Valid values
4:0	*BLK_SZ_X*	Micro-block width in terms of no. of pixels	4, 8, 16
15:5	Unused		

It is located from address 0xA000 to 0xA0FF.

The usage of these registers is described in further sections.

5.1.1 Micro-block Size

Registers *BLK_SZ_X* and *BLK_SZ_Y* hold the micro-block horizontal and vertical dimensions respectively in terms of number of pixels. With the current implementation, only three values are valid for micro-block dimensions. To maintain the simplicity of user programming, these registers do not use encoded values. Table 5.1 describes the register *BLK_SZ_X* and Table 5.2 describes the register *BLK_SZ_Y*.

Table 5.2: Address: 0x8004, Register: *BLK_SZ_Y*

Bit field	Field name	Description	Valid values
4:0	*BLK_SZ_Y*	Micro-block height in terms of no. of pixels	4, 8, 16
15:5	Unused		

5.1.2 Current Micro-block Top-Left Corner Pixel Coordinates

Register *CENTER_XY* specifies the Reference Frame Pixel Memory column and row numbers that correspond to the top-left corner of the Current Micro-block. The register *CENTER_XY* is described in Table 5.3.

Often times, the neighboring micro-blocks of a current frame have the same or very similar MVs. Registers *MVPRED_X* and *MVPRED_Y* can be programmed with MV of the previous micro-block to increase the performance of the MV search process. The register *MVPRED_X* is added to the register field *CENTER_X* to get *predicted_center_x*. The register *MVPRED_Y* is added to the register field *CENTER_Y* to get *predicted_center_y*. The *predicted_center_x* and the *predicted_center_y* are added to the search position coordinates *SRCH_PTRN_n_X[i]* and *SRCH_PTRN_n_Y[i]* respectively to get the micro-block position for MV search. The register *MVPRED_X* is described in Table 5.4 and the register *MVPRED_Y* is described in Table 5.5.

5.1.3 Search Pattern Definitions

The ME_ACC supports four programmable search patterns. Each pattern can have up to 64 search positions. A search position is specified by a set of 2 registers, x-coordinate register *SRCH_PTRN_n_X[i]* and y-coordinate register *SRCH_PTRN_n_Y[i]*, where n indicates the search pattern number and i gives the index of the search position in the selected search pattern.

Table 5.6 lists the x-coordinate registers for the four search patterns. The 6 bits field, *SRCH_PTRN_X*, in each of these registers holds a x-coordinate value for a search position. This value is added to the *predicted_center_x* to get the Reference Frame Pixel Memory column index for the corresponding search position. Valid values for the *SRCH_PTRN_X*

Table 5.3: Address: 0x8552, Register: *CENTER_XY*

Bit field	Field name	Description	Valid values
5:0	*CENTER_X*	Pixel column number of the Reference Frame Pixel Memory that corresponds with the Current Micro-block's top-left pixel column in the current frame	0 to 63
7:6	Unused		
13:8	*CENTER_Y*	Pixel row number of the Reference Frame Pixel Memory that corresponds with the Current Micro-block's top-left pixel row in the current frame	0 to 63
15:14	Unused		

Table 5.4: Address: 0x8010, Register: *MVPRED_X*

Bit field	Field name	Description	Valid values
5:0	*MVPRED_X*	X-axis motion vector predictor This offset is added to *CENTER_X* to get the *predicted center_x* in the reference frame search window	-32 to $+31$. 2's complement
15:6	Unused		

Table 5.5: Address: 0x8020, Register: *MVPRED_Y*

Bit field	Field name	Description	Valid values
5:0	*MVPRED_Y*	Y-axis motion vector predictor This offset is added to *CENTER_Y* to get the *predicted center_y* in the reference frame search window	-32 to $+31$, 2's complement
15:6	Unused		

Table 5.6: Search pattern x-coordinate registers

Address range	Register set name	Register set size	Bit field	Field name
0x8100-0x813F	*SRCH_PTRN_0_X*	64	5:0	*SRCH_PTRN_X*
			15:6	Unused
0x8140-0x817F	*SRCH_PTRN_1_X*	64	5:0	*SRCH_PTRN_X*
			15:6	Unused
0x8180-0x81BF	*SRCH_PTRN_2_X*	64	5:0	*SRCH_PTRN_X*
			15:6	Unused
0x81C0-0x81FF	*SRCH_PTRN_3_X*	64	5:0	*SRCH_PTRN_X*
			15:6	Unused

Table 5.7: Search pattern y-coordinate registers

Address range	Register set name	Register set size	Bit field	Field name
0x8300-0x833F	*SRCH_PTRN_0_Y*	64	5:0	*SRCH_PTRN_Y*
			15:6	Unused
0x8340-0x837F	*SRCH_PTRN_1_Y*	64	5:0	*SRCH_PTRN_Y*
			15:6	Unused
0x8380-0x83BF	*SRCH_PTRN_2_Y*	64	5:0	*SRCH_PTRN_Y*
			15:6	Unused
0x83C0-0x83FF	*SRCH_PTRN_3_Y*	64	5:0	*SRCH_PTRN_Y*
			15:6	Unused

field range from -32 to +31, represented in 2's complement form.

Table 5.7 lists the y-coordinate registers for the four search patterns. The 6 bits field, *SRCH_PTRN_Y*, in each of these registers holds a y-coordinate value for a search position. This value is added to the *predicted_center_y* to get the Reference Frame Pixel Memory row index for the corresponding search position. Valid values for the *SRCH_PTRN_Y* field range from -32 to +31, represented in 2's complement form.

5.1.4 Search Pattern Selection

One of the four search patterns is selected by register *SRCH_PTRN_CNT*. Table 5.8 gives the bit field description of the register *SRCH_PTRN_CNT*.

Table 5.8: Address: 0x8540, Register: *SRCH_PTRN_CNT*

Bit Field	Field name	Description	Valid values
2:0	*SRCH_PTRN_CNT*	Selects one of the 4 search patterns.	0, 1, 2, 3
15:3	Unused		

Table 5.9: Address: 0x8550, Register : *START_ME*

Bit Field	Field name	Description	Valid values
0	*START_ME*	If *START_ME* = “1” Start SAD computation at the search position given by *SRCH_PTRN_n_X[0]*, *SRCH_PTRN_n_Y[0]*, where n is given by *SRCH_PTRN_CNT*. If *START_ME* = “0” Unused	1, 0
15:1	Unused		

5.1.5 ME_ACC Control Registers

Three registers, *START_ME*, *CONT_ME*, and *ABNDN_ME*, are provided to control the operation of the ME_ACC. Table 5.9 describes the register *START_ME*, Table 5.10 describes the register *CONT_ME*, and Table 5.11 describes the register *ABNDN_ME*.

Table 5.10: Address: 0x8554, Register: *CONT_ME*

Bit Field	Field name	Description	Valid values
0	*CONT_ME*	If *CONT_ME* = “1” Increment the search position index to the selected *SRCH_PTRN* and continue the SAD computation. If *CONT_ME* = “0” Unused	1, 0
15:1	Unused		

Table 5.11: Address: 0x8556, Register: *ABNDN_ME*

Bit Field	Field name	Description	Valid values
0	*ABNDN_ME*	If *ABNDN_ME* = “1” Reset the search position index to 0 and terminate the SAD computation. If *ABNDN_ME* = “0” Unused	1, 0
15:1	Unused		

Table 5.12: Register List. The Configuration registers range from address 0x8000 to address 0x8556, Reference Frame Pixel Memory ranges from address 0x9000 to address 0x9FFF, and Current Frame Pixel Memory ranges from address 0xA000 to address 0xA0FF.

Address	Name
0x8000	Unused
0x8002	*BLK_SZ_X*
0x8004	*BLK_SZ_Y*
0x8006-0x800F	Unused
0x8010	*MVPRED_X*
0x8012-0x801F	Unused
0x8020	*MVPRED_Y*
0x8022-0x80FF	Unused
0x8100-0x813F	*SRCH_PTRN_0_X*[0:63]
0x8140-0x817F	*SRCH_PTRN_1_X*[0:63]
0x8180-0x81BF	*SRCH_PTRN_2_X*[0:63]
0x81C0-0x81FF	*SRCH_PTRN_3_X*[0:63]
0x8200-0x82FF	Unused
0x8300-0x833F	*SRCH_PTRN_0_Y*[0:63]
0x8340-0x837F	*SRCH_PTRN_1_Y*[0:63]
0x8380-0x83BF	*SRCH_PTRN_2_Y*[0:63]
0x83C0-0x83FF	*SRCH_PTRN_3_Y*[0:63]
0x8400-0x843F	Unused
0x8540	*SRCH_PTRN_CNT*

Table 5.12: Register List. The Configuration registers range from address 0x8000 to address 0x8556, Reference Frame Pixel Memory ranges from address 0x9000 to address 0x9FFF, and Current Frame Pixel Memory ranges from address 0xA000 to address 0xA0FF.

Address	Name
0x8542-0x854F	Unused
0x8550	*START_ME*
0x8552	*CENTER_XY*
0x8554	*CONT_ME*
0x8556	*ABNDN_ME*
0x9000-0x9FFF	*Reference_Frame_Pixel_Memory*
0xA000-0xA0FF	*Current_Frame_Pixel_Memory*

5.2 Memory Unit

The Memory Unit instantiates Reference Frame Pixel Memory and Current Frame Pixel Memory. This module is implemented in the Verilog file me_mem.v

The Reference Frame Pixel Memory comprises 8 banks, refblk_bnk[i], where i, the *Bank Index*, ranges from 0 to 7. Each bank is a dual-ported, 64 bits wide, and 64 rows (memory locations) deep SRAM. Each refblk_bnk[i] holds 8 pixel columns $(i \times 8)$ through $((i \times 8) + 7)$ of the search window. In this design, a pixel is a 8 bits value. Hence, each row (memory location) in a given bank holds 8 consecutive pixels in a row of the reference frame.

The Current Frame Pixel Memory comprises 2 banks, orgblk_bnk[i], where i, the *Bank Index*, ranges from 0 to 1. Each bank is a dual-ported, 64 bits wide, and 16 rows (memory locations) deep SRAM. Each orgblk_bnk[i], holds 8 pixel columns $(i \times 8)$ through $((i \times 8) + 7)$. Each row (memory location) in a given bank holds 8 consecutive pixels in a row of the current frame.

Figure 5.5 shows logical and physical addressing of the Reference Frame Pixel Memory. Physical address of a byte or a pixel, *Ref_Mem_Byte_Addr*, in the Reference Frame Pixel Memory is given by Equation 5.1.

$$Ref\_Mem\_Byte\_Addr = 0x9000 + (i \times 512) + (j \times 8) + k \tag{5.1}$$

, where i is the *Bank Index*, j is the *Row Index*, and k is the *Byte Offset* in the given memory location.

Figure 5.6 shows logical and physical addressing of the Current Frame Pixel Memory. Physical address of a byte or a pixel, *Curr_Mem_Byte_Addr*, in the Current Frame Pixel Memory is given by Equation 5.2.

$$Curr\_Mem\_Byte\_Addr = 0xA000 + (i \times 64) + (j \times 8) + k \tag{5.2}$$

, where i is the *Bank Index*, j is the *Row Index*, and k is the *Byte Offset* in the given memory location.

The IO Interface of the ME_ACC is 16 bits wide. Hence, the Reference Frame Pixel Memory and the Current Frame Pixel Memory can be accessed only on the 16-bits or

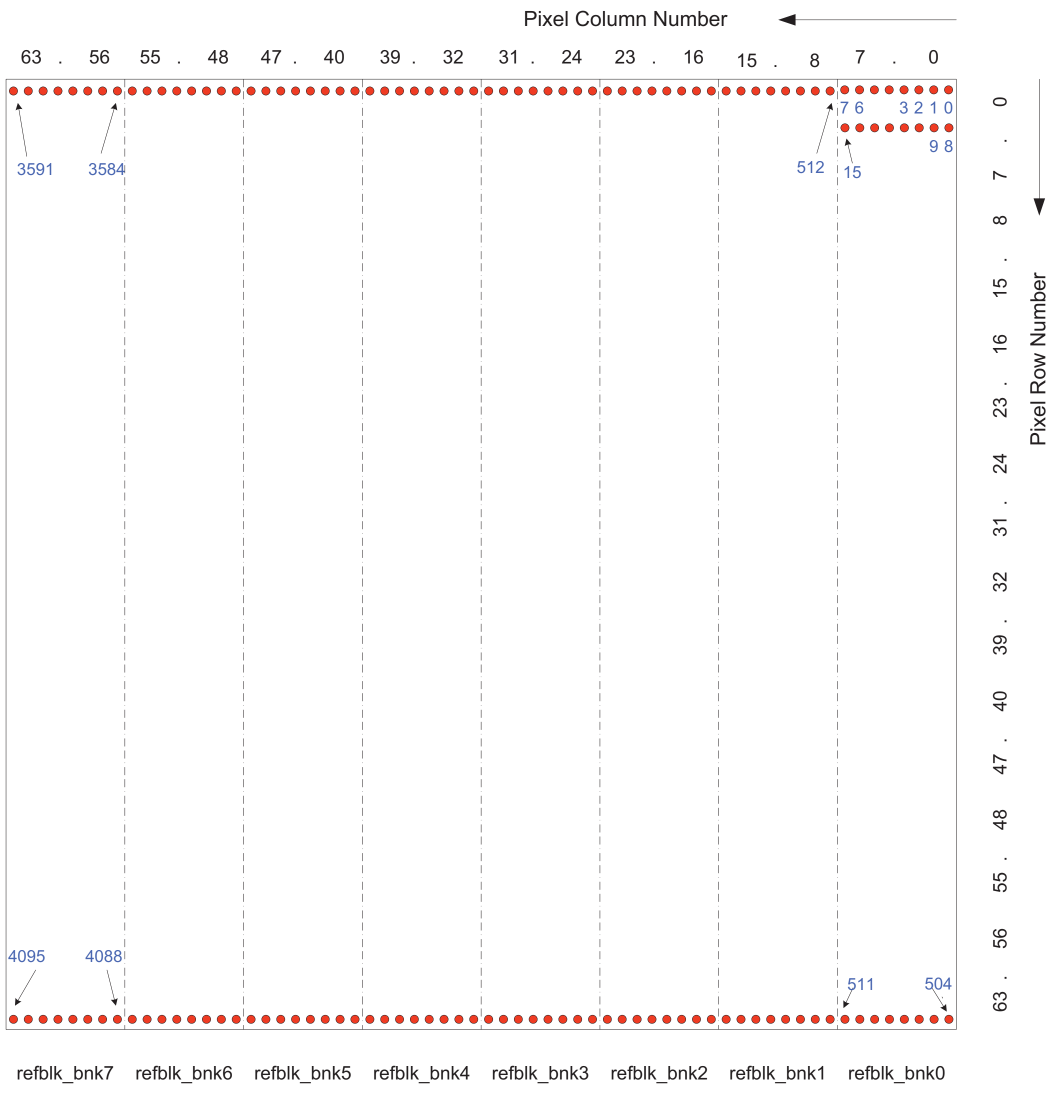


Figure 5.5: Reference frame pixel memory addressing

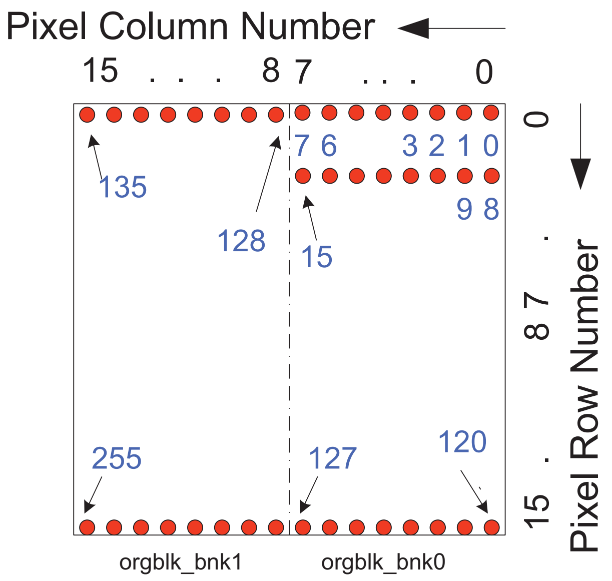


Figure 5.6: Current frame pixel memory addressing

2-bytes boundary.

Table 5.13: Row-wise memory address auto-increment of reference frame pixel memory

Address bits	15:12	11:9	8:3	2:0
	4'h9	*Bank Index*	*Row Index*	*Byte Offset*

The Reference Frame Pixel Memory and the Current Frame Pixel Memory can be accessed in either row-wise or column-wise orientation by auto-incrementing the address in row-wise or column-wise orientation, respectively. The Reference Frame Pixel Memory index is 12 bits wide. It comprises a 3-bits *Bank Index*, a 6-bits *Row Index*, and a 3-bits *Byte Offset*. Table 5.13 shows the row-wise memory addressing and Table 5.14 shows the column-wise memory addressing of the Reference Frame Pixel Memory. Index to the Current Frame Pixel Memory is 8 bits wide. It comprises a 1-bit *Bank Index*, a 4-bits *Row Index*, and a 3-bits *Byte Offset*. Table 5.15 shows the row-wise memory addressing and Table 5.16 shows

Table 5.14: Column-wise memory address auto-increment of reference frame pixel memory

Address bits	15:12	11:6	5:3	2:0
	4'h9	*Row Index*	*Bank Index*	*Byte Offset*

Table 5.15: Row-wise memory address auto-increment of current frame pixel memory

Address bits	15:8	7	6:3	2:0
	8'hA0	*Bank Index*	*Row Index*	*Byte Offset*

Table 5.16: Column-wise memory address auto-increment of current frame pixel memory

Address bits	15:8	7:4	3	2:0
	8'hA0	*Row Index*	*Bank Index*	*Byte Offset*

the column-wise memory addressing of the Current Frame Pixel Memory.

The only limitation on the memory access is that, the number of memory locations accessed by every memory write transaction needs to be a multiple of 8 bytes, aligned on 8-bytes boundary. This limitation comes from the fact that, the memory banks used are 64 bits wide and memory addressing logic is kept simple in order to achieve higher throughput under normal operating conditions. The memory access limitation however, should not affect the performance of the MV search as the "cache line size" of the Host CPU's cache memory is generally a multiple of 8 bytes.

Following paragraphs describe the mapping of frame pixels in the Reference Frame Pixel Memory and the Current Frame Pixel Memory.

Figure 5.7 shows a Current Frame with a Current Micro-block of size 16x16 pixels. In this example, the pixel coordinates of the top-left corner of the Current Micro-block are (48, 32). Figure 5.8 shows a reference frame with a search window of range *Srch_Rng_X_Min* to *Srch_Rng_X_Max* and *Srch_Rng_Y_Min* to $Srch\_Rng\_Y\_Max$, where

- *Srch_Rng_X_Min* is the minimum value of the search range along X axis,
- *Srch_Rng_X_Max* is the maximum value of the search range along X axis,
- *Srch_Rng_Y_Min* is the minimum value of the search range along Y axis, and
- *Srch_Rng_Y_Max* is the maximum value of the search range along Y axis.

In this example *Srch_Rng_X_Min* = *Srch_Rng_Y_Min* = -16 pixels, and *Srch_Rng_X_Max* = *Srch_Rng_Y_Max* = $+15$ pixels. Figure 5.8 also depicts the micro-block position corresponding to the Current Micro-block shown in Figure 5.7. The search

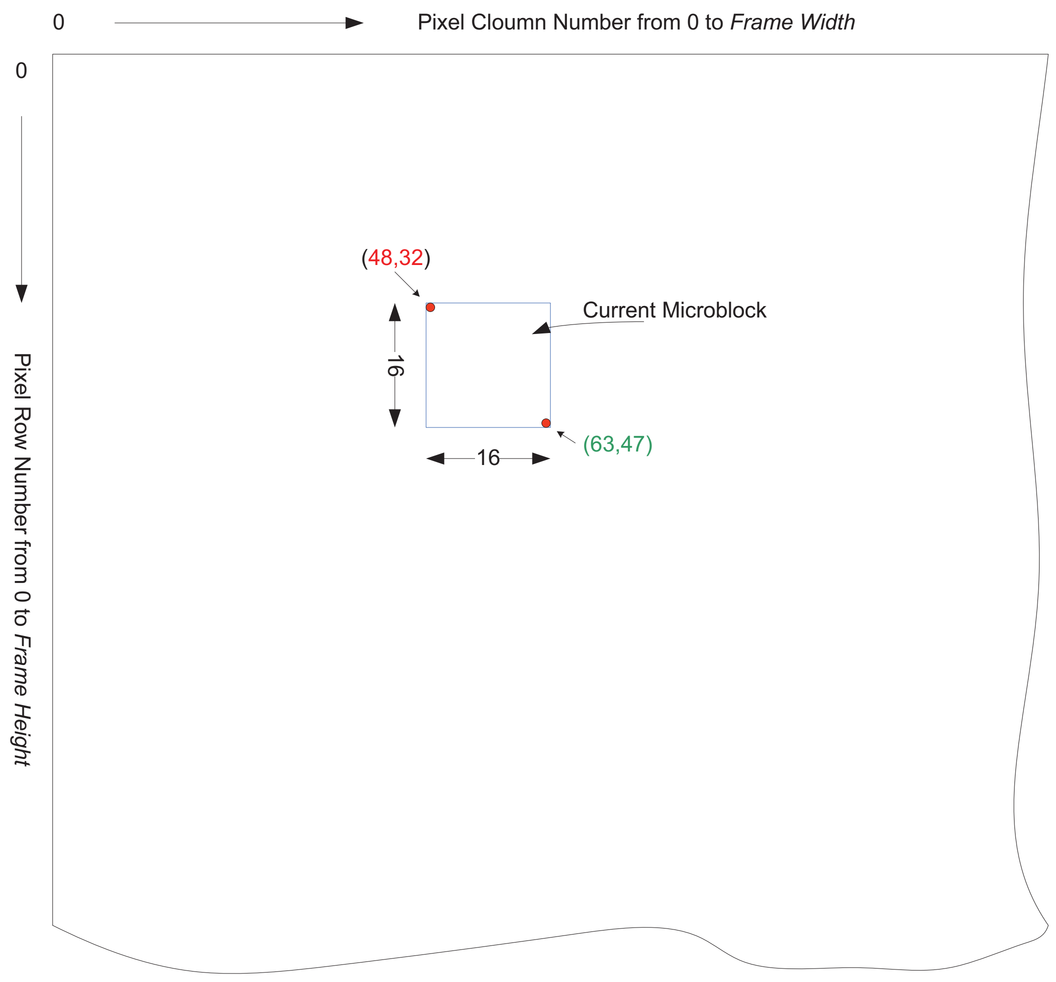

0
Pixel Cloumn Number from 0 to *Frame Width*
0
Pixel Row Number from 0 to *Frame Height*
(48,32)
Current Microblock
16
16
(63,47)

Figure 5.7: Current frame with current micro-block

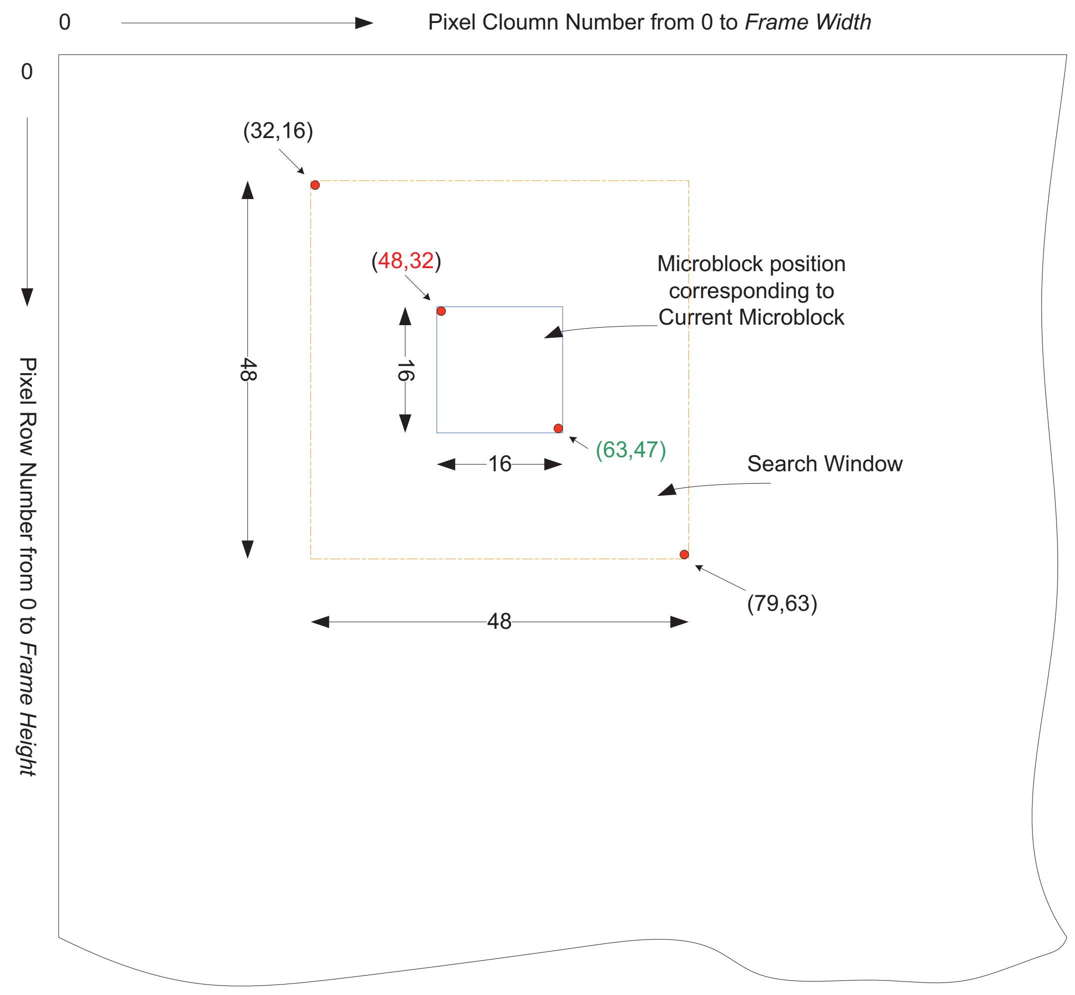

0
Pixel Cloumn Number from 0 to *Frame Width*
0
Pixel Row Number from 0 to *Frame Height*
(32,16)
48
(48,32)
16
Microblock position corresponding to Current Microblock
16
(63,47)
Search Window
(79,63)
48

Figure 5.8: Reference frame with search window

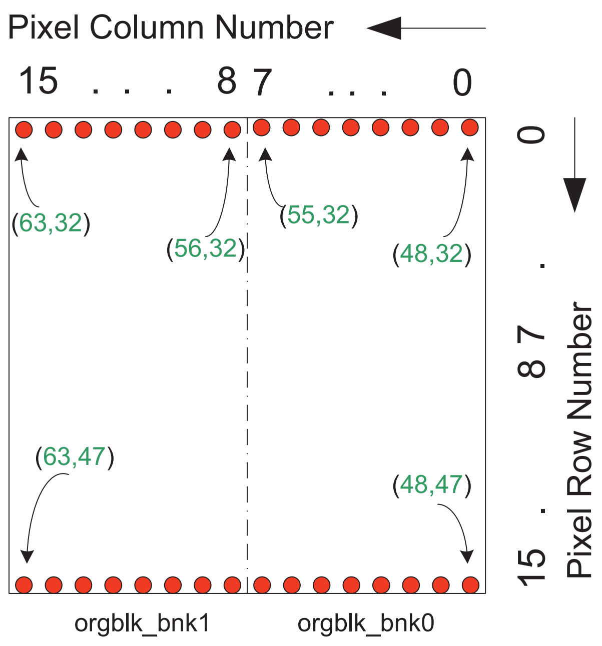


Figure 5.9: Mapping of current micro-block in the current frame pixel memory

window size ($Search\_Window\_Width \times Search\_Window\_Height$) is given by Equations 5.3 and 5.4.

$$\begin{aligned} Search\_Window\_Width = {} & Srch\_Rng\_X\_Max - Srch\_Rng\_X\_Min \\ & + BLK\_SZ\_X + 1 \end{aligned} \tag{5.3}$$

, where $Search\_Window\_Width$ is the search window width in terms of number of pixels along the X axis.

$$\begin{aligned} Search\_Window\_Height = {} & Srch\_Rng\_Y\_Max - Srch\_Rng\_Y\_Min \\ & + BLK\_SZ\_Y + 1 \end{aligned} \tag{5.4}$$

, where $Search\_Window\_Height$ is the search window height in terms of number of pixels along the Y axis.

Hence, in this example, the search window size is (48×48) pixels.

Figure 5.9 shows mapping of the Current Micro-block from the Figure 5.7 in the Current Frame Pixel Memory. Figure 5.10 shows mapping of the search window from the Figure 5.8 in the Reference Frame Pixel Memory. Unlike a sequential memory, the Current Frame Pixel Memory and the Reference Frame Pixel Memory hold pixel data in

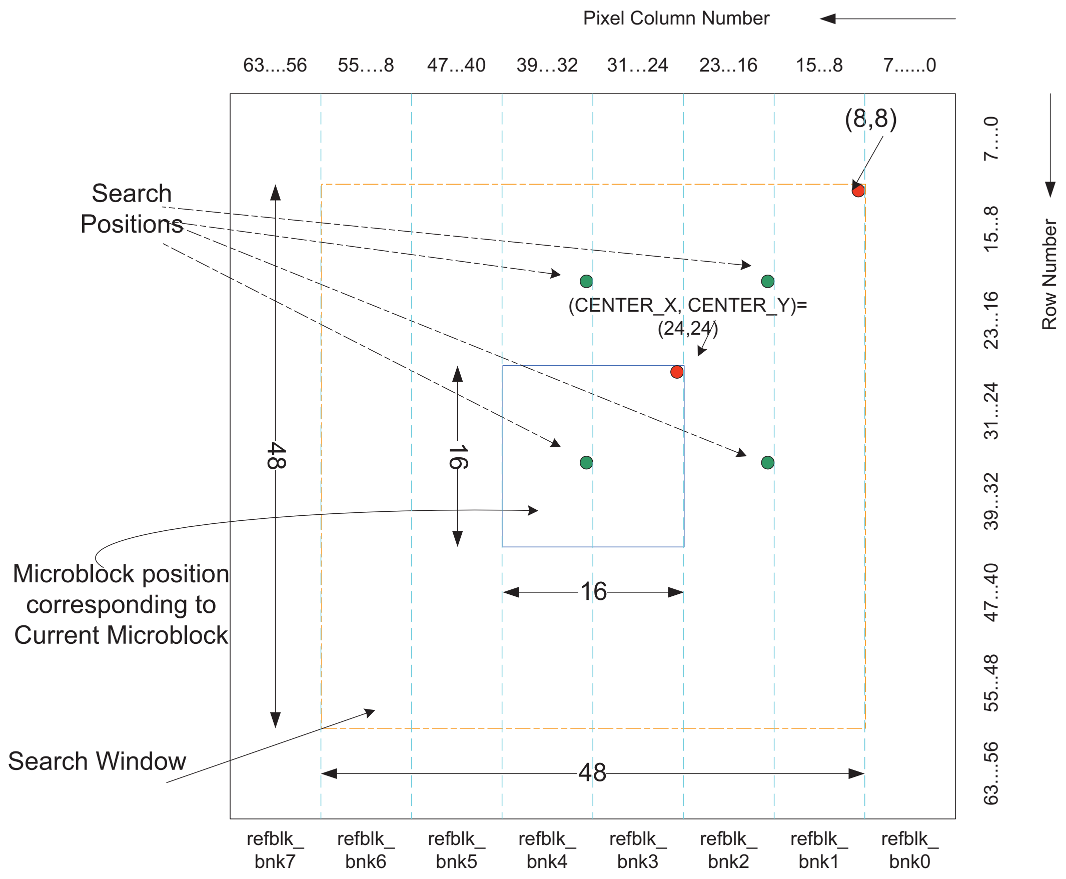


Figure 5.10: Mapping of search window pixels in the reference frame pixel memory

two-dimensional arrays with pixel rows and columns corresponding to those in a picture frame. Typically, pixel columns in a picture frame are numbered from left to right in the increasing order. By convention, memory bytes are ordered from right to left in the increasing order (i.e. least significant byte (LSB) at right and most significant byte (MSB) at left). Hence, the pixel data in the Reference Frame Pixel Memory and the Current Frame Pixel Memory appear like mirror images of pixel data in the corresponding picture frames along the Y axis.

The Current Micro-block top-left pixel at (48, 32) in the Figure 5.7 corresponds to the pixel at (0, 0) in the Current Frame Pixel Memory and the bottom-right corner pixel at (63, 47) in the Figure 5.7 corresponds to the pixel at (15, 15) in the Current Frame Pixel Memory, as shown in the Figure 5.9.

A given search window of size ($Search\_Window\_Width \times Search\_Window\_Height$) pixels can be loaded anywhere in the Reference Frame Pixel Memory in a contiguous memory block of the same size as that of the search window. In this example, the search window of size (48×48) pixels, shown in the Figure 5.8, is loaded at pixel rows 8 through 55 and pixel columns 8 through 55 in the Reference Frame Pixel Memory. Thus, the search window top-left pixel at (32, 16) in the Figure 5.8 corresponds to pixel at (8, 8) in the Reference Frame Pixel Memory, and the search window bottom-right pixel at (79, 63) in the Figure 5.8 corresponds to pixel at (55, 55) in the Reference Frame Pixel Memory as shown in the Figure 5.10.

The register *CENTER_XY* gives the top-left corner coordinates of the micro-block position corresponding to the Current Micro-block in the Reference Frame Pixel Memory. The register fields *CENTER_X* and *CENTER_Y* are given by Equation 5.5 and Equation 5.6 respectively.

$$CENTER\_X = Srch\_Win\_Top\_Left\_Pix\_Col - Srch\_Rng\_X\_Min \tag{5.5}$$

, where *Srch_Win_Top_Left_Pix_Col* is the column index of the pixel at the top-left corner of the search window in the Reference Frame Pixel Memory.

$$CENTER\_Y = Srch\_Win\_Top\_Left\_Pix\_Row - Srch\_Rng\_Y\_Min \tag{5.6}$$

, where *Srch_Win_Top_Left_Pix_row* is the row index of the pixel at the top-left corner of the search window in the Reference Frame Pixel Memory.

Hence, in this example, pixel coordinates of the top-left corner of the micro-block position corresponding to the Current Micro-block are (24, 24).

The search position coordinates, specified by a pair of registers *SRCH_PTRN_n_X[i]* and *SRCH_PTRN_n_Y[i]*, are relative to the pixel coordinates given by *predicted_center_x* and *predicted_center_y* respectively. The Figure 5.10 shows four search positions: (-8, -8), (-8, 8), (8, 8), and (8, -8). The coordinates of the pixel at a search position, (*SRCH_PTRN_n_X[i]*, *SRCH_PTRN_n_Y[i]*), in the Reference Frame Pixel Memory are given by Equation 5.7 and Equation 5.8.

$$\begin{aligned} &\mathit{predicted\_center\_x} = \mathit{CENTER\_X} + \mathit{MVPRED\_X} \\ &\mathit{Ref\_Mem\_Col} = \mathit{SRCH\_PTRN\_n\_X}[i] + \mathit{predicted\_center\_x} \end{aligned} \tag{5.7}$$

, where *Ref_Mem_Col* is the column index of the pixel at the search position (*SRCH_PTRN_n_X[i]*, *SRCH_PTRN_n_Y[i]*) in the Reference Frame Pixel Memory

$$\begin{aligned} &\mathit{predicted\_center\_y} = \mathit{CENTER\_Y} + \mathit{MVPRED\_Y} \\ &\mathit{Ref\_Mem\_Row} = \mathit{SRCH\_PTRN\_n\_Y}[i] + \mathit{predicted\_center\_y} \end{aligned} \tag{5.8}$$

, where *Ref_Mem_Row* is the row index of the pixel at the search position (*SRCH_PTRN_n_X[i]*, *SRCH_PTRN_n_Y[i]*) in the Reference Frame Pixel Memory

In the above example $\mathit{MVPRED\_X} = \mathit{MVPRED\_Y} = 0$, and hence the Reference Frame Pixel Memory coordinates at search position (-8, 8) are (16, 32).

Figure 5.11 shows the micro-block in the Reference Frame Pixel Memory at the search position (-8, 8), Figure 5.12 shows the micro-block in the Reference Frame Pixel Memory at the search position (8, 8), Figure 5.13 shows the micro-block in the Reference Frame Pixel Memory at the search position (8, -8), and Figure 5.14 shows the micro-block in the Reference Frame Pixel Memory at the search position (-8, -8).

The multi-bank organization of the memory offers two main advantages. The SAD Computation Unit can access 16 pixels in a row with only one memory access to two banks in parallel. In a normal sequential 64 bits wide memory, pixels are stored sequentially, one double-word after another. In such an arrangement, minimum two and up to three memory

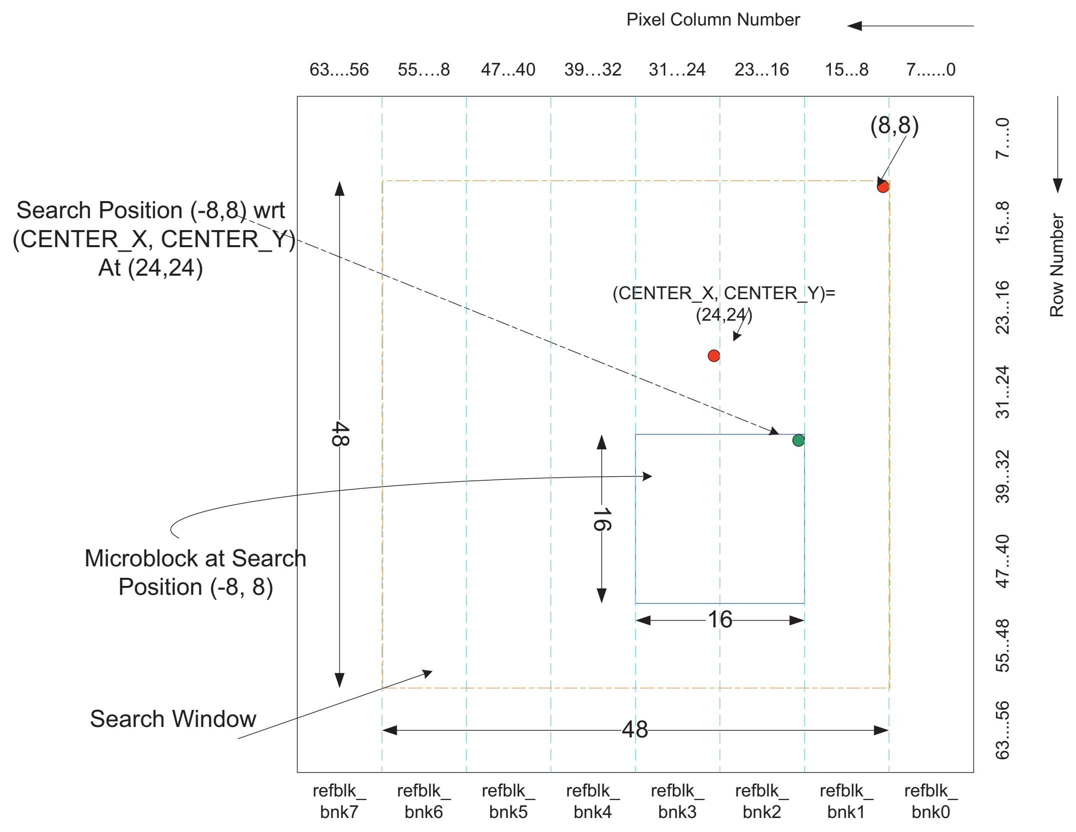


Figure 5.11: Micro-block at search position (-8, 8)

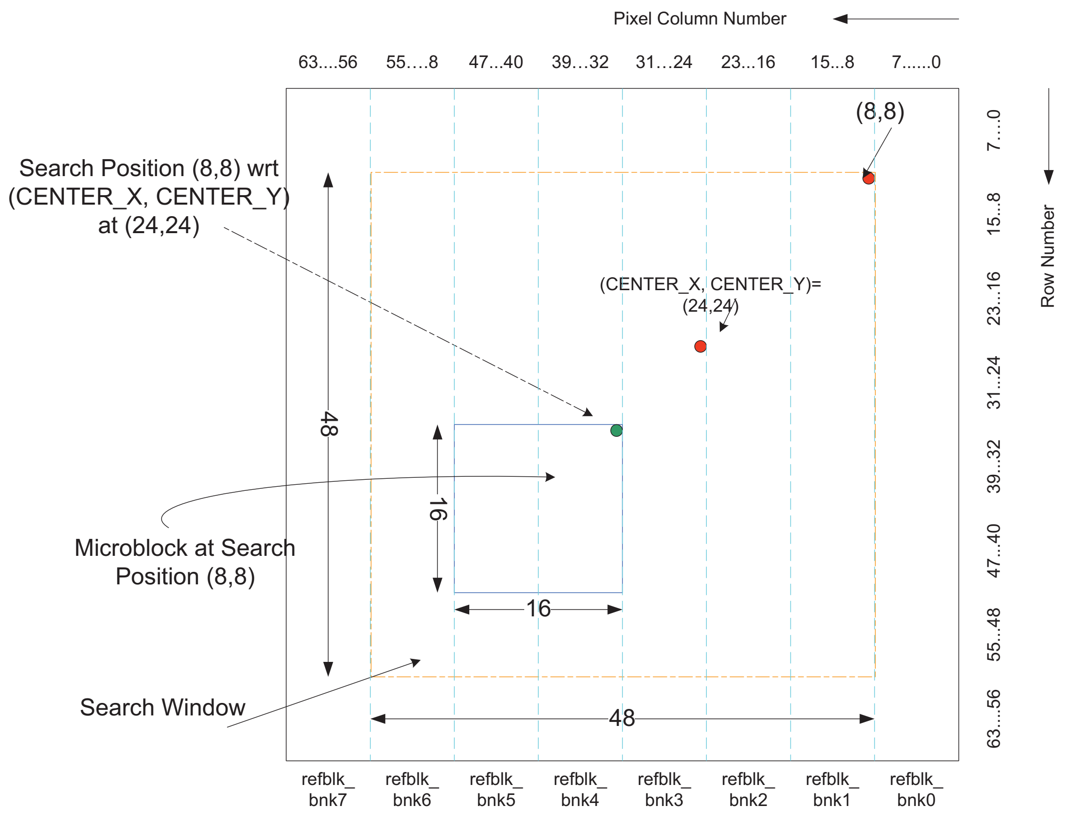


Figure 5.12: Micro-block at search position (8, 8)

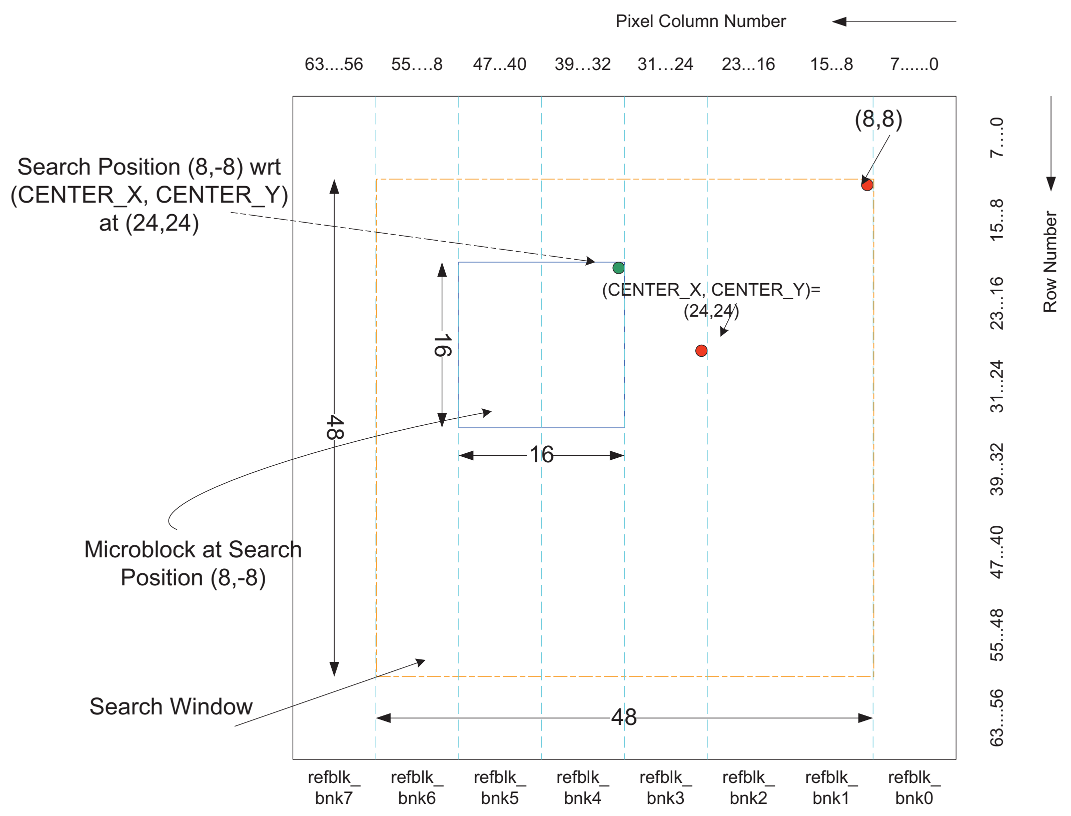


Figure 5.13: Micro-block at search position (8, -8)

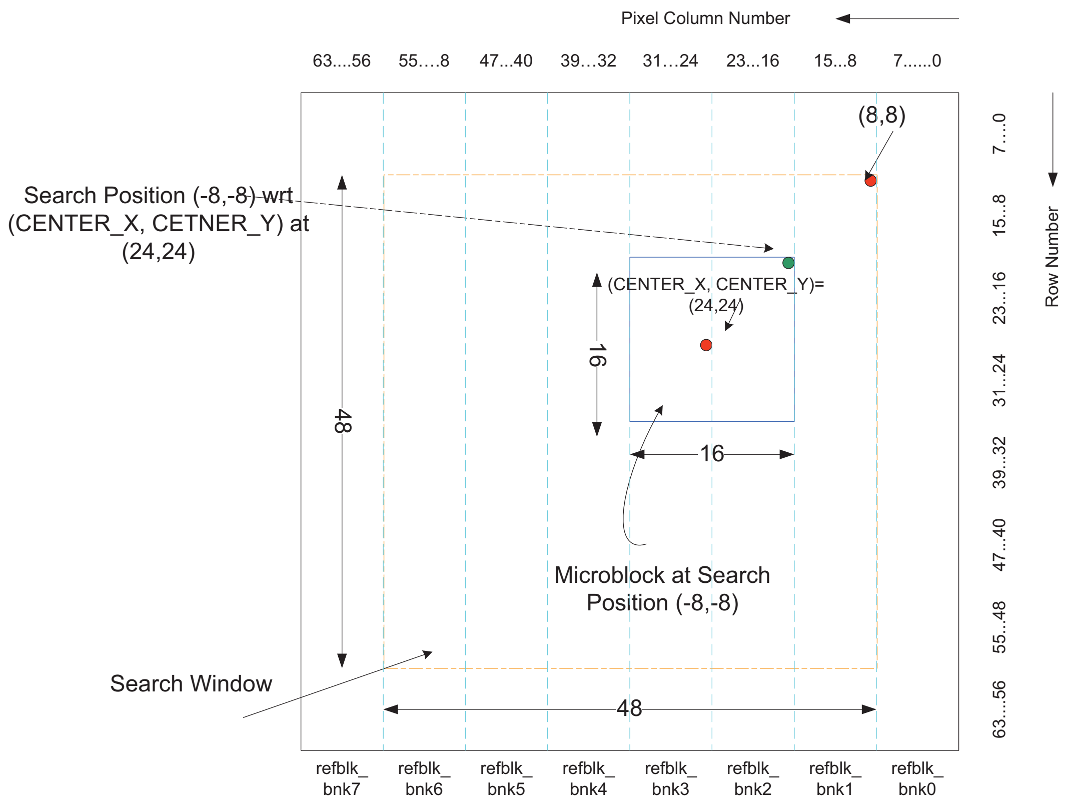


Figure 5.14: Micro-block at search position (-8, -8)

accesses are required to access 16 consecutive pixels from a row. Another advantage is that, the multi-bank organization of dual ported SRAM allows concurrent access to the SRAM by both the neighboring AsAP and the SAD Computation Unit. Thus, it enables us to exploit the task parallelism capability provided by the AsAP to minimize the memory access time. The IO Interface Unit supports memory access while ME_ACC is performing the SAD computation.

5.3 IO Interface Unit

The IO interface unit communicates with the neighboring AsAPs using 16 bits wide DATA_IN and DATA_OUT buses.

This module is implemented in the Verilog file me_regmem_config.v

Table 5.17: ME_ACC signal interface

Signal name	I/O	Description
DATA_IN [15:0]	Input	Data input from AsAP.
DATA_IN_VLD	Input	DATA_IN valid signal from AsAP. This is an active high signal.
DATA_IN_REQ	Output	DATA_IN request to AsAP. This is an active high signal. AsAP drives DATA_IN_VLD "1" only when DATA_IN_REQ is "1".
DATA_OUT [15:0]	Output	Data output to AsAP.
DATA_OUT_REQ	Input	Data output request from AsAP. This is an active high signal. ME_ACC drives DATA_OUT_VLD "1" only when DATA_OUT_REQ is "1".
DATA_OUT_VLD	Output	Data output valid signal to AsAP. This is an active high signal.
CLK	Input	Clock Input.
RST	Input	Active low asynchronous reset input.

Table 5.17 summarizes the IO Interface signals. The neighboring AsAP drives IO interface to access various registers in the Configuration Registers Unit and to access the Reference Frame Pixel Memory and the Current Frame Pixel Memory in the Memory Unit.

Input Bus Protocol consists of two control words followed by address and data words. Table 5.18 describes the fields of the Input Bus Protocol. Figure 5.15 illustrates the write transaction to multiple consecutive memory locations in the Memory Unit. Table 5.19 gives an example of a sequence of data words to be driven on DATA_IN bus in order to write to 4 consecutive memory locations. Figure 5.16 illustrates the write transaction to multiple registers with non-consecutive addresses in the Configuration Registers Unit. Table 5.20 gives an example of a sequence of data words to be driven on DATA_IN bus in order to

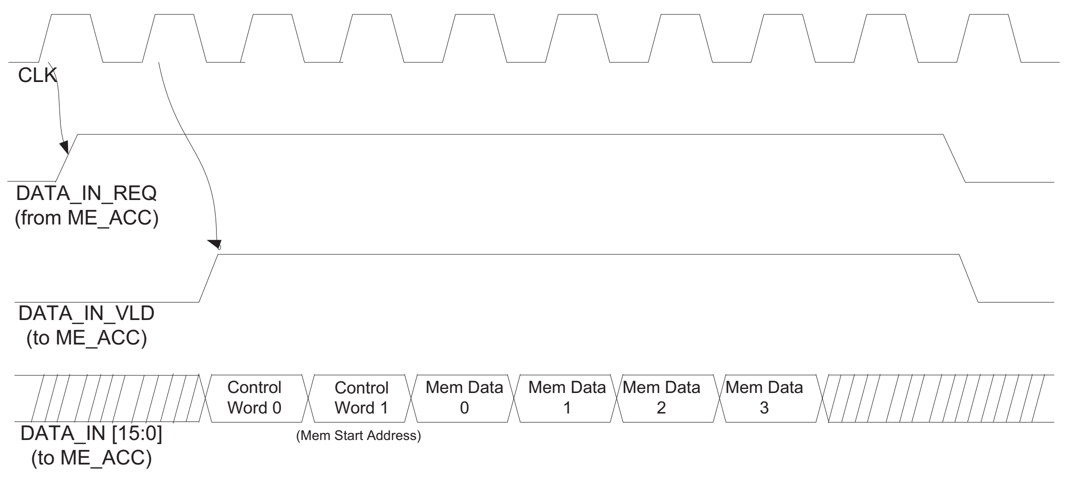


Figure 5.15: Multiple memory locations write transaction

write to 4 configuration registers.

The Input Bus Protocol is devised to minimize the number of clock cycles required for programming the registers and loading the frame pixel memories. It allows the user to program multiple discrete registers in a single transaction, thus minimizing the overhead of transferring control words. For memory programming, only the address of the start location and the number of locations need to be specified. This reduces the overhead of specifying address for every location. The Input Bus Protocol also lets the user load the Reference Frame Pixel Memory and the Current Frame Pixel Memory in either row-wise or column-wise mode. This feature further reduces the overhead of specifying the address, when only few columns in all the rows or few rows in all the columns need to be reloaded to take advantage of memory-reuse.

The IO Interface unit returns two types of Output Responses on the Output Bus to the neighboring AsAP using the Output Bus Protocol:

1. Response to SAD Compute Request. The SAD Compute Request is initiated by
   - Writing "1" to *START_ME* Register or
   - Writing "1" to *CONT_ME* Register
2. Response to Read Request

Table 5.18: Input bus protocol bit field description

Word	Bit field	Description

Control Word 0	15	if *Control Word 0*[15] = "1"
		Start a transaction.
		if *Control Word 0*[15] = "0"
		Ignore the data.

	14	if *Control Word 0*[14] = "1"
		This is a read transaction.
		if *Control Word 0*[14] = "0"
		This is a write transaction.

	13:12	if *Control Word 0*[13:12] = "01"
		This is a register access transaction.
		if *Control Word 0*[13:12] = "10"
		This is a row-wise memory access transaction, where address auto-increment is done in row-wise orientation.
		if *Control Word 0*[13:12] = "11"
		This is a column-wise memory access transaction, where address auto-increment is done in column-wise orientation.
		if *Control Word 0*[13:12] = "00"
		Unused

	11:0	Number of registers or memory locations to be accessed.

Control Word 1	15:0	if *Control Word 0*[13] = "1"
		Control Word 1 specifies memory access start address.
		if *Control Word 0*[13] = "0"
		Control Word 1 = 16'h0000.

Word 0	15:0	if *Control Word 0*[13] = "1"
		Word 0 specifies memory write data.
		if *Control Word 0*[13] = "0"
		Word 0 specifies register address.

Word 1	15:0	if *Control Word 0*[13] = "1"
		Word 1 specifies memory write data for next location.
		if *Control Word 0*[13] = "0"
		Word 1 specifies register write data.

Table 5.19: Example of write to four consecutive memory locations

Word index	Value	Description
i	0xA004	*Control Word 0* bit[15] = 1: Start a transaction bit[14] = 0: Write transaction bit[13:12] = 2'b10: Row-wise memory access bit[11:0] = 12'h004: Number of memory locations accessed is 4
$i+1$	0x9100	*Control Word 1* Memory access start address in the reference frame pixel memory
$i+2$	0x5577	2 bytes data to be written at word address 0x9100
$i+3$	0xAABB	2 bytes data to be written at word address 0x9102
$i+4$	0xCCDD	2 bytes data to be written at word address 0x9104
$i+5$	0xEEFF	2 bytes data to be written at word address 0x9106

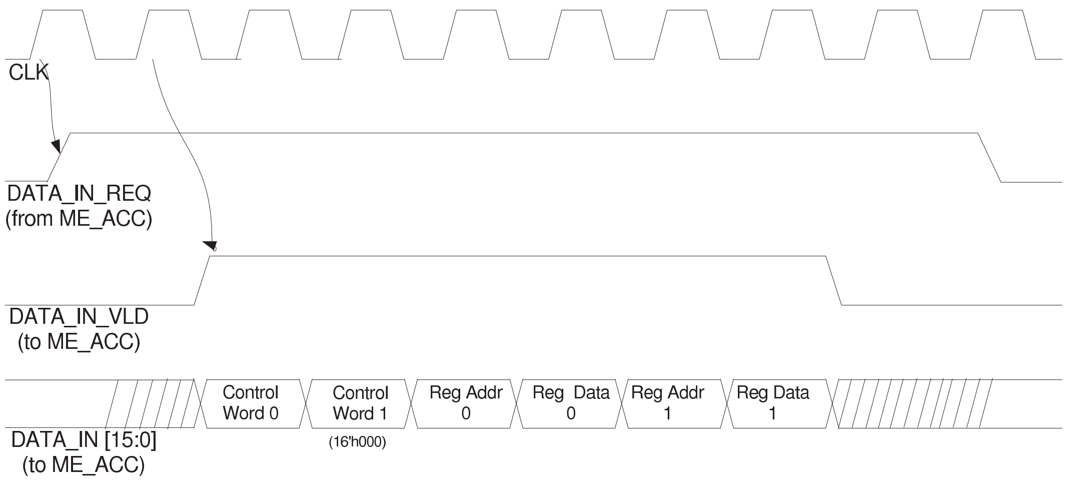


Figure 5.16: Multiple registers write transaction

Table 5.20: Example of write to four registers

Word index	Value	Description
i	0x9004	*Control Word 0* bit[15] = 1: Start a transaction bit[14] = 0: Write transaction bit[13:12] = 2'b01: Register access bit[11:0] = 12'h004: Number of memory locations accessed is 4
$i+1$	0x0000	*Control Word 1*
$i+2$	0x8002	Address of register *BLK_SZ_X*
$i+3$	0x0010	Data to be written at register *BLK_SZ_X*
$i+4$	0x8004	Address of register *BLK_SZ_Y*
$i+5$	0x0010	Data to be written at register *BLK_SZ_X*
$i+6$	0x8100	Address of register *SRCH_PTRN_0_X[0]*
$i+7$	0x0002	Data to be written at register *SRCH_PTRN_0_X[0]*
$i+8$	0x8300	Address of register *SRCH_PTRN_0_Y[0]*
$i+9$	0x0001	Data to be written at register *SRCH_PTRN_0_Y[0]*

The Output Bus Protocol consists of one control word followed by one or more data words. The bit-field description for the output response to a SAD Compute Request is given in Table 5.21. Figure 5.17 illustrates the output response to a SAD Compute Request. Table 5.22 describes bit-fields for the output response to Read Request. Figure 5.18 shows the output response to a Read Request.

The ME_ACC does not lock the IO Interface while performing SAD computation. Hence, the neighboring AsAPs can perform the Memory/Register Write/Read accesses safely, while ME_ACC is performing SAD computation, as long as the accesses do not conflict with the register/memory settings of the on-going best MV search. This feature enables the neighboring AsAPs to achieve task parallelism by programming the ME_ACC for $k+1^{\text{st}}$ current micro-block while the ME_ACC is performing MV search for k^{th} current micro-block.

Table 5.21: Bit field description for response to SAD compute request

Word name	Bit field	Value/ Description
Control Word	15:0	0x4000
Data Word 1	15:8 7:0	Search pattern index used for SAD compute request Search position index within search pattern used for SAD compute request
Data Word 2	15:0	SAD value

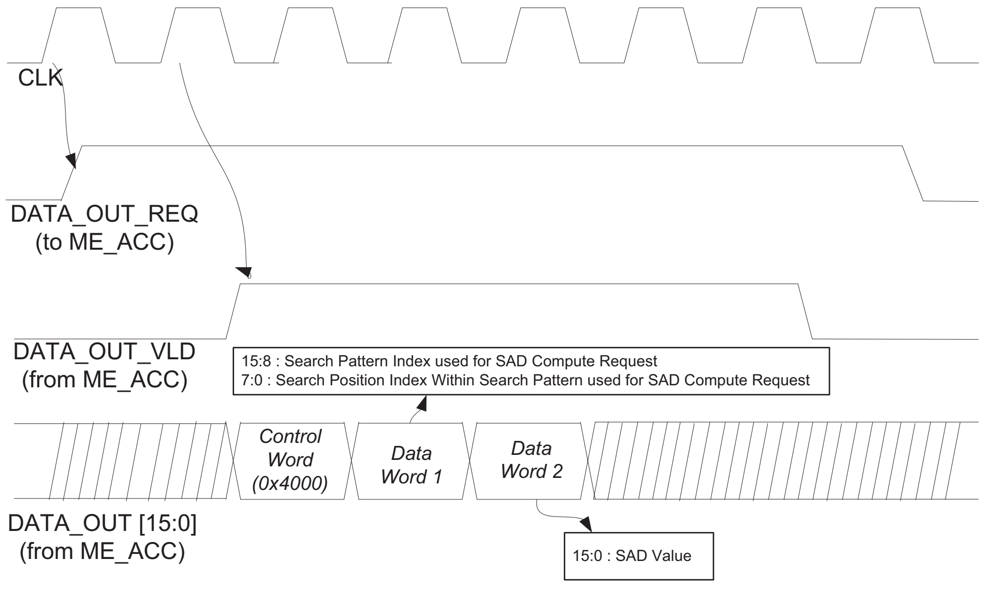

CLK
DATA_OUT_REQ (to ME_ACC)
DATA_OUT_VLD (from ME_ACC)
15:8 : Search Pattern Index used for SAD Compute Request
7:0 : Search Position Index Within Search Pattern used for SAD Compute Request
Control Word (0x4000)
Data Word 1
Data Word 2
DATA_OUT [15:0] (from ME_ACC)
15:0 : SAD Value

Figure 5.17: Response to SAD compute request

Table 5.22: Bit field description for response to read request

Word name	Bit field	Value/ Description
Control Word	15:0	0x2000
Data Word 1	15:0	Memory or register read data
Data Word 2	15:0	Memory read data in case of memory read request for more than one memory location
Data Word 3 through N	15:0	Memory read data in case of memory read request for N memory locations

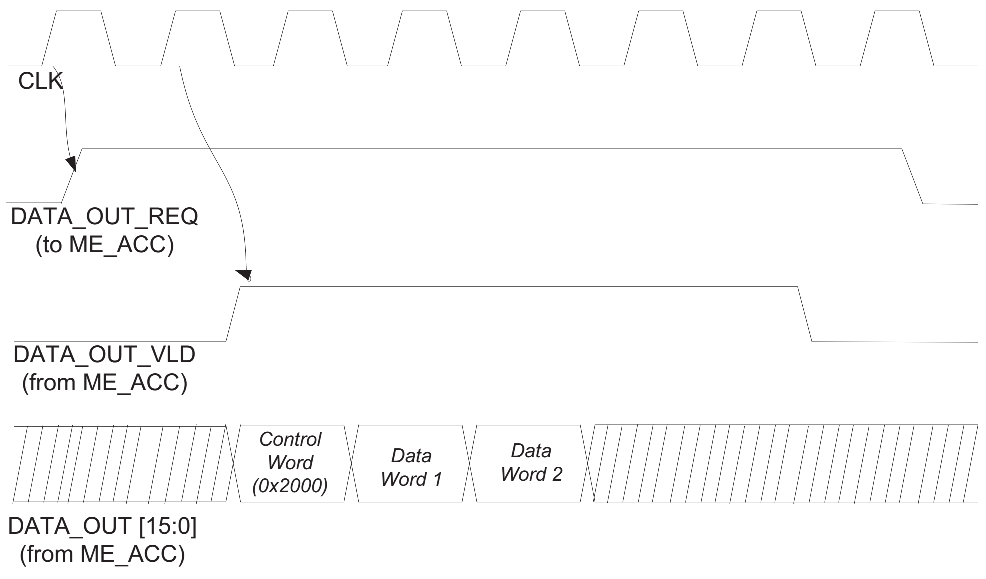


Figure 5.18: Response to read request

5.4 Address Generation Unit

The Address Generation Unit generates address to the Reference Frame Pixel Memory for the SAD Computation Unit. This module is implemented in the Verilog file me_refblk_adgen.v

The Address Generation Unit uses a set of configuration registers, *CENTER_Y*, *MVPRED_Y*, *BLK_SZ_Y*, and *SRCH_PTRN_n_Y[i]*, to generate a row index to the Reference Frame Pixel Memory. The row index selects a pixel row from the available 64 pixel rows in the 8 memory banks. Another set of configuration registers, *CENTER_X*, *MVPRED_X*, *BLK_SZ_X*, and *SRCH_PTRN_n_X[i]*, is used to generate column indices to select 4, 8, or 16 pixels of data from the selected row.

The Pixel Multiplexer multiplexes the pixel data using the column indices generated by Address Generation Unit and passes it on to the SAD Computation Unit.

The Reference Frame Pixel Memory address decoding for IO interface is handled in the Memory Unit.

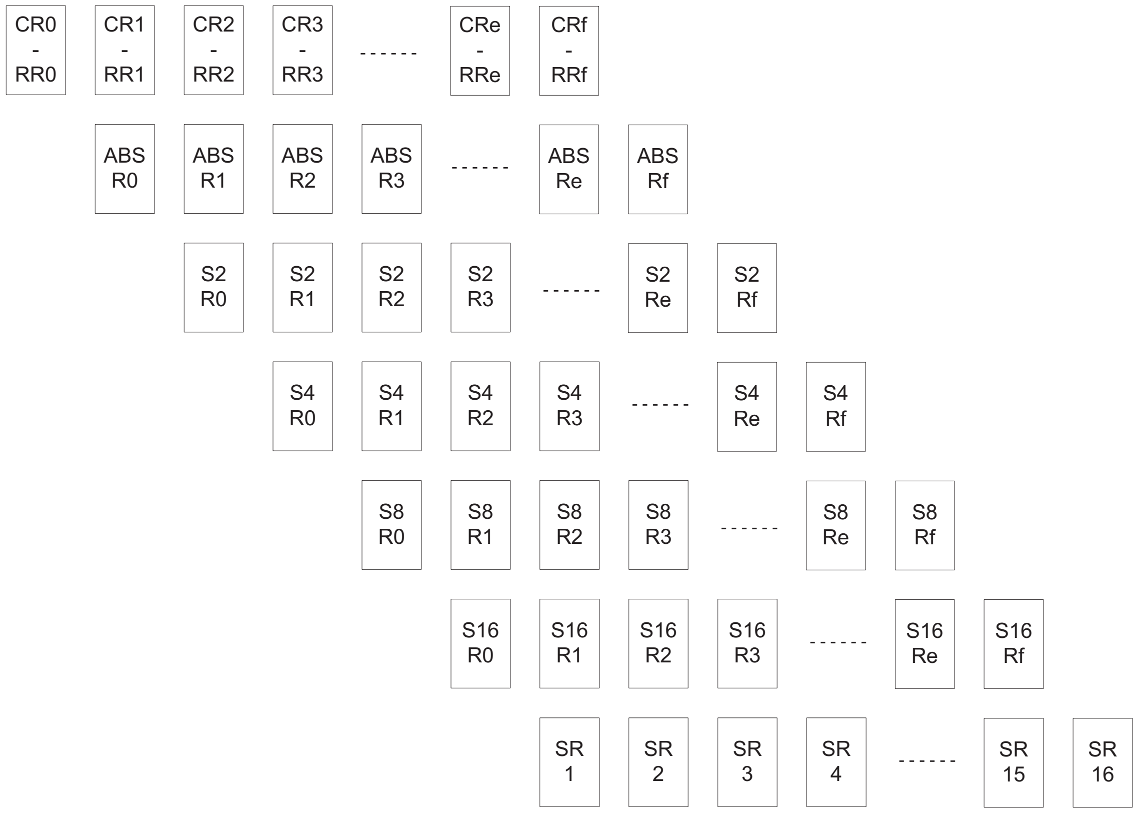


Figure 5.19: SAD computation pipe line

CRn: Current micro-block row n, RRn: Reference micro-block row n.

5.5 SAD Computation Unit

The SAD Computation Unit computes the sum of absolute differences between the pixels of the Reference Micro-block and the pixels of the Current Micro-block. It is implemented in the Verilog file me_sad_pipe.v.

The SAD Computation Unit receives up to sixteen pixels in a row from the Reference Frame Pixel Memory and an equal number of pixels in a row from the Current Frame Pixel Memory every clock cycle, until all the rows in the Current Micro-block are traversed. It uses seven stage pipeline as shown in Figure 5.19 to compute SAD between the pixels of the Reference Micro-block and the pixels of the Current Micro-block. The SAD value is returned after ($Initial\ Latency + BLK\_SZ\_Y$) number of clock cycles, where $Initial\ Latency$ is the pipeline depth, 7. Tasks performed in each of the seven pipeline stages are described below.

1. Stage "$CRn - RRn$": Up to 16 pixels of a given row (row index n) of the Reference Micro-block are subtracted from the corresponding row pixels of the Current Micro-block to generate up to 16 pixel differences, $pix\_diff[i]$, where $i = 0$ to 15.

2. Stage "$ABSRn$": Absolute pixel differences, $abs\_diff[i]$, are generated from $pix\_diff[i]$, where $i = 0$ to 15 for a given row index n.

3. Stage "$S2Rn$": Stages "$S2Rn$", "$S4Rn$", "$S8Rn$", and "$S16Rn$" form the 4 stages of the adder tree that combines up to 16 $abs\_diff[i]$ values to generate "sum of absolute differences" terms for a given row. In Stage $S2Rn$, pairs of $abs\_diff[i]$ are combined to generate up to 8 "sum of absolute differences" terms.

4. Stage "$S4Rn$": Up to 8 "sum of absolute differences" terms, generated in the Stage "$S2Rn$", are combined to generate up to 4 "sum of absolute differences" terms.

5. Stage "$S8Rn$": Up to 4 "sum of absolute differences" terms, generated in the Stage "$S4Rn$", are combined to generate up to 2 "sum of absolute differences" terms.

6. Stage "$S16Rn$": Up to 2 "sum of absolute differences" terms, generated in the Stage "$S8Rn$", are combined to generate "sum of absolute differences" for the given row n.

7. Stage "*SRn+1*": The "sum of absolute differences" term, generated in the Stage "$S16Rn$", for the row $n + 1$ is added to the accumulator, where the accumulator is initialized with "sum of absolute differences" for row 0. The "sum of absolute differences" terms for all the rows in the Current Micro-block are added together to generate the SAD value at a given search position. The number of rows in the current micro-block is given by the programmed value of the register *BLK_SZ_Y*.

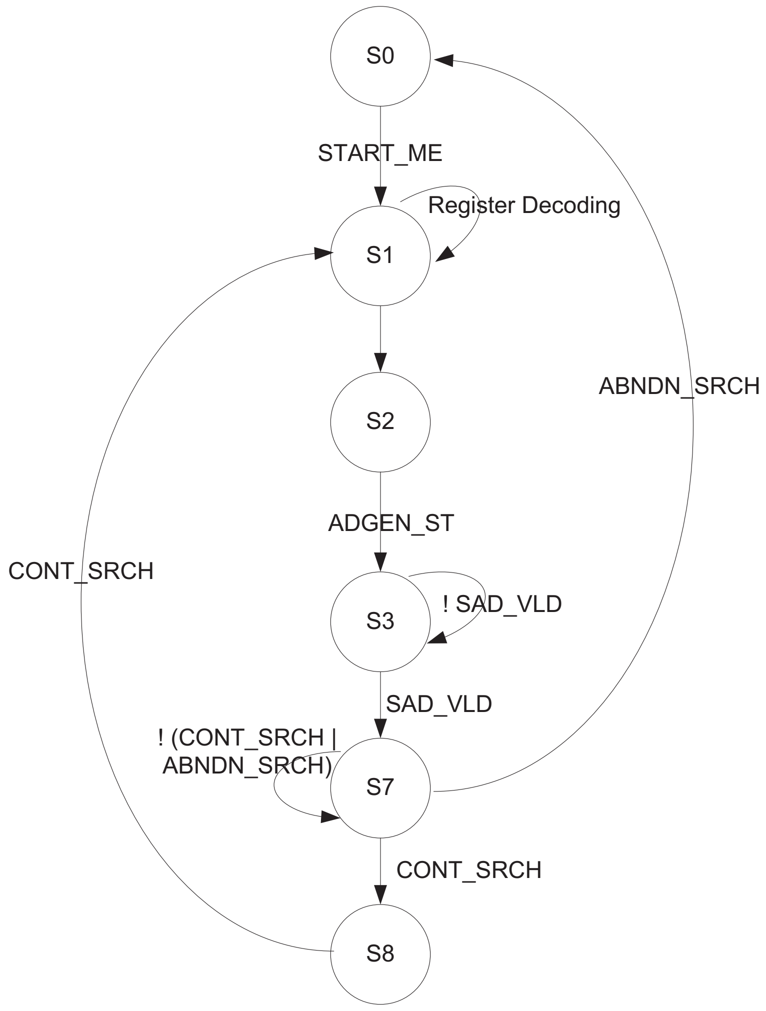


Figure 5.20: Motion estimation accelerator state machine

5.6 Control Unit

The Control Unit is implemented in Verilog file me_fsm.v

The Control Unit of the ME_ACC is a small state machine controlled by few register inputs, such as *START_ME*, *CONT_ME*, and *ABNDN_ME*, and a valid signal, *SAD_VLD*, from the SAD Computation Unit. When the register *START_ME* is set to 1, the ME_ACC starts the best MV search by computing SAD at the search position given by the registers $SRCH\_PTRN\_n\_X[0]$ and $SRCH\_PTRN\_n\_Y[0]$, where n is given by the register *SRCH_PTRN_CNT*. When the SAD Computation Unit completes the SAD computation at a given search position, it sets the *SAD_VLD* signal to "1". After receiving $SAD\_VLD = 1$, the Control Unit waits in State “S7”. If the Control Unit receives the register *CONT_SRCH* = “1”, the ME_ACC continues with computing SAD after incrementing the index i to the registers *SRCH_PTRN_n_X*[i] and *SRCH_PTRN_n_Y*[i]. If the Control Unit receives the register *ABNDN_ME* = “1”, the best MV search is terminated and the state machine returns to the “S0” state. The state machine is depicted in Figure 5.20.

Tasks performed in each state and the state transitions are described below.

- State “S0” : This is the idle state. The state machine is initialized to state “S0” out of reset. The state machine remains in “S0” until the register *START_ME* is set to “1”. When the register *START_ME* is set to “1”, the state machine transitions to state “S1”.

- State “S1” : The state machine remains in this state for two clock cycles. Register decoding is performed and various parameters are set for SAD computation during this state. After two clock cycles, the state machine transitions to state “S2”.

- State “S2” : Address generation is started for the Reference Frame Pixel Memory. The state machine transitions to state “S3”.

- State “S3” : The state machine remains in this state until the SAD computation at a given search position is done for the Current Micro-block. The number of clock cycles spent in this state is equal to the register *BLK_SZ_Y*. When the *SAD_VLD* signal from SAD Computation Unit is set to “1”, indicating that SAD computation is done, the state machine transitions to state “S7”.

- state “S7” : The state machine remains in this state until either the register *CONT_ME* or the register *ABNDN_ME* is set to “1” by the neighboring AsAP. If the register *CONT_ME* is set to “1”, the state machine transitions to state “S8”. If the register *ABNDN_ME* is set to “1”, the state machine transitions to state “S0”.

- State “S8” : The Search Position Index i to the selected search pattern registers, *SRCH_PTRN_n_X*[i] and *SRCH_PTRN_n_Y*[i], is incremented and the state machine transitions to state “S1”.

5.7 Performance Analysis of Constituent Tasks

Figure 5.2 illustrates the constituent tasks in the best MV search process. Following subsections describe the number of clock cycles taken by the ME_ACC for these tasks.

5.7.1 Load Configuration Registers

This initial setting is generally done only once for a given sequence. It includes setting the registers like *BLK_SZ_X*, *BLK_SZ_Y*, *SRCH_PTRN_n_X[i]*, and *SRCH_PTRN_n_Y[i]* etc. It does not affect the throughput of the ME_ACC engine. So, the number of clock cycles required for initial configuration is not considered in the performance analysis of the ME_ACC.

5.7.2 Load Search Window

Consecutive memory locations of the ME_ACC memory can be accessed in back-to-back clock cycles without specifying address for every location being accessed. Using the 16 bits wide Input Bus Interface, neighboring AsAP loads 2 pixels (1 pixel = 8 bits) in one clock cycle. Assuming search range of -16 to $+15$ and the micro-block size of 16x16 pixels, the search window size is (48×48) pixels. It takes 1152 clock cycles to load the search window of size (48×48) pixels in the Reference Frame Pixel Memory.

For the MV search for successive micro-blocks, only 16 pixel columns of new search window data needs to be loaded in the Reference Frame Pixel Memory. It takes 384 clock cycles to load 16×48 pixels of new search window data for successive micro-blocks in a given micro-block row. However, as described in section 5.2, ME_ACC allows memory access to non-interfering banks concurrently with the SAD computation. Hence, when ME_ACC is performing SAD computation on micro-block N, neighboring AsAP, can load search window data for the next micro-block $N + 1$ thus, completely hiding the time required for loading search window in the Reference Frame Pixel Memory.

Table 5.23: Clock cycles required for loading current micro-block

Micro-block size	No. of clock cycles
16x16	128
16x8	64
8x16	64
8x8	32
8x4	16
4x8	32
4x4	16

5.7.3 Load Current Frame Micro-block

As the 16 bits wide Input Bus interface can load 2 pixels per clock cycles, the number of clock cycles required can be given by number of pixels in the micro-block divided by 2. However, there is one exception to this rule. As the memory banks in the ME_ACC are 64 bits wide, we must write all 8 pixels in each memory location using data padding as necessary. Hence, for micro-blocks with pixel width less than 8, the number of clock cycles to load the micro-block memory is given by number of pixel rows in the micro-block multiplied by 4. Table 5.23 lists the number of clock cycles required to load the micro-blocks of various sizes.

5.7.4 SAD Compute Request

The neighboring AsAP issues the *SAD Compute Request* by writing "1" to the register *START_ME* or by writing "1" to the register *CONT_ME*. A single register write transaction takes 4 clock cycles. It takes 2 clock cycles to decode the transaction.

5.7.5 SAD Computation

Table 5.24 gives the breakdown of the clock cycles taken by SAD Computation process.

Table 5.24: Number of clock cycles required for SAD computation

Task	No. of clock cycles
Register decoding & Address generation	5
Memory read & Pixel multiplexing	5
Absolute difference pipe line	No. of rows in the micro-block
Sum of absolute difference	6

5.7.6 Report SAD

The ME_ACC takes 4 cycles to report SAD value and the associated search position to the neighboring AsAP.

5.7.7 Best MV Decision

The neighboring AsAP receives SAD value and associated search position from the ME_ACC. The best MV decision is made by the AsAP, based on the minimum SAD value and any other programmed factors, such as SAD threshold. The neighboring AsAP can issue next *SAD Compute Request* by writing "1" to the register *CONT_ME* without waiting for best MV decision.

Chapter 6

ME_ACC Physical Data

The AsAP2 chip with the ME_ACC was fabricated using 65 nm technology in June 2007. Figure 6.1 shows the die micro-graph of AsAP2, the 167-processor array [2]. The die occupies 39.4 mm^2 and contains 55 million transistors [1]. The square block marked with "Mot. Est." indicates the ME_ACC in the AsAP2. Figure 6.2 shows the die plot of the ME_ACC. Data summarizing the area and preliminary measurements of the ME_ACC are reported in Table 6.1. At a supply voltage of 1.3 V, the ME_ACC operates at the maximum frequency of 938 MHz and dissipates 195 mW. The power dissipation will drop substantially at lower supply voltages.

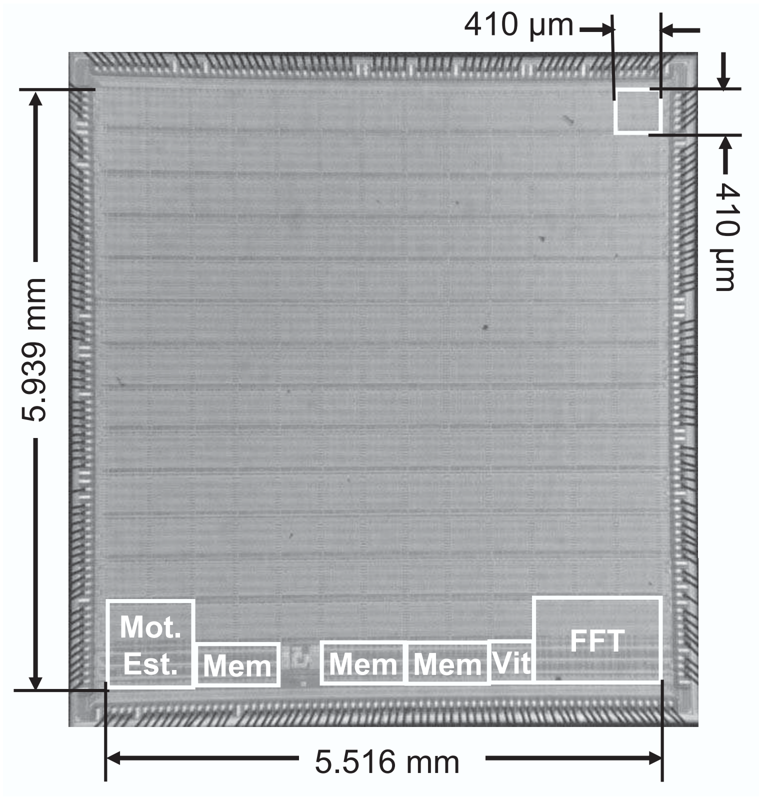


Figure 6.1: Die micro-graph of AsAP2

Table 6.1: Physical parameters of ME_ACC on AsAP2

Parameter name	Value
Technology	65 nm
Total area	0.67 mm^2
SRAM area	137,866 μm^2
Block dimensions	820 μm $\times$ 820 μm
No. of combinational cells	16137
No. of Flip-flops	6248
SRAMs used	8 instances of 64 $\times$ 64 dual port SRAM 2 instances of 16 $\times$ 64 dual port SRAM
Supply voltage	1.3 V
Power	195 mW
Maximum frequency	938 MHz

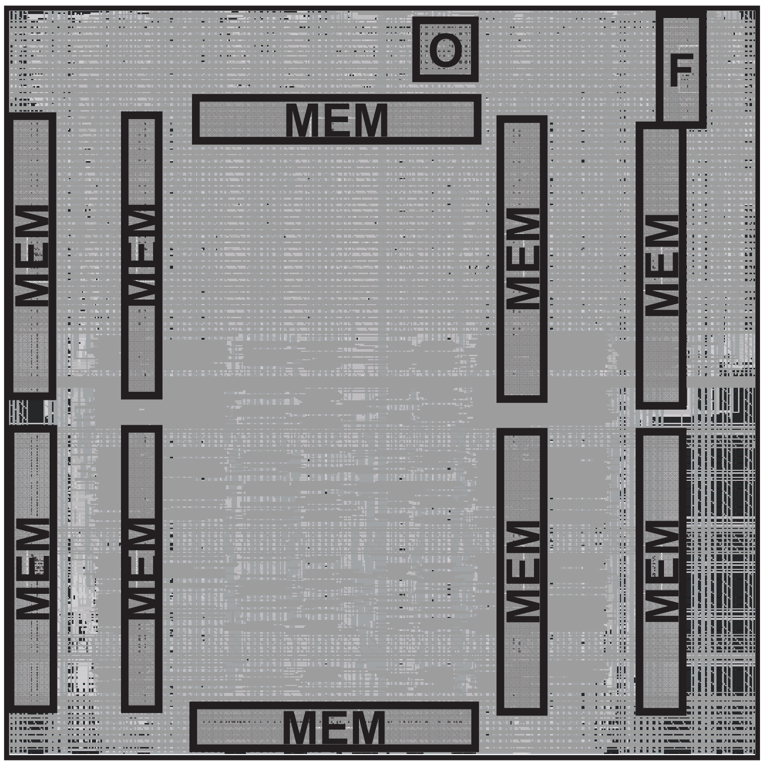


Figure 6.2: Die plot of ME_ACC

Chapter 7

Bit Accurate MATLAB Model for ME_ACC

A bit accurate model, MATLAB function named me_acc_model, has been developed to emulate the ME_ACC functionality. It takes the same input parameters as specified by the Input Bus Protocol of the ME_ACC. It generates the same output parameters as specified by the Output Bus Protocol of the ME_ACC.

The function me_acc_model defines the Me_ACC Reference Frame Pixel Memory, Current Frame Pixel Memory and Configuration Registers as "persistent" variables. MATLAB does not clear the "persistent" variables from memory when the function exits, so their values are retained from one function call to the next. The function me_acc_model is called for every write or read transaction to the ME_ACC Configuration Registers or memory.

The function me_acc_model takes one more input, *reset*, in addition to those defined by the Input Bus Protocol. When the function is called with the input parameter *reset* set to "1", the "persistent" variable for Reference Frame Pixel Memory, the "persistent" variable for Current Frame Pixel Memory and the "persistent" variables for Configuration Registers are reset to "0".

The syntax of the function me_acc_model is shown in the table 7.1.

Table 7.1: me_acc_model input parameters

Parameter index	Parameter name	Description
1	*reset*	If set to 1 reset all persistent variables If set to any value other than 1 Unused
2	*InputControlWord0*	Corresponds to *Control Word 0* of input bus protocol of the ME_ACC. See table 5.18.
3	*InputControlWord1*	Corresponds to *Control Word 1* of input bus protocol of the ME_ACC. See table 5.18.
4	*InputData*	Single dimensional matrix of 16 bit data words. Corresponds to *word 0* to *word n* of input bus protocol. See table 5.18.

7.1 Simulation Results from me_acc_model

To validate the me_acc_model, MV search was performed for two micro-blocks of size 16x16 pixels from 2nd frame of "Carphone" quarter-CIF (QCIF) video sequence using three step search algorithm. The results were bit matched to those generated by the ME_ACC.

The MATLAB code for the me_acc_model is given in Appendix A.

Chapter 8

MV Search Algorithms Simulation

The most important feature of the ME_ACC is its ability to support virtually any search algorithm with least overhead of configuration and memory access time. This feature is important because various efficient algorithms for best MV search consume much less energy than that consumed by the full search algorithm and offer to achieve PSNR close to that achieved by the full search algorithm.

We simulated four MV search algorithms, including the full search algorithm, in order to compare the efficiency of the search algorithms and the efficiency of the ME_ACC in executing those.

The PSNR of a decoded video bit stream depends on multiple factors, such as quantization and bit-rate in addition to the quality of best MV obtained during motion estimation. The ME_ACC module improves the performance of only motion estimation step. Other steps such as quantization, entropy encoding etc in the video compression process are out of the scope of this work. So, to compare the performance of MV search algorithms, we use Mean Absolute Error (MAE) per pixel between the motion compensated frame and the current frame. For a current frame, the MAE per pixel is given by Equation 8.1.

$$MAE = \frac{\sum_{0 \le i \le n} \sum_{0 \le j \le m} (|current\_frame[i][j] - mc\_frame[i][j]|)}{n \times m} \tag{8.1}$$

where, $n =$ *frame width in terms of number of pixels*, $m =$ *frame height in terms of number of pixels*, $current\_frame$ is the current frame, *current_frame*$[i][j]$ gives the current frame pixel value at coordinates (i,j), mc_frame is the motion compensated frame, and $mc\_frame[i][j]$ gives the motion compensated frame pixel value at coordinates (i,j).

8.1 Motion Vector Search Algorithms

Following four MV search algorithms were implemented.

- Full search algorithm: This algorithm searches all 256 positions in the search window of size -8 to $+7$. This gives the best possible MV for the micro-block. However, it takes a very large number of clock cycles. For the full search algorithm using 16x16 block size and search window of size -8 to $+7$, 1,715,292 number of cycles are required per QCIF frame to find the best MV. This algorithm is data-independent.

- Three step search algorithm [36]: At every step, the three step search algorithm tests 9 points, the center *[cx,cy]* and the 8 points around the center given by $[cx-s, cy-s]$, $[cx-s,\ cy]$, $[cx-s,\ cy+s]$, $[cx,\ cy-s]$, $[cx,\ cy]$, $[cx,\ cy+s]$, $[cx+s,\ cy-s]$, $[cx+s,\ cy]$, $[cx+s,\ cy+s]$, where s is the step size. After every step, the center is moved to the point with minimum block distortion measure, *i.e.,* minimum SAD. The initial step size is set to 3 and the step size decremented after every step by 1. Thus, the algorithm is terminated after 3 steps. Figure 8.1 depicts the search patterns in the three step search algorithm. This algorithm searches 25 locations $(9+8+8)$ in the search window of size -4 to $+4$ to find the best MV for a given micro-block. This algorithm is data-independent and searches approximately 10% locations of that searched by the full search algorithm. Hence, the number of clock cycles required is also approximately 10% as that of the full search algorithm. However, Mean Absolute Error (MAE) achieved by the three step search algorithm is moderately higher as compared to that achieved by the full search algorithm.

- Four step search algorithm [37]: The four step search algorithm is slightly different than the three step search algorithm. It starts with the 9 point search over the search window of size -2 to $+2$. In the 2nd and 3rd step, it follows 2 different search patterns based on the location of the point of minimum block distortion, *i.e.,* minimum SAD.

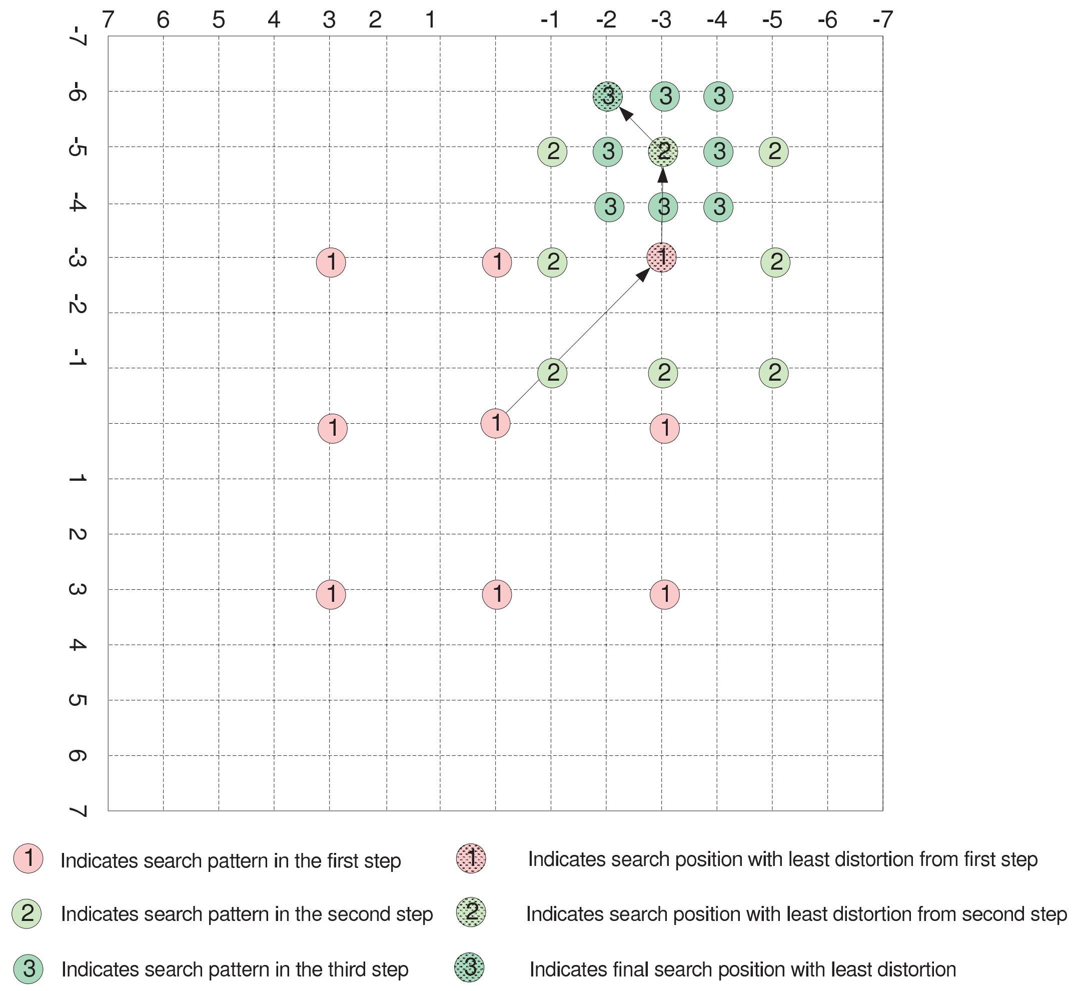

7 6 5 4 3 2 1 -1 -2 -3 -4 -5 -6 -7
-7 -6 -5 -4 -3 -2 -1 1 2 3 4 5 6 7
Indicates search pattern in the first step
Indicates search pattern in the second step
Indicates search pattern in the third step
Indicates search position with least distortion from first step
Indicates search position with least distortion from second step
Indicates final search position with least distortion

Figure 8.1: Three step search algorithm

The final step uses the search window of size −1 to +1. Figure 8.2 depicts the search patterns in the different steps of the four step search algorithm. Two different search paths are shown in Figure 8.3. This algorithm performs the best MV search over the search window of size −7 to +7. This search algorithm is data dependent. It takes less number of clock cycles than those taken by the three step search algorithm and gives similar or slightly better result in terms of MAE per pixel than that given by the three step search algorithm.

- Diamond search algorithm [38]: The diamond search algorithm is very similar to the four step search algorithm. It follows different search patterns in the search steps however, the decision flow remains the same. The search patterns in the diamond search algorithms are shown in Figure 8.4. This algorithm performs the best MV search over the search window of size −7 to +7. It is data dependent. It takes less number of clock cycles than those taken by the four step search algorithm and produces result similar or better than that produced by the four step search algorithm.

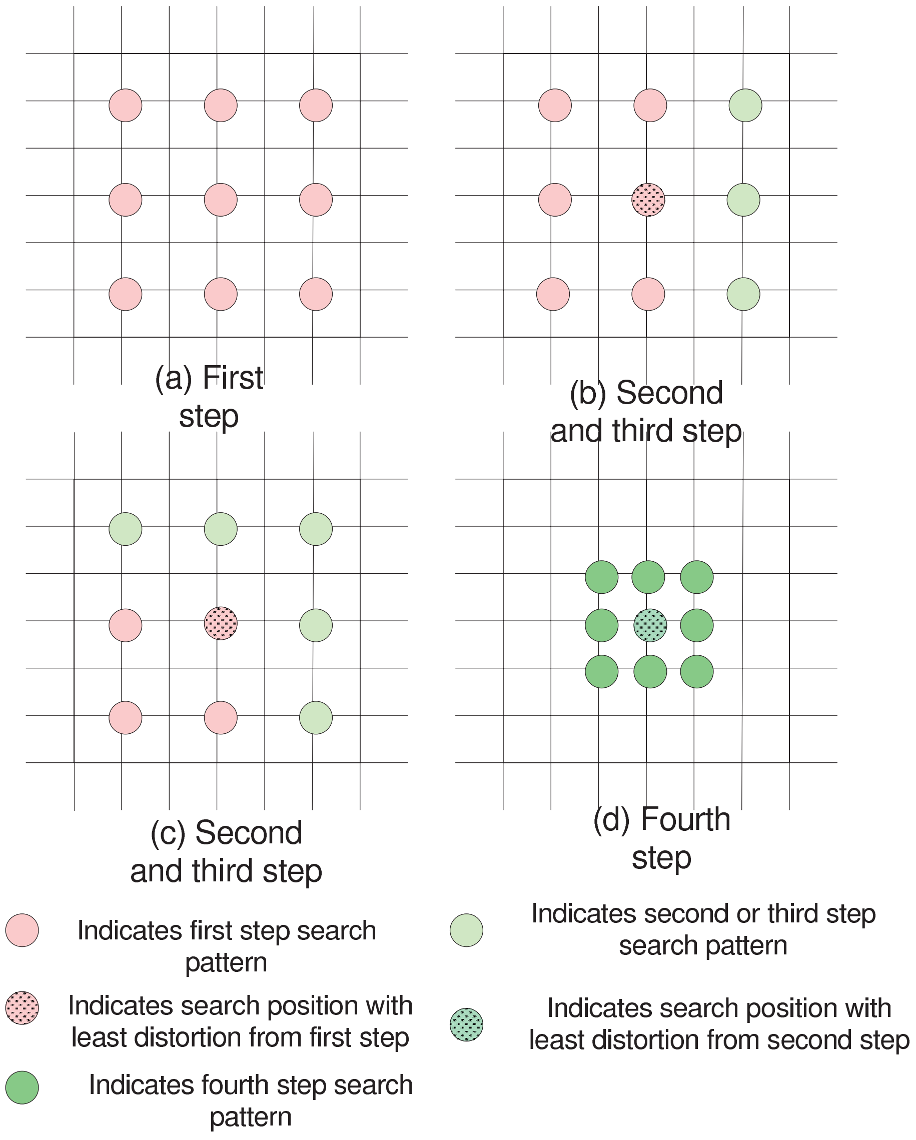

(a) First step
(b) Second and third step
(c) Second and third step
(d) Fourth step
Indicates first step search pattern
Indicates second or third step search pattern
Indicates search position with least distortion from first step
Indicates search position with least distortion from second step
Indicates fourth step search pattern

Figure 8.2: Search patterns in four step search algorithm

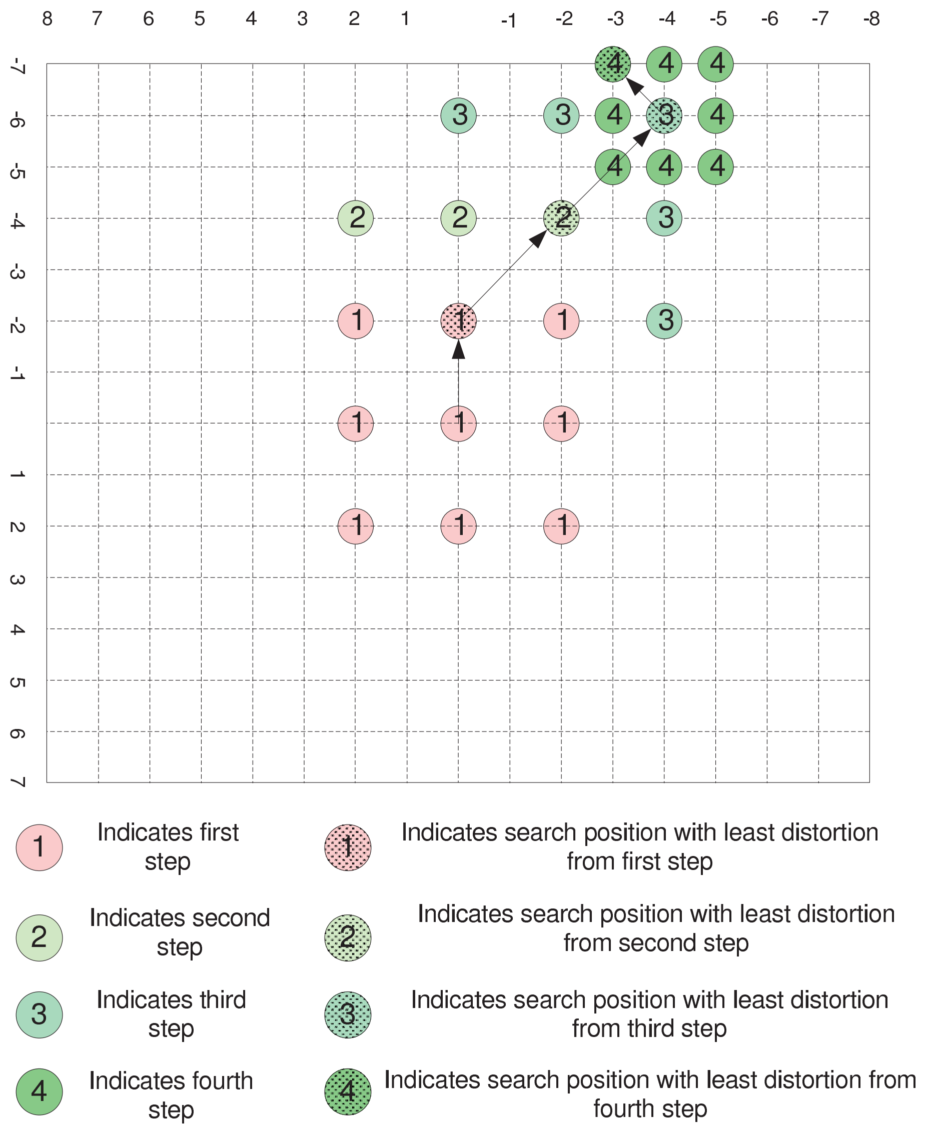

8
7
6
5
4
3
2
1
-1
-2
-3
-4
-5
-6
-7
-8
-7
-6
-5
-4
-3
-2
-1
1
2
3
4
5
6
7
Indicates first step
Indicates second step
Indicates third step
Indicates fourth step
Indicates search position with least distortion from first step
Indicates search position with least distortion from second step
Indicates search position with least distortion from third step
Indicates search position with least distortion from fourth step

Figure 8.3: Search path in four step search algorithm

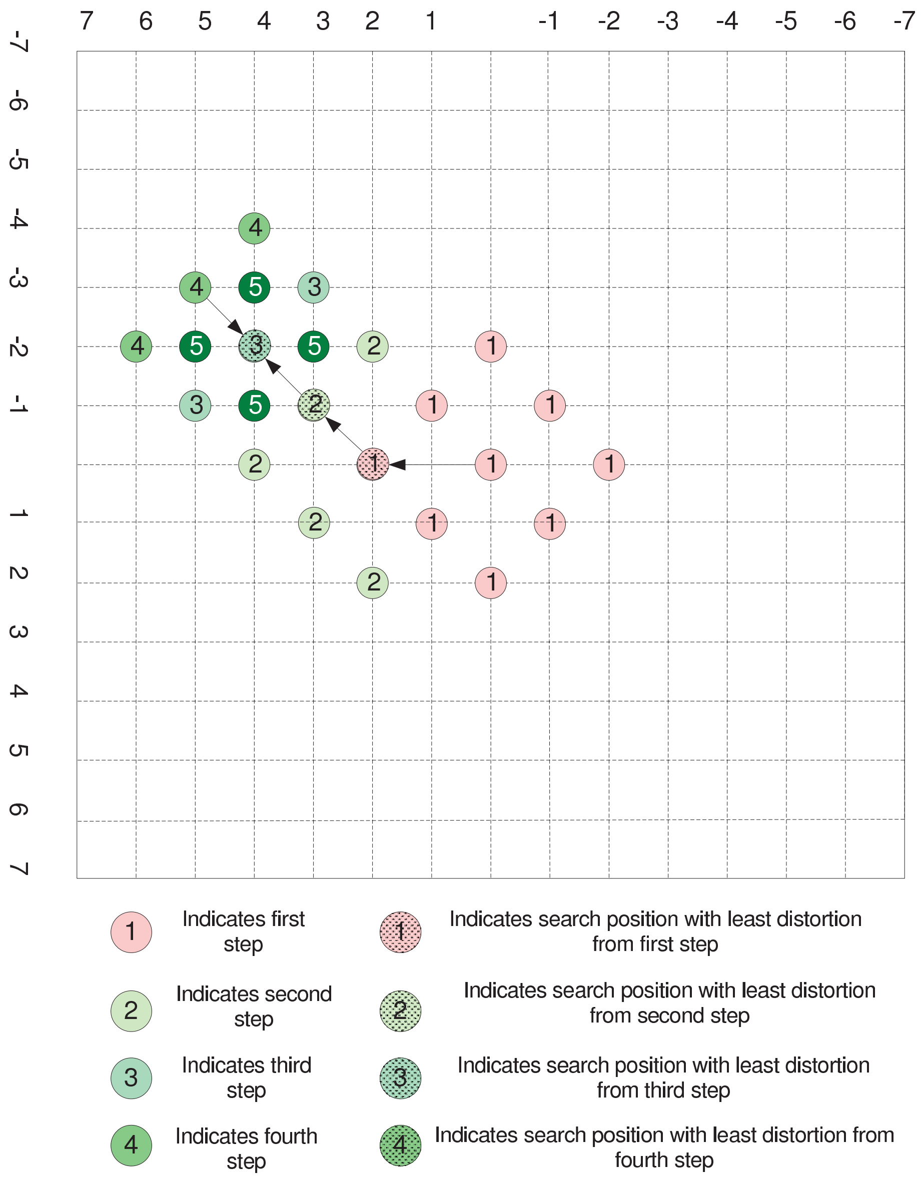

7 6 5 4 3 2 1 -1 -2 -3 -4 -5 -6 -7
-7 -6 -5 -4 -3 -2 -1 1 2 3 4 5 6 7
1 Indicates first step
2 Indicates second step
3 Indicates third step
4 Indicates fourth step
1 Indicates search position with least distortion from first step
2 Indicates search position with least distortion from second step
3 Indicates search position with least distortion from third step
4 Indicates search position with least distortion from fourth step

Figure 8.4: Diamond search algorithm

Table 8.1: Percentage increase in MAE obtained by efficient algorithms with respect to MAE obtained by full search algorithm

Video sequence	Three step search algorithm	Four step search algorithm	Diamond search algorithm
Carphone	+2.943%	+3.237%	+1.372%
Coastguard	+0.302%	+0.313%	+0.315%
Foreman	+7.875%	+5.481%	+3.674%
Mobile	+0.012%	+0.012%	+0.012%
News	+0.125%	+0.064%	+0.065%

8.2 Performance of MV search algorithms

As the full search algorithm is the most extensive search algorithm, it returns the highest quality best MV (least MAE). The performance of 3 MV search algorithms with respect to that of the full search algorithm is presented in Table 8.1 as a percentage increase in the MAE over the MAE achieved by the full search algorithm. The MAE values given in the Table 8.1 are the average of MAE per pixel values computed over first 20 frames of the corresponding QCIF sequence. The micro-block size of 16x16 pixels was used for the best MV search.

The full search algorithm performs the most extensive search and hence, requires the maximum number of clock cycles. The efficient search algorithms require a lot less number of clock cycles and their performances are similar or slightly inferior than that of the full search algorithm. As shown in the Table 8.1, the quality of the best MV obtained by efficient algorithms is inferior by less than 10% as compared to that obtained by the full search algorithm. Table 8.2 presents the number of clock cycles required for the 3 MV search algorithms as a percentage of number of clock cycles required for the full search algorithm. The percentage of the number of clock cycles is computed over the first 20 frames of the respective QCIF video sequence. Figures 8.5, 8.6, 8.7, 8.8, and 8.9 illustrate the number of clock cycles required and the MAE per pixel for five QCIF video sequences: “Carphone”, “Coastguard”, “Foreman”, “Mobile”, and “News”.

Various factors such as the micro-block size and the search algorithm itself affect the quality of the best MV and the computational effort required to achieve it. The perfor-

Table 8.2: Number of clock cycles taken by efficient algorithms per frame as a percentage of that taken by full search algorithm

Video sequence	Three step search algorithm	Four step search algorithm	Diamond search algorithm
Carphone	10.178%	7.259%	5.938%
Coastguard	10.177%	6.997%	5.469%
Foreman	10.177%	7.476%	6.162%
Mobile	10.177%	6.941%	5.395%
News	10.177%	6.920%	5.369%

mance of a MV search algorithm depends on the motion and texture characteristics of the video sequence. For a low-motion/low-texture sequence like "News", the least expensive search (diamond search with micro-block size of 16x16 pixels) gives MV quality that is very close to that achieved by the most expensive search (full search with micro-block size of 8x8 pixels). However, for a high-motion/high-texture video sequence like "Foreman", the MV search quality deteriorates significantly with a less expensive search algorithm. The most significant advantage of the ME_ACC is that, user can adapt search algorithm and other motion estimation parameters, such as micro-block size and search window size, depending on the video characteristics to achieve the best trade off between the MV search quality and the computational complexity. The distinctive advantage of the programmable features of the ME_ACC is highlighted by Tables 8.3, 8.4, 8.5, 8.6, and 8.7. It can be clearly seen that, with micro-block size of 8x8, efficient algorithms can achieve better MV search and use less number of clock cycles as compared to those used by the full search algorithm with micro-block size of 16x16.

Table 8.3: Carphone: Motion vector search performance with varying micro-block size

Micro-block size	Search algorithm	MAE per pixel	# Clock cycles per QCIF frame
16x16	Full search	4.301	1,715,292
16x16	Three step search algorithm	4.428	174,582
16x16	Four step search algorithm	4.441	124,510
16x16	Diamond search algorithm	4.360	101,860
8x8	Full search algorithm	3.909	5,558,640
8x8	Three step search algorithm	4.136	571,128
8x8	Four step search algorithm	4.147	425,553
8x8	Diamond search algorithm	4.024	358,725

Table 8.4: Coastguard: Motion vector search performance with varying micro-block size

Micro-block size	Search algorithm	MAE per pixel	# Clock cycles per QCIF frame
16x16	Full search	3.885	1,715,292
16x16	Three step search algorithm	3.896	174,582
16x16	Four step search algorithm	3.897	120,010
16x16	Diamond search algorithm	3.897	93,810
8x8	Full search algorithm	3.652	5,558,640
8x8	Three step search algorithm	3.682	571,128
8x8	Four step search algorithm	3.687	405,038
8x8	Diamond search algorithm	3.678	325,350

Table 8.5: Foreman: motion vector search performance with varying micro-block size

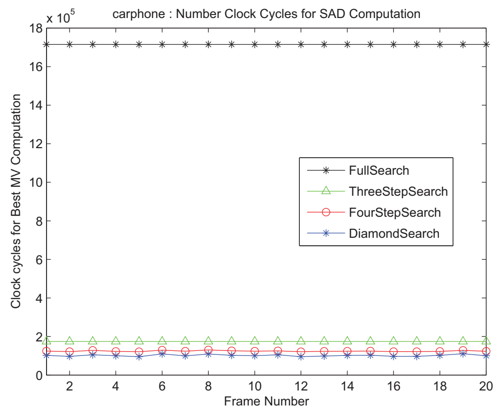
Micro-block size	Search algorithm	MAE per pixel	# Clock cycles per QCIF frame
16x16	Full search	4.910	1,715,292
16x16	Three step search algorithm	5.297	174,582
16x16	Four step search algorithm	5.179	128,234
16x16	Diamond search algorithm	5.091	105,702
8x8	Full search algorithm	4.486	5,558,640
8x8	Three step search algorithm	5.092	571,128
8x8	Four step search algorithm	4.929	435,108
8x8	Diamond search algorithm	4.772	363,538

Table 8.6: Mobile: Motion vector search performance with varying micro-block size

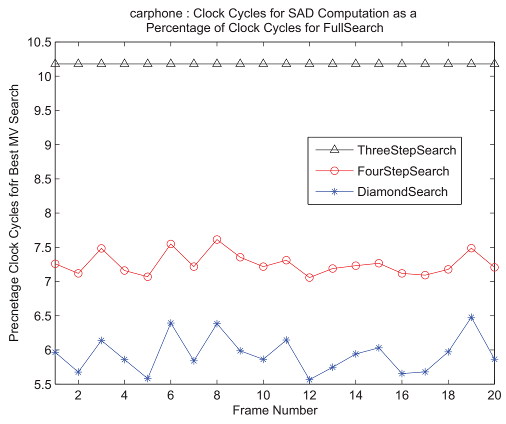
Micro-block size	Search algorithm	MAE per pixel	# Clock cycles per QCIF frame
16x16	Full search	10.164	1,715,292
16x16	Three step search algorithm	10.165	174,582
16x16	Three step search algorithm	10.165	119,059
16x16	Diamond search algorithm	10.165	92,544
8x8	Full search algorithm	9.860	5,558,640
8x8	Three step search algorithm	9.910	571,128
8x8	Four step search algorithm	9.891	396,185
8x8	Diamond search algorithm	9.903	313,992

Table 8.7: News: Motion vector search performance with varying micro-block size

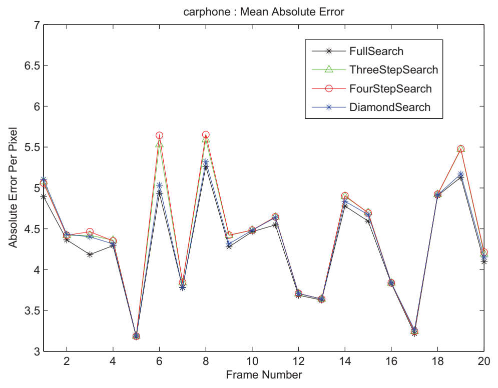
Micro-block size	Search algorithm	MAE per pixel	# Clock cycles per QCIF frame
16x16	Full search	1.723	1,715,292
16x16	Three step search algorithm	1.725	174,582
16x16	Four step search algorithm	1.724	118,704
16x16	Diamond search algorithm	1.724	92,086
8x8	Full search algorithm	1.617	5,558,640
8x8	Three step search algorithm	1.639	571,128
8x8	Four step search algorithm	1.636	396,604
8x8	Diamond search algorithm	1.630	315,018



(a) No. of clock cycles for Carphone MV search

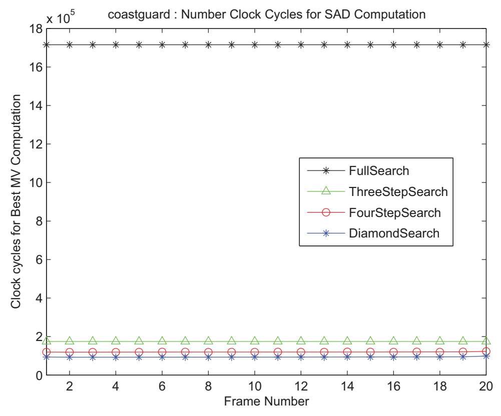


(b) No. of clock cycles for MV search as percentage of full search

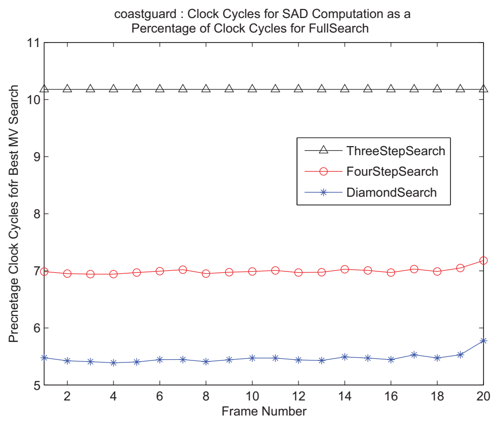


(c) Mean absolute error from Carphone MV search

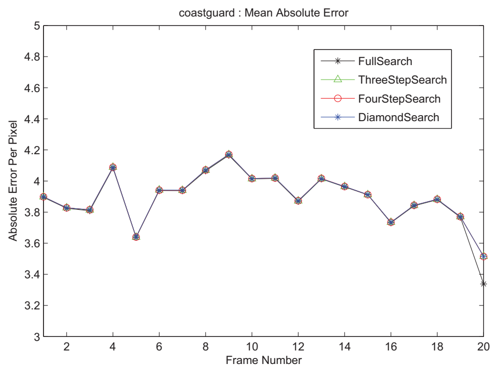
Figure 8.5: Carphone MV search using different algorithms



(a) No. of clock cycles for Coastguard MV search

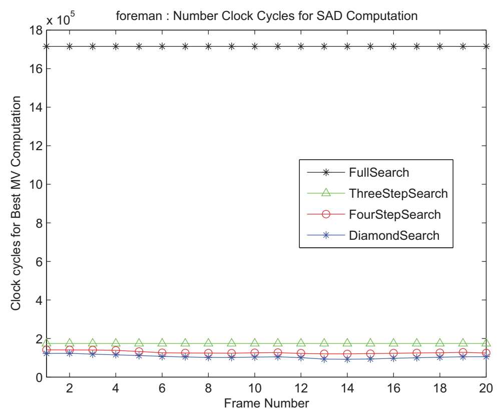


(b) No. of clock cycles for MV search as percentage of full search

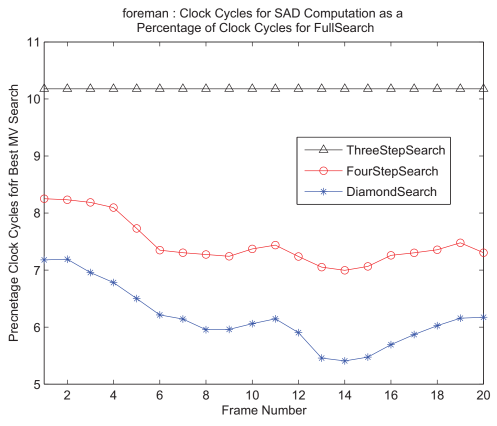


(c) Mean absolute error from Coastguard MV search

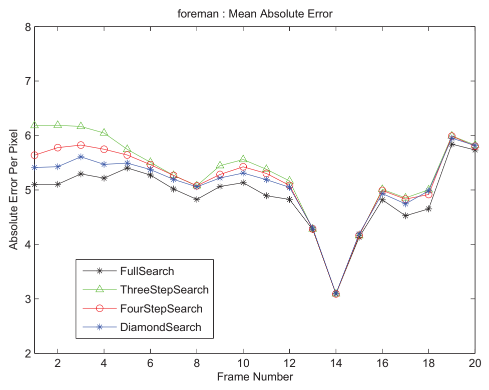
Figure 8.6: Coastguard MV search using different algorithms



(a) No. of clock cycles for Foreman MV search

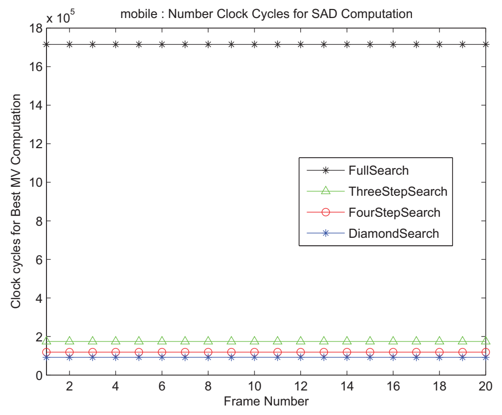


(b) No. of clock cycles for MV search as percentage of full search

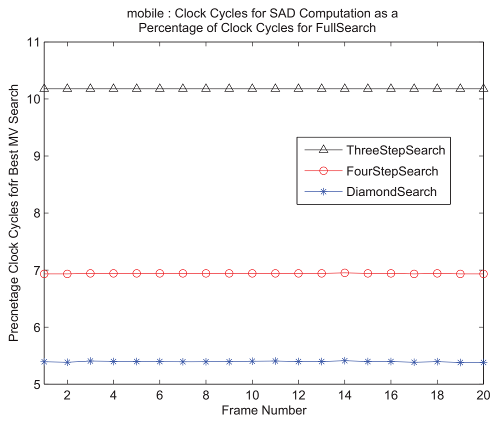


(c) Mean absolute error from Foreman MV search

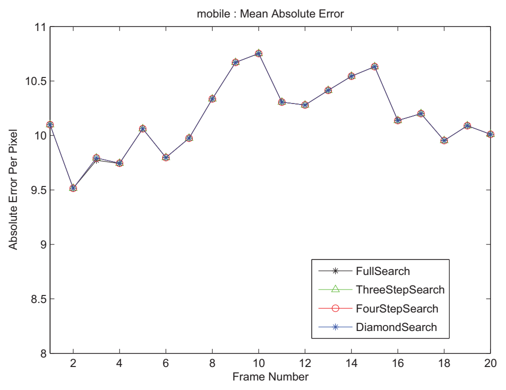
Figure 8.7: Foreman MV search using different algorithms



(a) No. of clock cycles for Mobile MV search

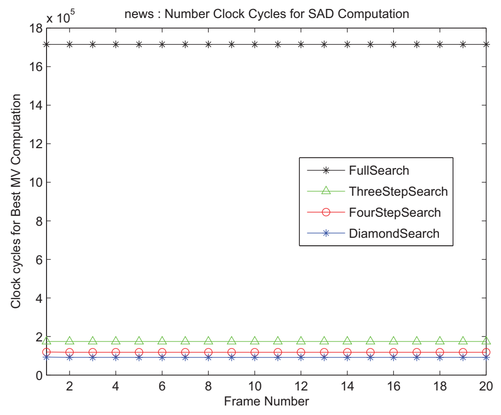


(b) No. of clock cycles for MV search as percentage of full search

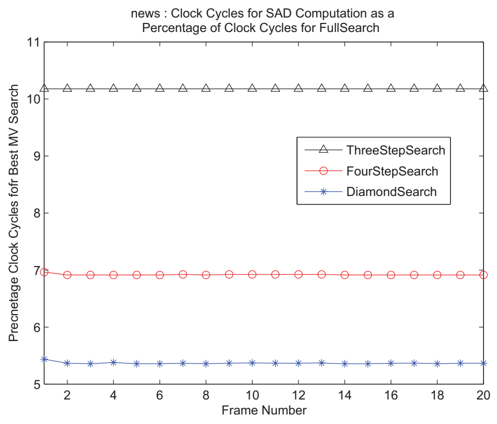


(c) Mean absolute error from Mobile MV search

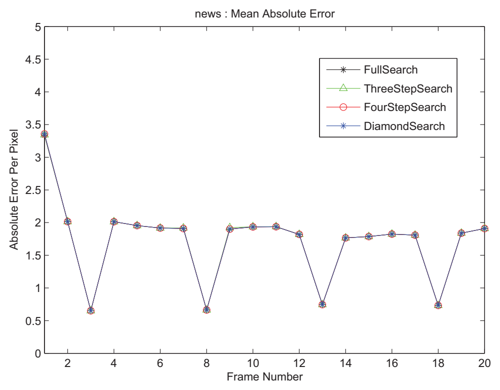
Figure 8.8: Mobile MV search using different algorithms



(a) No. of clock cycles for News MV search



(b) No. of clock cycles for MV search as percentage of full search



(c) Mean absolute error from News MV search

Figure 8.9: News MV search using different algorithms

Chapter 9

Conclusion

9.1 Contributions

The contributions of this work include architecting, RTL coding, synthesizing, and verifying the programmable Motion Estimation Accelerator, ME_ACC, that supports a multitude of video compression standards and MV search algorithms. The ME_ACC supports programmable search window, micro-block size, and MV search algorithm and supports a large range of frame size. It also allows flexible best MV criterion based on SAD. Various MV search algorithms are implemented on the ME_ACC during the course of research.

9.2 Future Work

There are three primary categories where future effort can be applied to this work. Firstly, a few modifications can be done to the implementation in order to improve the performance of the ME_ACC. Secondly, clock gating and other power saving techniques can be to applied to reduce the power consumption of the ME_ACC. And lastly, additional features such as programmable interpolation filter can be implemented.

9.2.1 Performance Improvement

The Finite State Machine, the Address Generation Unit, and the SAD Computation Unit are coded in such a way that the design can be synthesized to support maximum operating frequency. It might have resulted in some logic paths with very small path delays. These paths can be combined with neighboring stages in order to reduce the number of clock cycles taken by the ME_ACC for SAD computation, without affecting the maximum operating frequency supported.

One more 16×16 pixel memory can be added to hold one more current frame micro-block. This will eliminate the time required to load the current frame micro-block as the alternate memories can be loaded with the next current frame micro-block pixels when the ME_ACC is computing SAD for previously loaded current frame micro-block. To offset the increase in memory size, the depth of the Reference Frame Pixel memory can be reduced from 64 to 48 without significantly affecting the performance of the ME_ACC.

9.2.2 Power Saving

- Multiple power saving techniques can be applied to reduce the power consumption of the ME_ACC. Few examples are, clock gating-off the memory banks when no read/write operation is being performed and using synthesis tool supported clock-gating to convert data multiplexers on the D-input of the pipeline stages to clock gates etc.

- The search pattern registers, *SRCH_PTRN_n_X* and *SRCH_PTRN_n_Y*, can be implemented using a SRAM instead of flip-flops.

9.2.3 Feature Enhancement

The ME_ACC can be enhanced to add more features.

- The addition of a programmable *SAD threshold* value that would allow the accelerator to make consecutive SAD calculations without individual “Continue” commands from the controlling processor. The basic idea is that the accelerator would continue SAD calculations along the search pattern path until the SAD calculation reached a value less than the threshold.

This would decrease total power consumption since the controlling processor would not have to check each SAD value, or issue a Continue instruction for each SAD calculation.

It would also increase the throughput of the accelerator since it would not need to pause and wait between SAD calculations.

- The current data-path calculates "sum of absolute differences" terms for up-to 16 pixels per row of pixels. Micro-blocks that are 4 or 8 pixels wide do not use the entire data-path. A proposed enhancement is to compute four 4-pixel-wide blocks or two 8-pixel-wide blocks in parallel.

  This enhancement would dramatically increase throughput for smaller block widths, but would require the addition of a small amount of hardware. The largest impact may be that carry-propagate adders would be needed for the shorter-width outputs that may increase the critical path.

- The addition of a "mask" and offset pointer for the current frame pixel memory that would allow different micro-blocks within the memory to be used as current blocks.

  For example, the entire 16×16 pixel memory could be filled and SAD values for the four micro-blocks of size 8x8 pixels could be calculated without reloading the memory, by only changing the offset pointer value.

- Programmable sub-pixel interpolater to support half-pixel and quarter-pixel motion estimation.

- Save intermediate 4x4 micro-block SAD values for any given micro-block size and report to neighboring AsAP, if requested. This will add performance improvement to the search algorithms employing dynamic mode selection

- Save the motion-compensated pixel values for the micro-block and report to neighboring AsAP if requested. When neighboring AsAP determines the best motion vector, it can simply read out the motion-compensated micro-block and pass it on to transform coding. A lot of redundant computations of the neighboring AsAP will be eliminated.

Appendix A

Matlab Code for ME_ACC Model

```
%--------------------------------------------------------------------------
% Description :
% Function to model behavior of the Motion Estimation Accelerator, ME\_ACC,
%on AsAP2
% The inputs parameters to the model are same as the input data dirven by
% neighboring AsAP to ME\_ACC. One extra input parameter ‘‘reset’’, is used
% to reset the values of persistent variables.
% Configuration registers of the ME\_ACC are modeled by using persistent
% variables
%-------------------------------------------------------------------------
% Revision History : Rev 1.0 05/15/2009
%-------------------------------------------------------------------------

function [OutputControlWord, OutputData] =
         me_acc_model (reset, InputControlWord0, InputControlWord1, InputData)

% Variable Declarations

%OutputData = zeros(1, 2048);

% ME_ACC Configuration Registers

persistent blk_sz_x;
if ((isempty(blk_sz_x)) || (reset==1))
    blk_sz_x = zeros(1,1);
end

persistent blk_sz_y;
if ((isempty(blk_sz_y)) || (reset==1))
    blk_sz_y = zeros(1,1);
end
```

```
persistent srch_pos_cnt_array;
if ((isempty(srch_pos_cnt_array)) || (reset==1))
    srch_pos_cnt_array = zeros(1,4);
end

persistent srch_ptrn_cnt;
if ((isempty(srch_ptrn_cnt)) || (reset==1))
    srch_ptrn_cnt = zeros(1,1);
end

persistent center_x;
if ((isempty(center_x)) || (reset==1))
    center_x = zeros(1,1);
end

persistent center_y;
if ((isempty(center_y)) || (reset==1))
    center_y = zeros(1,1);
end

persistent srch_ptrn_x_array;
if ((isempty(srch_ptrn_x_array)) || (reset==1))
    srch_ptrn_x_array = zeros(4,64);
end

persistent srch_ptrn_y_array;
if ((isempty (srch_ptrn_y_array)) || (reset==1))
    srch_ptrn_y_array = zeros(4, 64);
end

persistent ref_blk_mem;
if ((isempty(ref_blk_mem)) || (reset==1))
    ref_blk_mem = zeros(64, 64);
end

persistent org_blk_mem;
if ((isempty(org_blk_mem)) || (reset==1))
    org_blk_mem = zeros(16, 16);
end


% Single Pulse Registers
start_me = 0;
cont_me = 0;
abndn_me = 0;


% Local Variables
```

```
%persistent srch_ptrn_idx;
%if ((isempty(srch_ptrn_idx)) || (reset==1))
%    srch_ptrn_idx = 0;
%end

persistent srch_pos_idx;
if ((isempty(srch_pos_idx)) || (reset==1))
    srch_pos_idx = 0;
end

srch_pos_x = 0;
srch_pos_y = 0;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  Register Read Write Operation Logic
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

OutputControlWord = 0;
InCtrlWd0 = InputControlWord0;
InCtrlWd1 = InputControlWord1;

InData = InputData;

% Bit 15 of InputControlWord0 indicates valid transaction
InCtrlWd0_Bit15 = bitand(InCtrlWd0, 2.^15)/ 2.^15;

% Bit 14 of InputControlWord0 inidicates
% (0=write, 1=read) operation
InCtrlWd0_Bit14 = bitand(InCtrlWd0, 2.^14)/ 2.^14;

% Bit 13 of InputControlWord0 indicates
% (1=Mem Aceess, 0= Reg Access) operation
InCtrlWd0_Bit13 = bitand(InCtrlWd0, 2.^13)/ 2.^13;

% Bit 12 of InputControlWord0 indicates
% (1=Columnwise memory access, 0=Rowwise memory access)
InCtrlWd0_Bit12 = bitand(InCtrlWd0, 2.^12)/ 2.^12;

% Bits [11:0] of InputControlWord0 indicate
%Transfer Length
TransferLength  = bitand(InCtrlWd0, (2.^12)-1);

if (InCtrlWd0_Bit13 == 1)
   Mem_Acc_St_Addr = InputControlWord1;
end
```

```
[m,n] = size(InputData);

if (m ~= 1)
    error('InputData must be a One Dimentional Array')
end

DataWordcnt = 1;
EightPixelRow = zeros(1,8);
EightPixelColumn = zeros(8,1);


if (InCtrlWd0_Bit15 == 1)       % Valid Input Control Word0
 if (InCtrlWd0_Bit14 == 0)      % Write Operation
  if (InCtrlWd0_Bit13 == 1)     % Memory Write
      OutputControlWord = 0;

   for DataWordCnt=1:TransferLength
    mem_addr = Mem_Acc_St_Addr + ((DataWordCnt-1) * 2);
    mem_addr_bit15 = bitand(mem_addr, 2.^15)/2.^15;
    mem_addr_bit2_1 = bitand(mem_addr, 2.^2+2.^1)/2.^1;
    bank_sel = bitand(mem_addr, (2.^11+2.^10+2.^9))/2.^9;
    row_sel  = bitand(mem_addr,
                  (2.^8+2.^7+2.^6+2.^5+2.^4+2.^3))/2.^3;

    DataByte1 = bitand(InputData(DataWordCnt),
                      (2.^8)-1);
    DataByte2 = bitand(InputData(DataWordCnt),
                      ((2.^16)-1) - ((2.^8)-1))/2.^8;

    switch mem_addr_bit2_1
        case 0
            EightPixelRow(1) = DataByte1;
            EightPixelRow(2) = DataByte2;
        case 1
            EightPixelRow(3) = DataByte1;
            EightPixelRow(4) = DataByte2;
        case 2
            EightPixelRow(5) = DataByte1;
            EightPixelRow(6) = DataByte2;
        case 3
            EightPixelRow(7) = DataByte1;
            EightPixelRow(8) = DataByte2;
    end

    % Valid memory address
    if (mem_addr_bit15 == 1)
     mem_addr_bit13_12 = bitand(mem_addr, (2.^13+2.^12))/2.^12;
    % RefBlk memory address
     if (mem_addr_bit13_12 == 1)
      % Write to memory after receving
      % all 8 pixels in the bank location
      if (mem_addr_bit2_1 == 3)
       ref_blk_mem(row_sel+1, (bank_sel*8)+1:(bank_sel*8)+8) =
           EightPixelRow;
      else
      end
     else
      % OrgBlk memory address
      if (mem_addr_bit13_12 == 2)
       % Write to memory after receving all 8 pixels in
       % the bank location
       if (mem_addr_bit2_1 == 3)
        org_blk_mem(row_sel+1, (bank_sel*8)+1:(bank_sel*8)+8) =
            EightPixelRow;
       else
       end
      else                        % Invalid Address
      end
     end
    else                           % Invalid memory address
    end
   end
   % Temporary code to confirm memory is loaded correctly

   if (mem_addr_bit13_12 == 1)  % RefBlk memory address

    fr_x_offset = 0;
    fr_y_offset = 0;
    seq_name = 'qcif/carphone';
    color_idx = 1;
    ref_fr_idx = 0;
    color_name = ['Y' 'U' 'V'];

    ref_fr_name = sprintf('..\\..\\BAAS_LINUX.tar\\BAAS_LINUX\\work1\\
                      me_acc\\dv3\\sequences\\matlab\\%s\\Fr%s_%d.m',
                      seq_name, color_name(color_idx), ref_fr_idx);
    ref_fr = dlmread(ref_fr_name);

    if(bank_sel == 5)
     if (ref_blk_mem(1:48, 1:48) ==
         ref_fr(fr_x_offset+1:fr_x_offset+48,
                fr_y_offset+1 : fr_y_offset+48))
      disp('ref_blk_mem loaded correctly');
     else
      disp('Error ! ref_blk_mem not loaded correctly');
     end
    end
   end

   if (mem_addr_bit13_12 == 2)  % OrgBlk memory address

    fr_x_offset = 0;
    fr_y_offset = 0;
    seq_name = 'qcif/carphone';
    color_idx = 1;
    fr_idx = 1;
    color_name = ['Y' 'U' 'V'];

    fr_name = sprintf('..\\..\\BAAS_LINUX.tar\\BAAS_LINUX\\work1\\
                       me_acc\\dv3\\sequences\\matlab\\%s\\Fr%s_%d.m',
                       seq_name, color_name(color_idx), fr_idx);
    fr = dlmread(fr_name);

    if(bank_sel == 1)
     if (org_blk_mem(1:16, 1:16) ==
         fr(fr_x_offset+1:fr_x_offset+16,
            fr_y_offset+1 : fr_y_offset+16))
         disp('org_blk_mem loaded correctly');
     else
         disp('Error ! org_blk_mem not loaded correctly');
     end
    end
   end


  else                                  % Register Write
   OutputControlWord = 0;
   for DataWordCnt=1:TransferLength
    reg_addr = InputData(((DataWordCnt-1)*2)+1);
    reg_data = InputData(DataWordCnt*2);
    reg_addr_bit15_12 = bitand(reg_addr,
         (2.^15 + 2.^14 + 2.^13 + 2.^12))/2.^12;
    reg_addr_bit11_0  = bitand(reg_addr,
        ((2.^16-1) - (2.^15 + 2.^14 + 2.^13 + 2.^12)));
    array8_idx = bitand(reg_addr, (2.^3 + 2.^2 + 2.^1))/ 2.^1;
    array16_idx = bitand(reg_addr, (2.^4 + 2.^3 + 2.^2 + 2.^1))/2.^1;
    array256_idx = bitand(reg_addr,
        (2.^7 + 2.^6 + 2.^5 + 2.^4 + 2.^3 + 2.^2 + 2.^1 + 2.^0));
    if (reg_addr_bit15_12 == 8)

     if (reg_addr_bit11_0 == 2)
      blk_sz_x = reg_data;
     end
     if (reg_addr_bit11_0 == 4)
      blk_sz_y = reg_data;
     end

     if ((reg_addr_bit11_0 >= 256) && (reg_addr_bit11_0 <= 511))
      ptrn_sel = bitand(reg_addr, (2.^7 + 2.^6))/2.^6;
      pos_idx  = bitand(reg_addr,
                  (2.^5 + 2.^4 + 2.^3 + 2.^2 + 2.^1 + 2.^0));
      srch_ptrn_x_array(ptrn_sel+1, pos_idx+1) = reg_data;
     end

     if ((reg_addr_bit11_0 >= 768) && (reg_addr_bit11_0 <= 1023))
      ptrn_sel = bitand(reg_addr, (2.^7 + 2.^6))/2.^6;
      pos_idx  = bitand(reg_addr,
                   (2.^5 + 2.^4 + 2.^3 + 2.^2 + 2.^1 + 2.^0));
      srch_ptrn_y_array(ptrn_sel+1, pos_idx+1) = reg_data;
     end

     if ((reg_addr_bit11_0 >= 1280) && (reg_addr_bit11_0 <= 1295))
      srch_pos_cnt_array(array8_idx+1) = reg_data;
     end

     if (reg_addr_bit11_0 == 1344)
      srch_ptrn_cnt = reg_data;
     end

     if (reg_addr_bit11_0 == 1362)
      center_x = bitand(reg_data, 255);
      center_y = bitand(reg_data, 65280)/2.^8;
     end

     if (reg_addr_bit11_0 == 1360)
      start_me = reg_data;
     end

     if (reg_addr_bit11_0 == 1364)
      cont_me = reg_data;
     end

      if (reg_addr_bit11_0 == 1366)
       abndn_me = reg_data;
     end

    end

   end
  end
 else                                      % Read Operation
  OutputControlWord = 8192;
  if (InCtrlWd0_Bit13 == 1)             % Memory Read
   for DataWordCnt=1:TransferLength
    mem_addr = Mem_Acc_St_Addr + ((DataWordCnt-1) * 2);
    mem_addr_bit15 = bitand(mem_addr, 2.^15)/2.^15;
    mem_addr_bit13_12 = bitand(mem_addr, (2.^13+2.^12))/2.^12;
    mem_addr_bit2_1 = bitand(mem_addr, 2.^2+2.^1)/2.^1;
    bank_sel = bitand(mem_addr, (2.^11+2.^10+2.^9))/2.^9;
    row_sel  = bitand(mem_addr, (2.^8+2.^7+2.^6+2.^5+2.^4+2.^3))/2.^3;
    word_sel = bitand(mem_addr, 2.^2+2.^1)/2.^1;
    if (mem_addr_bit15 == 1)       % Valid memory address
     if (mem_addr_bit13_12 == 1)  % RefBlk memory address
      ReadDataByte1 = ref_blk_mem(row_sel+1,
                             (bank_sel*8)+1+(word_sel*2));
      ReadDataByte2 = ref_blk_mem(row_sel+1,
                             (bank_sel*8)+1+(word_sel*2)+1);
      OutputData(DataWordCnt) = (ReadDataByte2* 2.^8) +
                              ReadDataByte1;
     else
      ReadDataByte1 = org_blk_mem(row_sel+1, (bank_sel*8)+1+word_sel);
      ReadDataByte2 = org_blk_mem(row_sel+1, (bank_sel*8)+1+word_sel+1);
      OutputData(DataWordCnt) = (ReadDataByte2* 2.^8) + ReadDataByte1;
     end
    end
   end

  else                                  % Register Read
   for DataWordCnt=1:TransferLength
    reg_addr = InputData(((DataWordCnt-1)*2)+1);
    reg_data = InputData(DataWordCnt*2);
    reg_addr_bit15_12 = bitand(reag_addr,
          (2.^15 + 2.^14 + 2.^13 + 2.^12))/2.^12;
    reg_addr_bit11_0  = bitand(reg_addr,
         ((2.^16-1) - (2.^15 + 2.^14 + 2.^13 + 2.^12)));
    array8_idx = bitand(reg_addr, (2.^3 + 2.^2 + 2.^1));
    array16_idx = bitand(reg_addr, (2.^4 + 2.^3 + 2.^2 + 2.^1));
    array256_idx = bitand(reg_addr,
         (2.^7 + 2.^6 + 2.^5 + 2.^4 + 2.^3 + 2.^2 + 2.^1 + 2.^0));
    if (reg_addr_bit15_12 == 8)

     if (reg_addr_bit11_0 == 2)
      ReadRegData = blk_sz_x;
     end

     if (reg_addr_bit11_0 == 4)
      ReadRegData = blk_sz_y;
     end

     if ((reg_addr_bit11_0 >= 256) && (reg_addr_bit11_0 <= 511))
      ptrn_sel = bitand(reg_addr, (2.^7 + 2.^6))/2.^6;
      pos_idx  = bitand(reg_addr,
                  (2.^5 + 2.^4 + 2.^3 + 2.^2 + 2.^1 + 2.^0));
      ReadRegData = srch_ptrn_x_array(ptrn_sel, pos_idx);
     end

     if ((reg_addr_bit11_0 >= 768) && (reg_addr_bit11_0 <= 1023))
      ptrn_sel = bitand(reg_addr, (2.^7 + 2.^6))/2.^6;
      pos_idx  = bitand(reg_addr,
                  (2.^5 + 2.^4 + 2.^3 + 2.^2 + 2.^1 + 2.^0));
      ReadRegData = srch_ptrn_y_array(ptrn_sel, pos_idx);
     end

     if ((reg_addr_bit11_0 >= 1280) && (reg_addr_bit11_0 <= 1295))
      ReadRegData = srch_pos_cnt_array(array8_idx);
     end

     if (reg_addr_bit11_0 == 1344)
      ReadRegData = srch_ptrn_cnt;
     end

     if (reg_addr_bit11_0 == 1362)
      ReadRegData = (center_y * 2.^8) + center_x;
     end

    end
    OutputData(DataWordCnt) = ReadRegData;
   end
  end
 end
 % Invalid Input Control Word 0, Ignore function call
else

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  SAD Computation Logic
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Set appropriate registers and variables if start_me
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if (start_me == 1)
 srch_pos_idx = 0;
```

```
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Set appropriate registers and variables if con_me
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if (cont_me == 1)
  % Go to next search position in the search pattern
 srch_pos_idx = srch_pos_idx + 1;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Perforam SAD computation and return
%% appropriate results
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if ((start_me == 1) || (cont_me == 1))
 % All counters/array-indices in verilog start from 0
 % All counters/array-incdices in matlab start from 1
 % Adding 1 at appropriate places to make the results consistent
 srch_pos_x = srch_ptrn_x_array(srch_ptrn_cnt+1, srch_pos_idx +1);
 srch_pos_y = srch_ptrn_y_array(srch_ptrn_cnt+1, srch_pos_idx +1);
  ref_blk_x_top_left = center_x + srch_pos_x + 1;
 ref_blk_x_top_left = center_x + srch_pos_x;
 ref_blk_mem_x_wrap_around = zeros(64,64);
 ref_blk_mem_xy_wrap_around = zeros(64,64);

 if (ref_blk_x_top_left < 0)
  ref_blk_mem_x_wrap_around(1:64, 1:abs(ref_blk_x_top_left)) =
        ref_blk_mem(1:64, 64 - abs(ref_blk_x_top_left)+1:64);
  ref_blk_mem_x_wrap_around(1:64, abs(ref_blk_x_top_left) +1 : 64) =
        ref_blk_mem(1:64, 1:64- abs(ref_blk_x_top_left));
  center_x_adj = center_x + abs(srch_pos_x);
 else
     ref_blk_mem_x_wrap_around = ref_blk_mem;
     center_x_adj = center_x;
 end

 ref_blk_y_top_left = center_y + srch_pos_y + 1;
 ref_blk_y_top_left = center_y + srch_pos_y;
 if (ref_blk_y_top_left < 0)
     ref_blk_mem_xy_wrap_around(1:abs(ref_blk_y_top_left), 1:64) =
       ref_blk_mem_x_wrap_around(64 - abs(ref_blk_y_top_left)+1:64, 1:64);
     ref_blk_mem_xy_wrap_around(abs(ref_blk_y_top_left) +1 : 64, 1:64) =
       ref_blk_mem_x_wrap_around(1:64- abs(ref_blk_y_top_left), 1:64);
     center_y_adj = center_y + abs(srch_pos_y);
 else
     ref_blk_mem_xy_wrap_around = ref_blk_mem_x_wrap_around;
     center_y_adj = center_y;
 end

 ref_blk_x_top_left = center_x_adj + srch_pos_x;
 ref_blk_y_top_left = center_y_adj + srch_pos_y;

 diff_blk(1:blk_sz_y, 1:blk_sz_x) =
   ref_blk_mem_xy_wrap_around(ref_blk_y_top_left+1 :
       ref_blk_y_top_left+blk_sz_y, ref_blk_x_top_left+1 :
       ref_blk_x_top_left+blk_sz_x) -
   org_blk_mem(1:blk_sz_y, 1:blk_sz_x);

 SAD = sum(sum(abs(diff_blk(1:blk_sz_y, 1:blk_sz_x))));

 OutputControlWord = 16384;
 OutputData(1) = (srch_ptrn_cnt *2.^8) + srch_pos_idx;
 OutputData(2) = SAD;

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Reset local variables and return if abndn_me
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if (abndn_me == 1)
    srch_pos_idx = 0;
end
```

Glossary

AsAP For *Asynchronous Array of simple Processors.* A parallel DSP processor consisting of a 2-dimensional mesh array of very simple CPUs clocked independently with each other.

CABAC For *Context Adaptive Binary Arithmetic Coding.* This is a process of loss-less entropy coding used in Main and Higher Profile of H.264 Video Compression Standard.

CAVLC For *Context Adaptive Variable Length Coding.* This is a process of loss-less entropy coding supported by all profiles of H.264 Video Compression Standard.

CIF For *Common Intermediate Format.* This is a format used to standardize the horizontal and vertical resolutions in pixels of YCbCr sequences in video signals. CIF resolution equals 352x288 pixels.

CMOS For *Complementary metal-oxide-semiconductor.*

DCT For *Discrete Cosine Transform.* DCT is used to transform a signal or image from the spatial domain to the frequency domain.

DSP For *digital signal processing or the processors for digital signal processing.*

H.264 A standard for video compression. It is also known as MPEG-4 part 10.

HDTV For *High Definition Television.*

MAD For *Mean Absolute Difference.*

MAE For *Mean Absolute Error.*

Micro-block is a region of a frame that is coded as a unit.

MPEG-4 MPEG-4 is a collection of methods defining compression of audio and visual (AV) digital data.

MSD For *Mean Square Difference*.

MV For *Motion Vector*.

NTSC For *National Television System Committee*.

PAL For *Phase Alternating Line*. A color-encoding system used in broadcast television systems in large parts of the world.

PSNR For *Peak Signal to Noise Ratio*. The ratio between the maximum possible power of a signal and the power of corrupting noise that affects the fidelity of its representation.

Partition In H.264, luminance micro-block of size 16x16 pixels can be split in to smaller micro-blocks of sizes: 16x8, 8x16, 8x8, 8x4, 4x8, or 4x4 pixels, defining 41 possible pixel regions. Each of these regions is called as a Partition.

QCIF For *quarter CIF*. QCIF resolution equals 176x144 pixels.

SAD For *Sum of Absolute Differences* between the pixels of the current micro-block and the pixels of the reference micro-block of the given size.

"sum of absolute differences" term For sum of the absolute differences between the pixels in a row of the current micro-block and the pixels in a row of the reference micro-block.

VLC *Variable Length Code*. In coding theory, a variable-length code is a code which maps source symbols to a variable number of bits.

Bibliography

[1] Dean N. Truong, Wayne H. Cheng, Tinoosh Mohsenin, Zhiyi Yu, Anthony T. Jacobson, Gouri Landge, Michael J. Meeuwsen, Anh T. Tran, Zhibin Xiao, Eric W. Work, Jeremy W. Webb, Paul V. Mejia, and Bevan M. Baas. A 167-processor computational platform in 65 nm cmos. *IEEE Journal of Solid-State Circuits (JSSC)*, 44(4):1130–1144, April 2009.

[2] Dean Truong, Wayne Cheng, Tinoosh Mohsenin, Zhiyi Yu, Toney Jacobson, Gouri Landge, Michael Meeuwsen, Christine Watnik, Paul Mejia, Anh Tran, Jeremy Webb, Eric Work, Zhibin Xiao, and Bevan M. Baas. A 167-processor 65 nm computational platform with per-processor dynamic supply voltage and dynamic clock frequency scaling. In *Symposium on VLSI Circuits, (VLSI '08)*, pages 22–23, Jun 2008.

[3] Z. Yu and B. M. Baas. A low-area multi-link interconnect architecture for GALS chip multiprocessors. *IEEE Transactions on Very Large Scale Integration Systems (TVLSIS)*. In press.

[4] Michael Meeuwsen, Zhiyi Yu, and Bevan M. Baas. A shared memory module for asynchronous arrays of processors. *EURASIP Journal on Embedded Systems*, 2007(86273):13, 2007.

[5] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, T. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, P. Mejia, A. Tran, J. Webb, E. Work, Z. Xiao, and B. Baas. A 167-processor computational array for highly-efficient DSP and embedded application processing. In *IEEE HotChips Symposium on High-Performance Chips (HotChips 2008)*, Aug 2008.

[6] Z. Yu and B. M. Baas. High performance, energy efficiency, and scalability with GALS chip multiprocessors. *IEEE Transactions on Very Large Scale Integration Systems (TVLSIS)*, 17(1):66–79, January 2009.

[7] Z. Xiao and B. M. Baas. A high-performance parallel CAVLC encoder on a fine-grained many-core system. In *IEEE International Conference of Computer Design (ICCD)*, pages 248–254, Oct 2008.

[8] Z. Yu and B. M. Baas. Low-area interconnect architecture for chip multiprocessors. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2857–2860, May 2008.

[9] Zhiyi Yu, Michael Meeuwsen, Ryan Apperson, Omar Sattari, Michael Lai, Jeremy Webb, Eric Work, Tinoosh Mohsenin, and Bevan Baas. Architecture and evaluation of an asynchronous array of simple processors. *Journal of VLSI Signal Processing Systems*, 53(3):243–259, Mar 2008.

[10] Zhiyi Yu, Michael Meeuwsen, Ryan Apperson, Omar Sattari, Michael Lai, Jeremy Webb, Eric Work, Dean Truong, Tinoosh Mohsenin, and Bevan Baas. AsAP: An asynchronous array of simple processors. *IEEE Journal of Solid-State Circuits (JSSC)*, 43(3):695–705, mar 2008.

[11] R. W. Apperson, Z. Yu, M. J. Meeuwsen, T. Mohsenin, and B. M. Baas. A scalable dual-clock FIFO for data transfers between arbitrary and haltable clock domains. *IEEE Transactions on Very Large Scale Integration Systems (TVLSIS)*, 15(10):1125–1134, Oct 2007.

[12] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, and B. Baas. An asynchronous array of simple processors for DSP applications. In *IEEE International Solid-State Circuits Conference, (ISSCC '06)*, volume 49, pages 428–429, Feb 2006.

[13] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Prentice Hall, 2007.

[14] A. Murat Tekalp. *Digital Video Processing*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[15] Alan C. Bovik. *Handbook of Image and Video Processing*. Academic Press, 2005.

[16] Iain Richardson and Iain E. G. Richardson. *H.264 and MPEG-4 Video Compression: Video Coding for Next Generation Multimedia*. Wiley, 2003.

[17] Rob Koenen. Iso/iec jtc1/sc29 wg11, coding of moving pictures and audio. Technical report, International Organization For Standardisation, 2002.

[18] Thomas Wiegand and Gary Sullivan. Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 — ISO/IEC 14496-10 AVC). Technical report, Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG, 2003.

[19] Syed Ali Khayam. The discrete cosine transform, theory and application. ECE 802-602 : Information Theory and Coding, Seminar 1, March 2003.

[20] Lai-Man Po and Wing-Chung Ma. A novel four-step search algorithm for fast block motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(3), June 1996.

[21] Shan Zhu and Kai-Kuang Ma. A new daimond search algorithm for fast block-matching motion estimation. *IEEE Transactions on Image Processing*, 9(2), February 2000.

[22] Y. He Z. Chen, P. Zhou and Y. Chen. Fast Integer Pel Motion Estimation for JVT, Doc. #JVT-F017. Technical report, ITU-T, Dec 2002.

[23] Sandro Moiron and Mohammed Ghanbari. Fast motion estimation for h.264/avc using dynamic search window. In *ConfTele2009*, number 67, May 2009.

[24] Muhammad Javed Mirza Gulistan Raja and Tian Song. H.264/avc deblocking filter based on motion activity in video sequences. *IEICE Electronics Express*, 5(19), 2008.

[25] S. Saponara and L. Fanucci. Data-adaptive motion estimation algorithm and vlsi architecture design for low-power video systems. In *Computers and Digital Techniques, IEE Proceedings*, volume 151, pages 51–59, January 2004.

[26] Pyen S.M., Min K.Y., Chong J.W., and S. Goto. An Efficient Hardware Architecture for Full-Search Variable Block Size Motion Estimation in H.264/AVC. In *ISVC06*, pages 554–563, 2006.

[27] Nuno Roma and Leonel Sousa. Efficient and configurable full-search block-matching processors. *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, VOL. 12, NO. 12, DECEMBER*, 12(12), Dec 2002.

[28] N. Roma S. Momcilovic, T. Dias and L. Sousa. Application specific instruction set processor for adaptive video motion estimation. In *9th EUROMICRO Conference on Digital System Design (DSD'06)*. IEEE, August 2006.

[29] N. Bellas and M. Dwyer. A programmable, high performance vector array unit used for real-time motion estimation. In *Multimedia and Expo, 2003. ICME '03. Proceedings. 2003 International Conference on*. IEEE, July 2003.

[30] Cao Wei, Hou Hui, Lai Jin Mei, Mao Zhi Gang, Tong Jia Rong, and Min Hao. A novel reconfigurable vlsi architecture for motion estimation. In *ASIC, 2007, ASICON '07, 7th International Conference on*, pages 774–777, Oct 2007.

[31] M. Ribeiro and L. Sousa. A run-time reconfigurable processor for video motion estimation. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 726–729, Aug 2007.

[32] A. Horng-Dar Lin, Anesko and B. Petryna. A 14 gops programmable motion estimator for h.26x video coding. In *Solid-State Circuits Conference, Digest of Technical Papers. 42nd ISSCC*, pages 246, 454, Feb 1996.

[33] Zhong L. He, Ming L. Liou, Philip. C. H. Chan, and C. Y. Tsui. Generic vlsi architecture for block-matching motion estimation algorithms. *International Journal of Imaging Systems and Technology, John Wiley & Sons, Inc.*, 9:257–273, 1998.

[34] Nikolaos Bellas and Malcolm Dwyer. Programmable motion estimation module with vector array unit. US Patent 6868123, May 2005.

[35] Mike Butts. Synchronization through communication in a massively parallel processor array. *IEEE Micro*, 2007.

[36] T. Koga, K. Iinuma, A. Hirano, Y. Iijima, and T. Ishiguro. Motion-compensated interframe coding for video conferencing. In *Proceedings NTC'81*, pages C9.6.1–9.6.5. IEEE, Nov 1981.

[37] Lai-Man Po and Wing-Chung Ma. A novel four-step search algorithm for fast block motion estimation. *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY*, 6(3), June 1996.

[38] Shan Zhu and Kai-Kuang Ma. A new diamond search algorithm for fast block-matching motion estimation. *IEEE TRANSACTIONS ON IMAGE PROCESSING*, 9(2), Feb 2000.