OPTIMIZATION OF INTEGER-PEL MOTION ESTIMATION FOR H.264 VIDEO ENCODING ON TMS 320C6416T DIGITAL SIGNAL PROCESSOR

MUHAMMET YAZICI

IŞIK UNIVERSITY
2007

OPTIMIZATION OF INTEGER-PEL MOTION ESTIMATION FOR H.264 VIDEO ENCODING ON TMS 320C6416T DIGITAL SIGNAL PROCESSOR

MUHAMMET YAZICI

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

IŞIK UNIVERSITY
2007

OPTIMIZATION OF INTEGER-PEL MOTION ESTIMATION FOR H.264 VIDEO ENCODING ON TMS 320C6416T DIGITAL SIGNAL PROCESSOR

APPROVED BY:

Assist. Prof. Hasan F. ATEŞ (Thesis Supervisor)	(Işık University)	____________________
Assoc. Prof. Ercan SOLAK	(Işık University)	____________________
Assist. Prof. Onur KAYA	(Işık University)	____________________

APPROVAL DATE: 3/12/2007

OPTIMIZATION OF INTEGER-PEL MOTION ESTIMATION FOR H.264 VIDEO ENCODING ON TMS 320C6416T DIGITAL SIGNAL PROCESSOR

Abstract

Video processing is used in many applications such as broadcast television and home entertainments. Video applications have been revolutionized by the advent of digital TV and DVD-video players. The standardization of video compression technology is essential for many video applications. Today the state-of-the-art compression standard is the H.264 standard. In this thesis, an H.264 encoder implementation is optimized on Texas Instruments TMS320C6416T board for real-time processing.

C6416 is a high performance and a low cost digital signal processor (DSP) chip that can achieve real time implementation of the algorithm. Thus, we choose C6416 because of based on our analysis of performance and cost.

In this thesis, hierarchal motion estimation module is implemented for the H.264 encoder. First of all algorithm code was written in C language. Then performance critical parts are written in assembly. The resulting code is an optimized implementation on the Texas Instruments TMS320C6416T DSP.

Simulations on TMS320C6416T reveal that the encoder processes 39-65 CIF frames per second, which satisfies 25 fps requirement for real-time applications.

H.264 KODLAYICISI İÇİN TAM SAYI PİXEL DEĞERLİ DEVİNİM KESTİRİMİNİN TMS 320C6416T İŞLEMCİSİ ÜZERİNDE OPTİMİZASYONU

Özet

Video işleme teknolojisi, televizyon yayıncılığı ve ev eğlenceleri gibi birçok uygulamada kullanılmaktadır. Video uygulamaları dijital televizyon ve DVD videolar sayesinde yaygınlaşmıştır. Standart video sıkıştırma teknolojisi bir çok video uygulamalarında zaruri hale gelmiştir. Bugün, teknolojide gelinen son nokta sıkıştırma standardı olan H.264'dür. Bu tezde, sıkıştırma algoritması olarak H.264 kullanılmış ve Texas Instruments'a ait olan TMS320C6416T Platformunda gerçek zamanlı olarak uygulanmıştır.

Performans ve maliyet analizlerinden dolayı, bu algoritmayı gerçek zamanlı uygulamalarda uygulaya bilmek için C6416 platformu seçildi.

Bu tezin algoritma mimarisinde, hiyerarşik hareket dengeleme algoritması H.264 kodlayıcısında uygulandı. Algoritmanın kodu ilk önce C dilinde daha sonra makine dilinde yazıldı. Kodun son hali optimize edilip TMS320C6416T DSP platformunun üzerinde uygulandı.

Sonuç olarak, TMS320C6416T DSP platformu üzerinde ölçümler gösteriyor ki, gerçek zamanlı uygulamalarda saniyedeki standart kare işleme hızı en az 25CIF kare iken yazılan kodun saniyede 39–65 CIF kare kadar işleyebildiği gözlemlenmiştir.

Acknowledgements

I would like to express my special gratitude to supervisor Assist. Prof. Hasan F. ATEŞ for help, support, interest and valuable contributions while completing this thesis. I am very glad and grateful to study with such a great supervisor. It was really a pleasure and honor for me.

I am also indebted to Ender Meriç from Vestek Electronics for sharing the H.264 Baseline Encoder code with me. I also want to thank to Mükerrem Aydoğan, who made corrections to my theses, for her morale support and encourage.

Besides all these people, I want to thank my brothers Fatih Yazıcı and Mustafa Yazıcı for their love and support. Also my dear friends Alper Şişman, Coşkun Tekeş, Erdem Kavurmacı, Nurcan Kabay and Kristin Surpuhi Benli, for their close friendship and encourage. I want to thank to Sevgi Dikmen, who answers all my questions sincerely.

Lastly, I would like to thank Işık University for supporting our project.

To my parents...

Hatice - Yüksel Yazıcı

Table of Contents

List of Figures

List of Tables

List of Abbreviations

1.	ASO	Arbitrary Slice Ordering
2.	CABAC	Context Adaptive Binary Arithmetic Coding
3.	CAVLC	Context-adaptive variable-length coding
4.	CIF	Common Intermediate Format
5.	DSP	Digital Signal Processor
6.	FMO	Flexible macroblock ordering
7.	FPGA	Field-Programmable Gate Array
8.	I slice	Intra- coded slice
9.	ISO/IEC	International Organisation for Standardisation / International Electrotechnical Commission
10.	ITU-T	International Telecommunication Union
11.	JVT	Joint Video Team
12.	MB	Macro Block
13.	MPEG	Moving Picture Experts Group
14.	MSB	Most Significant Bit
15.	MV	Motion Vector
16.	NAL	Network Abstrection Layer
17.	P slice	Predictive- coded slice
18.	PSNR	Peak Signal-To-Noise Ratio
19.	RS	Redundant Slices
20.	SAD	Sum of Absolute Differences
21.	SAE	Sum of Absolute Errors
22.	SIMD	Single Instruction, Multiple Data
23.	TI	Texas Instruments
24.	UVLC	Exp-Golomb Codes
25.	VCEG	Video Coding Experts Group
26.	VCL	Video Coding Layer

Chapter 1
Introduction

For several years, broadcast television and home entertainment sectors have definitely been the fastest worldwide growing markets which will increase considerably during the next years.

However, an increasing number of services and growing popularity of high definition TVs are creating greater needs for higher coding efficiency. H.264 is the most popular video compression and currently the most reliable standard, and was developed by a Joint Video Team (JVT) consisting of introduce from ITU-T's Video Coding Experts Group (VCEG) and ISO/IEC's Moving Picture Experts Group (MPEG) [1].

All the compression-decompression steps of H.264 are shown as a block diagram in Figure 1-1.

A video sequence is divided into pictures which are called frames. Following frames are usually similar, so contain a lot of redundancy parts. Removing this redundancy parts provide the better compression ratios. Motion estimation therefore aims to find a 'match' to the current block or region that minimizes the energy in the difference between the current block and the reference area. In intra mode, a prediction block predictive- coded slice is formed which is based on previously encoded and reconstructed blocks in the current slice. Mode decision compares the required amount of bits to encode a MB and the quality of the decoded MB for both of these modes (inter and intra), and chooses the mode with better quality and bit-rate performance. A deblocking filter is applied to reduce the effects of blocking artifacts in the reconstructed frame [3].

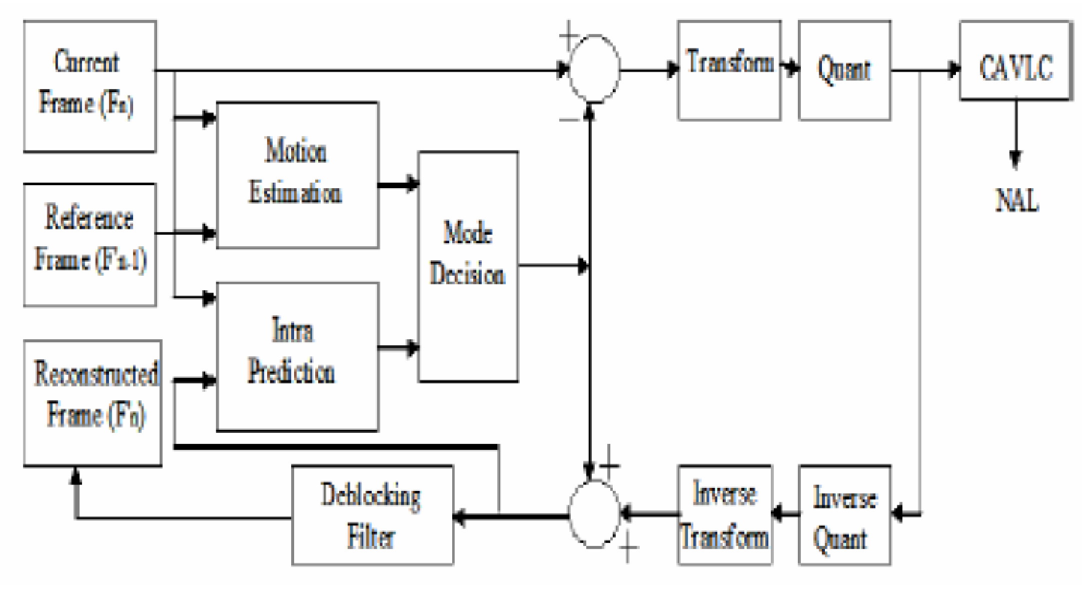


Figure 1.1 H.264 Encoder block diagram [3]

In this thesis, H.264 video encoding based on hierarchical motion estimation algorithm has been implemented and optimized for real-time implementation of H.264 / MPEG4 Part 10 video encoder. Hierarchical motion estimation algorithm speeds up the motion estimation process by using fewer number of search locations and computing the Sum of Absolute Differences (SAD) at a search location by using fewer number of pixels than full-search algorithm. The algorithm decrement the number of pixels used at a search location by down-sampling the current Macro Block (MB) and reduce the search area by performing the search operation in lower resolution. A 3-level hierarchical motion estimation algorithm is shown in Figure 1.2.

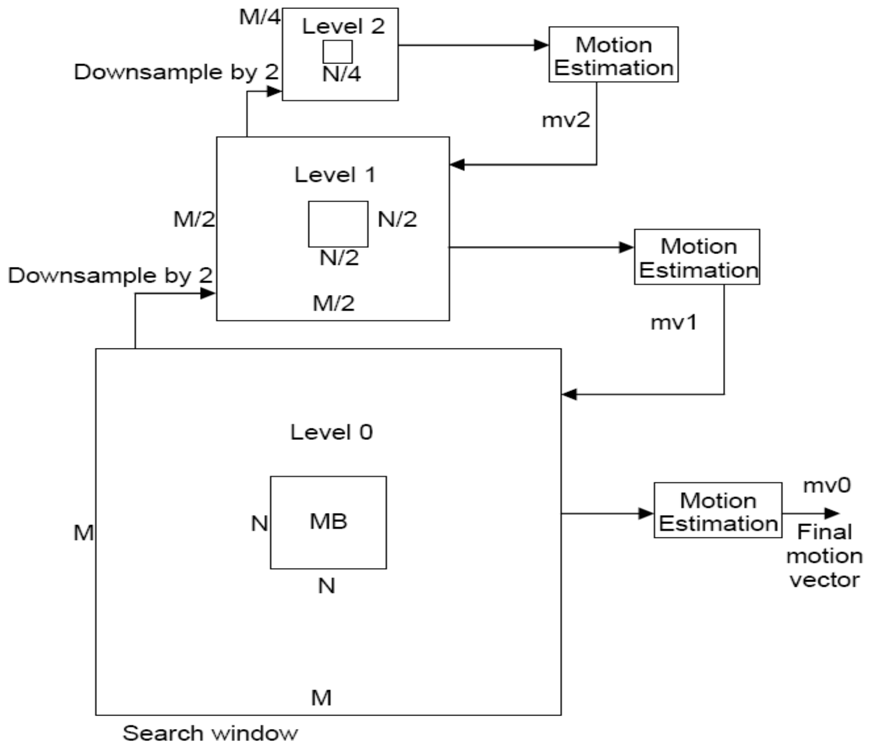


Figure 1.2 Hierarchical motion estimation algorithm [2]

The high coding efficiency of the H.264 standard increases algorithmic complexity; so for the encoders in many real-time applications a high performance core needs to be used. Today's processors, however, are very different from their ancestors. Older processors took several cycles to execute even simple operations like addition or storing data in memory. Performance was measured in thousands of instructions per second. Modern processors achieve high performance through enabling technologies such as parallel processing, deep pipelines, specialized internal compute engines, and integrated peripherals. Performance of these processors is often measured in millions or billions of operations per second.

TI TMS320C6416T DSP platform is reliable and has high performance. Therefore, real-time application of H.264 software encoder is implemented on TI TMS320C6416T DSP. It provides real-time performance, simulation and emulation, several multiply-accumulate operations per cycle, programming flexibility, reliability, increased system performance, and reduced system cost.

The proposed H.264 encoder implementation in this thesis, which is verified with the JM reference software [4], achieves real-time encoding for CIF resolution format on a TMS320C6416T core. Therefore, this implementation can be used in a real world implementation, especially in video conferencing and mobile applications. Also, the realization and optimization of the encoder on TI's TMS320C6416 DSP core is represented based of the target platform features. Moreover, the flexibility and programmability of DSP implementation enables easy adaptation for higher performance solutions as a future work.

Organization of the Thesis

Chapter 2 is an overview of H.264 algorithm and a description of it.

Chapter 3 explains motion estimation and mainly hierarchical motion estimation algorithm

Chapter 4 explains the C6416 digital signal processor core.

Chapter 5 explains the code development flow to increase performance.

Chapter 6 is the discussion part of the simulations results.

Finally, Chapter 7 presents the conclusions and the future work.

Chapter 2
H.264 Overview

H.264 was designed by the International Telecommunication Union (ITU-T) Video Coding Experts Group (VCEG) together with the International Organisation for Standardisation / International Electrotechnical Commission (ISO/IEC); Moving Picture Experts Group (MPEG) formed a Joint Video Team (JVT). JVT's_goal was to achieve a factor-of-2 reduction in bit rate compared to any competing standard. H.264 is the next-generation video compression technology in the MPEG-4 standard that is noted for achieving very high data compression. H.264 design's contains a Video Coding Layer (VCL), which manipulate the original frame and remove redundancy of the video picture content. Also, a Network Adaptation Layer (NAL) transmit or storage the representation of the video [2-4].

There are four profiles defined for H.264: Baseline, main, extended and high profiles. The baseline profile is perfectly designed for applications of video conferencing and allows to low delayed coding and decoding, which make video conferences be more natural.

The main and extended profiles are better suited for television applications and for applications of video streaming where latency is less critical.
The high profile is designed for disc storage applications, high definition television and digital broadcast.

APPLICATION

H.264 standard is designed for technical solutions that are used in the following application areas

• Broadcast cable, digital television on satellite, cable modem, video conferencing and streaming video over the Internet.

• Interactive or serial storage on optical and magnetic devices and high-definition television on DVD.

•Conversational services over Integrated Services Digital Network (ISDN), Ethernet, Local Area Network (LAN), Digital Subscriber Line (DSL), modems, wireless and mobile networks.

• Video-on-demand or multimedia streaming services over ISDN and MP3.

• Multimedia Messaging Services (MMS) over ISDN, DSL, Ethernet, LAN, wireless and mobile networks.

It seems as a fact that new important video applications will be developed by H.264 technology.

The common coding parts of H.264 profiles are listed as follows:

- Intra coded slice (I slice): Coded slice in which all macro blocks of the slice are coded using intra prediction.
- Predictive- coded slice (P slice): Macro blocks of the P slice can also be coded using inter prediction.
- Context-adaptive variable-length coding (CAVLC): Used for entropy coding.
- Deblocking filtering: Block-based video coding produces artifacts known as blocking artifacts which are deblocked by filtering.

Baseline decoders support the following features:

- Common parts : I slice, P slice, CAVLC
- Flexible macro block ordering (FMO) : A new ability to split the frame called the slice groups. A slice consists of one or more MBs and it is independently decodable.
- Arbitrary slice ordering (ASO): Each slice of a coded picture can be decoded independently without the other slices of the picture.
- Redundant slices (RS): An error/loss robustness feature allowing an encoder to send an extra representation of a picture region (typically at lower fidelity) that can be used if the primary representation is corrupted or lost (not supported in all profiles).

Coding parts for Main Profile

- Common parts : I slice, P slice, CAVLC
- Bi-directionally predictive-coded slice (B slice) is coded slice by using inter prediction from previously-decoded reference pictures, using at most two motion vectors and reference indices to predict the sample values of each blocks
- Weighted prediction is scaling operation by applying a weighting factor to the samples of motion-compensated prediction data in P or B slice
- Context-based Adaptive Binary Arithmetic Coding (CABAC) is used for entropy coding

Coding parts for Extended Profile

- Common parts : I slice, P slice, CAVLC
- SP slice is the specially coded slice for efficient switching between video streams. SP is similar to coding of a P slice
- SI slice is the switched slice that is similar to coding of an I slice
- Data partition is the coded data that is placed in separate data partitions. Each partition can be placed in different layer units.
- FMO
- ASO
- RS
- B slice

2.1 Video Coding Layer

Video stream is divided into pictures/ frames/ MBs for encoding. There are two modes as intra and inter. When the video stream is splinted, modes discard the redundancy part and subtract the original frame. Then the transform, quantization and entropy applied consecutively to the difference frame which is subtracted from the original frame sent to the NAL.

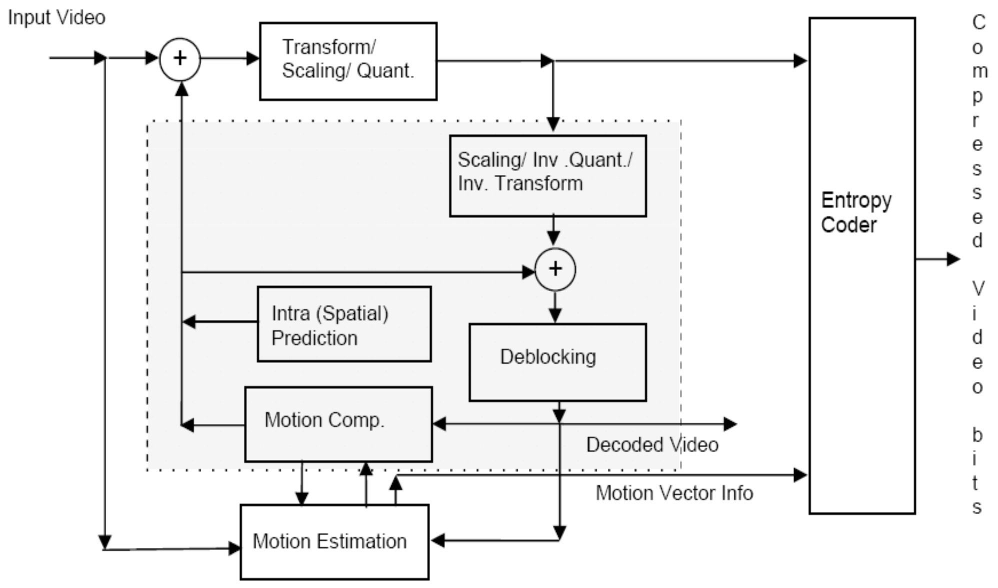


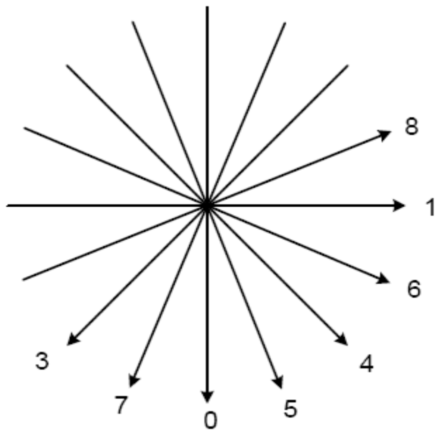
Figure 2.1 H.264 Encoder [5]

Figure 2.1 shows a block diagram of the video coding layer for a MB. The input video signal is split into MBs and then each MB has a process as shown.

2.2 Intra Mode

In intra mode, a prediction block P is formed which is based on previously encoded and reconstructed blocks in the current slice. There are three basic types of intra spatial predictions. First one is Full-MB prediction for 16x16 luma or the corresponding chroma block size, second one is 8x8 luma prediction and third one is 4x4 luma prediction.

Full-MB prediction can be performed in one way for intra spatial prediction and there are four ways that can be selected by the encoder for the prediction of each particular MB: vertical, horizontal, DC and planar. 8x8 luma blocks are similar to the 16x16 luma block prediction modes, but only the numbering of the modes is different. DC, horizontal, vertical and planar ways are applied consecutively. There are nine optional prediction modes for 4x4 luma block in this method. (Figure2.2) In the 4x4 spatial prediction mode, the values of each 4x4 block of luma samples are predicted from the neighboring pixels. The encoder typically selects the prediction mode for each block that minimizes the difference between P and the block to be encoded.



4x4 Luma block intra prediction modes

Mode 0: Vertical
Mode 1: Horizontal
Mode 2: DC
Mode 3: Diagonal Down-Left
Mode 4: Diagonal Down-Right
Mode 5: Vertical-Right
Mode 6: Horizontal-Down
Mode 7: Vertical-Left
Mode 8: Horizontal-Up

Figure 2.2 Spatial prediction of a 4x4 block [5]

2.3 Inter Mode

Inter prediction creates a prediction model from the reference pictures using block-based motion compensation and motion estimation. A H.264 standard provides flexibility in the motion compensation blocks size selection. The current picture can be partitioned into 16x16 MBs or the smaller blocks. The luminance component of each MB can be split in four ways as shown in Figure 2.3: 16x16, 16x8, 8x16 or 8x8. If the 8 x 8 mode is selected, each of the four MB partitions within the MB may be divided in a further four ways as shown in Figure 2.4: 8x8, 8x4, 4x8 or 4x4. This method of partitioning MBs into motion compensated sub-blocks of varying size is known as tree structured motion compensation [5].

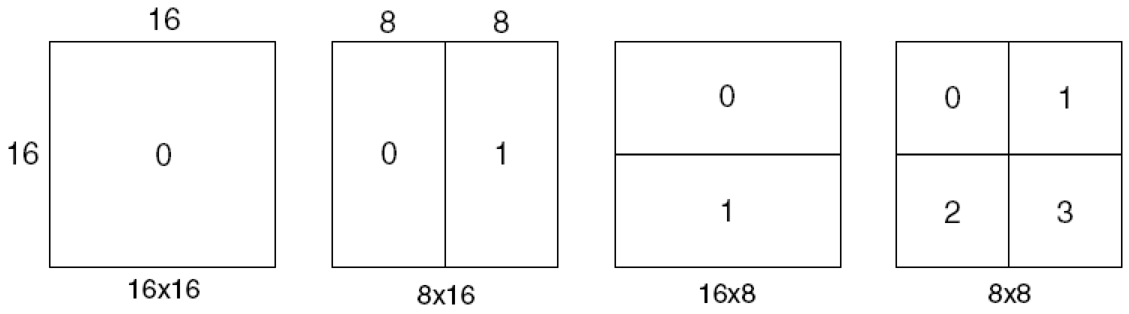


Figure 2.3 MB partitions: 16 x 16, 8 x 16, 16 x 8, and 8 x 8 [3]

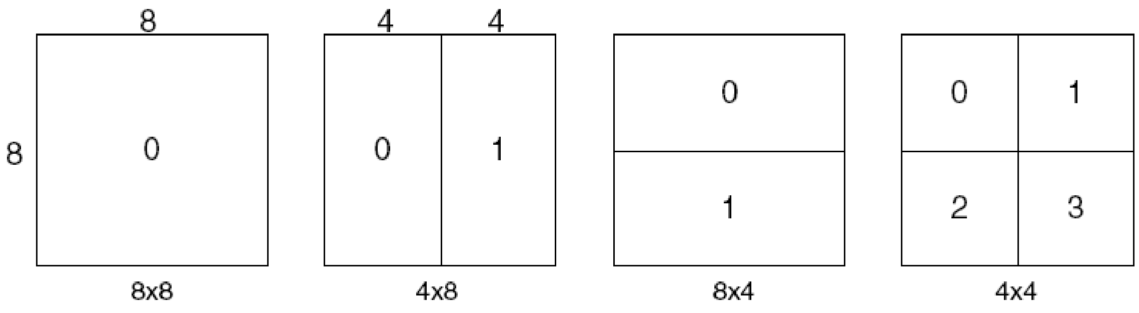


Figure 2.4 Sub-MB partitions: 8 x 8, 4 x 8, 8 x 4, and 4 x 4 [3]

2.4 Transform and Quantization

A critical parameter is the step size QP between successive re-scaled values. If the step size becomes larger, the range of quantized values that we get becomes smaller. Therefore, encoding can be efficiently represented during transmission. However, if the step size is smaller and quantized values range is larger, decoding could be done with a lost of efficiency.

There are three definitions for transforms:

1. Hadamard transform for the 4×4 array of luma DC coefficients in intra MBs predicted in 16×16 mode
2. Hadamard transform for the 2 × 2 array of chroma DC coefficients.
3. DCT-based transform for all other 4 × 4 blocks in the residual data.

2.5 Motion Vector Prediction

Encoding a motion vector for each partition can cost a significant number of bits, especially if small partition sizes are chosen. Motion vectors are usually correlated of neighboring blocks so each motion vector is predicted from previously coded. When a predicted motion vector is found, the difference between the current vector and the predicted vector is encoded and transmitted. At the decoder, the predictive vector consist the same method and added transmitted motion vector difference in order to find the current vector.

2.6 Entropy Coding

In H.264/AVC, two methods of entropy coding are supported: CAVLC which one is context adaptive variable length coding and CABAC that is context adaptive binary arithmetic coding. Syntax elements at and below the slice layer can be adaptively coded. As in previous standards, the MB is the fundamental unit of the syntax and decoding process. All syntax elements other than residual transform coefficients are encoded by the Exp-Golomb codes (UVLC) that scan order to read the residual data: zig-zag and alternate

2.7 H.264 PROFILES

H.264 standard includes the following four sets of capabilities and profiles.

Table 2.1 H.264 Baseline, Extended, Main and High profiles

	Baseline	**Extended**	**Main**	**High**
I and P Slices	Yes	Yes	Yes	Yes
B Slices	No	Yes	Yes	Yes
SI and SP Slices	No	Yes	No	No
In-Loop Deblocking Filter	Yes	Yes	Yes	Yes
CAVLC Entropy Coding	Yes	Yes	Yes	Yes
CABAC Entropy Coding	No	No	Yes	Yes
Flexible MB Ordering (FMO)	Yes	Yes	No	No
Arbitrary Slice Ordering (ASO)	Yes	Yes	No	No
Redundant Slices (RS)	Yes	Yes	No	No

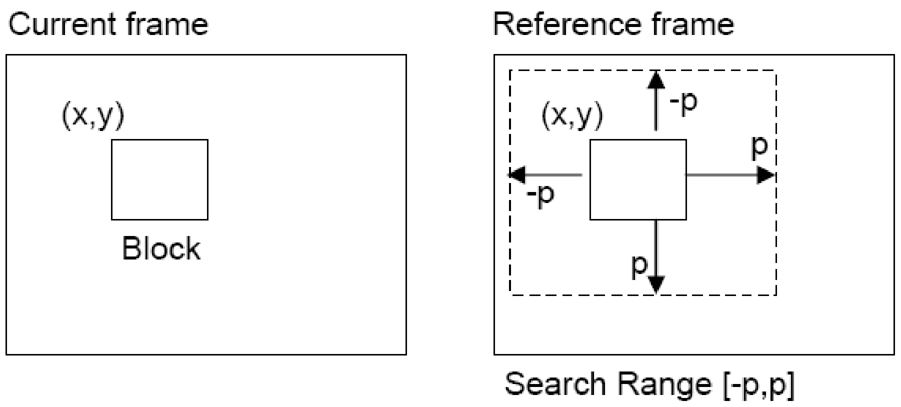
Chapter 3
Motion Estimation

In video compression, video sequence is divided into pictures which are called frames. Motion estimation guesses the next frame by observing the previous one. Following frames are usually similar, so containing a lot of redundancy part. Removing this redundancy part provide better compression ratios. Motion estimation therefore aims to find a 'match' to the current block or region that minimizes the energy in the difference between the current block and the reference area. The location of a block in a frame is given by using the (x,y) coordinates of top-left corner of the block in the current frame. The search window in the reference frame is the [-p,p] size region around the location of the current block in the current frame.

The SAD value for a current block in the current frame and a candidate block in the reference frame are calculated by accumulating the absolute differences of corresponding pixels in the two blocks as shown in the following formula:

$$SAD_{B_{mxn}}(d) = \sum_{x=1,y=1}^{m,n} | c(x,y) - r(x+d_x, y+d_y) |$$

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



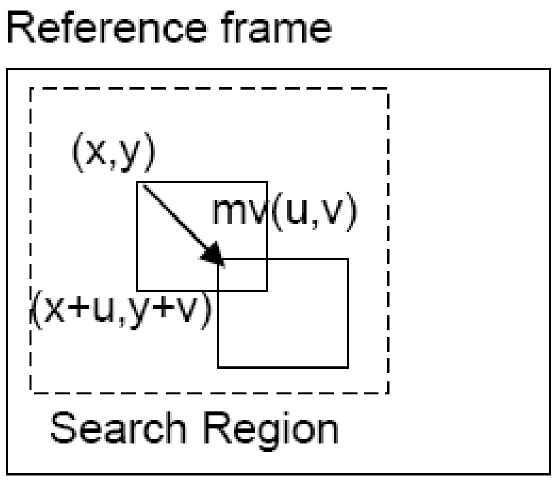


Figure 3.1 Motion estimation process [2]

3.1 Hierarchical Three Step Search

There are a lot of motion search algorithms that are used for video coding. Although the best one is full search algorithm which guarantees to find the motion vector with minimum SAD, it is not used in implementations. In implementations hierarchical three step search algorithm is a popular algorithm, which is showed in Figure 3.2.

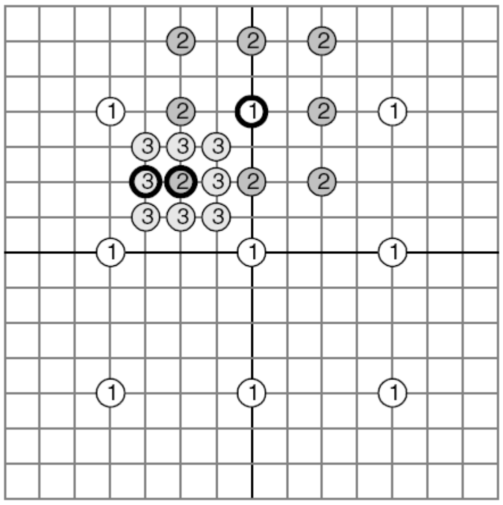


Figure 3.2 Three step search [3]

SAD is calculated at position (0,0) (the centre of the Figure 3.2) and eight points are located ±2*N*−1(S) unit far away from the center. In the figure, *S* is 7 and the first nine search locations are numbered as '1'. The search location that gives the smallest SAD is chosen as the new search centre and a further eight locations are searched. The distance between the center and the point becomes half of the previous distance from the search origin (numbered '2' in the Figure 3.2). After the best location had been chosen as the new search centre, the algorithm was repeated until the search distance cannot be subdivided further. The TSS is considerably simpler than Full Search. However the TSS does not usually perform as well as Full Search.

3.1.1 Hierarchical Motion Estimation Algorithm

Using fewer number of search locations and computing the SAD at a search location in which fewer number of pixels is used than it is used in full search algorithm, the hierarchical motion estimation algorithm accelerates the motion estimation process. As a result of down-sampling the current MB and using lower resolution in the search area, the algorithm reduces the number of pixels used for SAD calculation at a search location. A 3-level hierarchical motion estimation algorithm is shown in Figure 3.3.

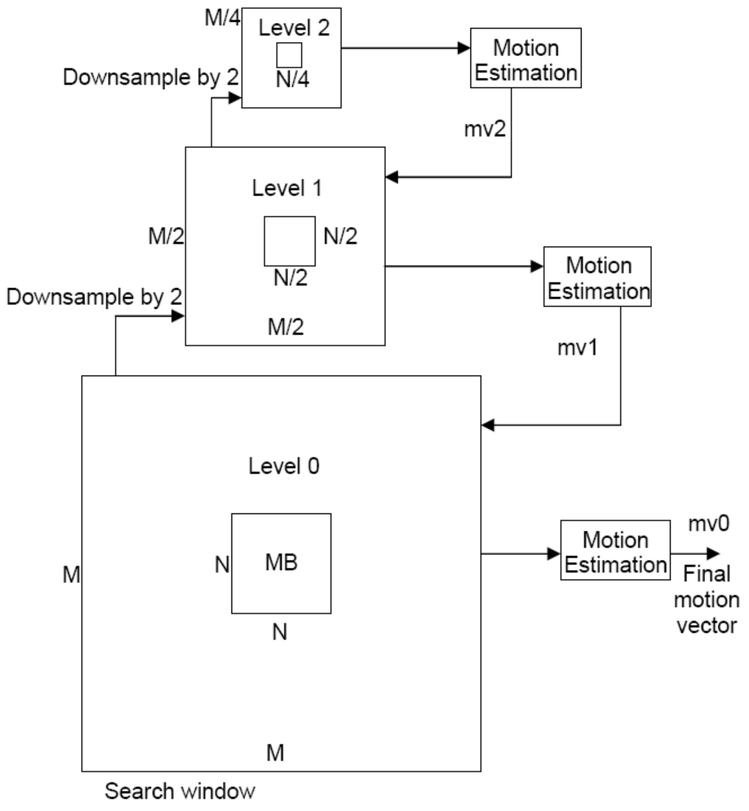


Figure 3.3 Hierarchical motion estimation algorithm [3]

The MB in level 0, which has N x N pixels, becomes N/2 x N/2 in level 1 by down-sampling the search area. Again by down sampling, the MB reaches to the level_2 with N/4 x N/4 pixels. This process is performed for all MBs of both the reference frame and the current frame.

While observing the location of MB, a proportional increase from level_2 to level_0 can be easily seen. For instance, when N is equal to 16 and if the location of the current 4x4 MB in level_2 is (x,y), the locations of the 8x8 current block in level_1 and the 16x16 current block in level_0 are (2x,2y) and (4x,4y) respectively.

The algorithm applies full search in level 2 for the 4x4 block with a search range as [-p2,+p2], and finds the motion vector as $mv_2(u_2,v_2)$. Then for 8x8 block the algorithm calculates the motion vector as $mv_1(u_1,v_1)$ by performing a limited search in level1 around the location pointed by $2mv_2$. The algorithm, finally, finds the motion vector $mv_0(u_0,v_0)$ by performing limited search in $level_0$ around the location pointed by $2mv_1$ for the 16x16 current MB with a search range p_0. The 3-level hierarchical motion estimation algorithm produces the optimal motion vector as mv_0.

In this thesis, with the help of SAD values, the vector of a 4x4MB at Level 2 is found by full search method. Also, the vector of a 8x8 MB at Level 1 is found by [-p2,+p2] search around 2 times the vector that is found at Level 2.

At Level 0, motion vectors for 8x8, 8x16, 8x16, and 16x16 subblocks of the MB are found by [-p1,+p1] refinement around 2 times the vector found in Level 1 for the full MB .

The further improve the motion vector estimation accuracy, another [-p1,+p1] refinement is carried out around [0,0] motion vector for each of the above subblocks. The minimum SAD results of these two different cases are compared and the motion vector which has the smallest SAD value is selected for the current subblock.

Chapter 4
Texas Instruments TMS 320C6416 DSP

The TMS320C6416T DSP platform is generally used in video performance audio and imaging applications. The C6416 DSK is a low-cost stand-alone development platform that enables users to evaluate and develop applications for the TI C6416 DSP family. The DSK also serves as a hardware reference design for the TMS320C6416T DSP. There are several features defined for C6416T DSK.

1. TMS320C6416T DSP operating at 1 Gigahertz.
2. An AIC23 stereo codec
3. 16 Mbytes of synchronous DRAM
4. 512 Kbytes of non-volatile Flash memory
5. 4 user accessible LEDs and DIP switches
6. Software board configuration through registers implemented in CPLD
7. Configured boot options and clock input selection
8. Standard expansion connectors for daughter card use
9. JTAG emulation through on-board JTAG emulator with USB host interface or external emulator
10. Single voltage power supply (+5V)

TMS320C6416 DSPs are industry's highest performance DSPs offer clock speeds up to 1 GHz and reduced system cost through peripheral integration. AIC23 codec allows the DSP to transmit and receive analog signals. The codec samples line inputs or analog signals on the microphone and converts them into digital data so it can be processed by the DSP. There are two busses 64-bit wide EMIFA and the 8-bit wide EMIFB. The SDRAM, Flash and CPLD are each connected to one of the busses. EMIFA is also connected to the daughter card expansion connectors which are used for third party add-in boards.

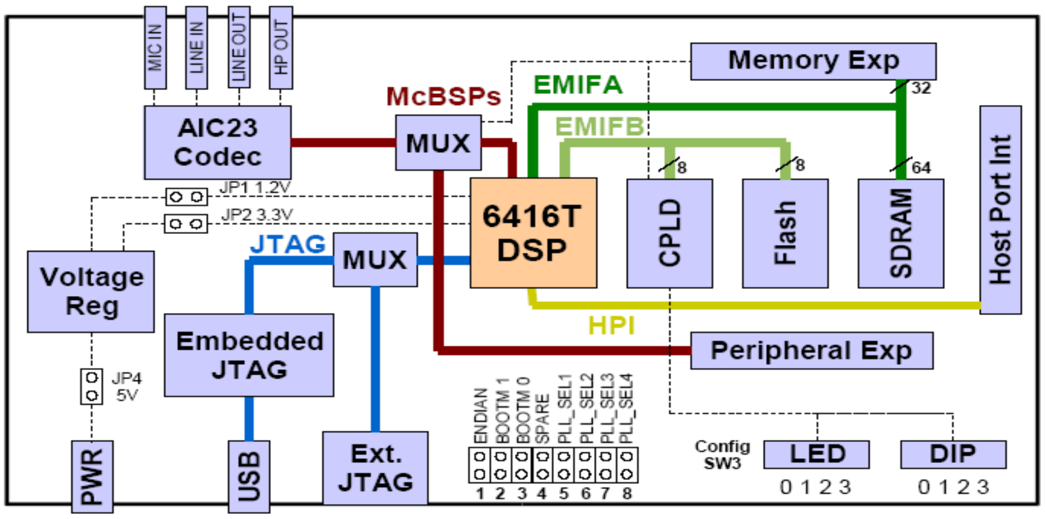


Figure 4.1 A block diagram of the profiles DSP board [9]

4.1 C6416 CPU

The C6000 DSP family with the VelociTI architecture addresses the needs of video and imaging applications. The C6000 family uses advanced very long instruction word (VLIW) architecture. The architecture contains multiple execution units running in parallel, which allow them to execute multiple instructions in a single clock cycle. Parallelism is the key word for high performance. The C6416 adds another significant performance boost to the C6000 DSP family. In addition to clock rate, C6416 introduced VelociTI.2 extensions to the VelociTI architecture. These extensions allow more work to be done in each cycle by including new instructions to accelerate performance in key application areas including video and imaging.

The C6416 CPU components consist of:

1. Two general-purpose register files (A and B)
2. Eight functional units (.L1, .L2, .S1, .S2, .M1, .M2, .D1, and .D2)
3. Two load-from-memory data paths (LD1 and LD2)
4. Two store-to-memory data paths (ST1 and ST2)
5. Two data address paths (DA1 and DA2)
6. Two register file data cross paths (1X and 2X)

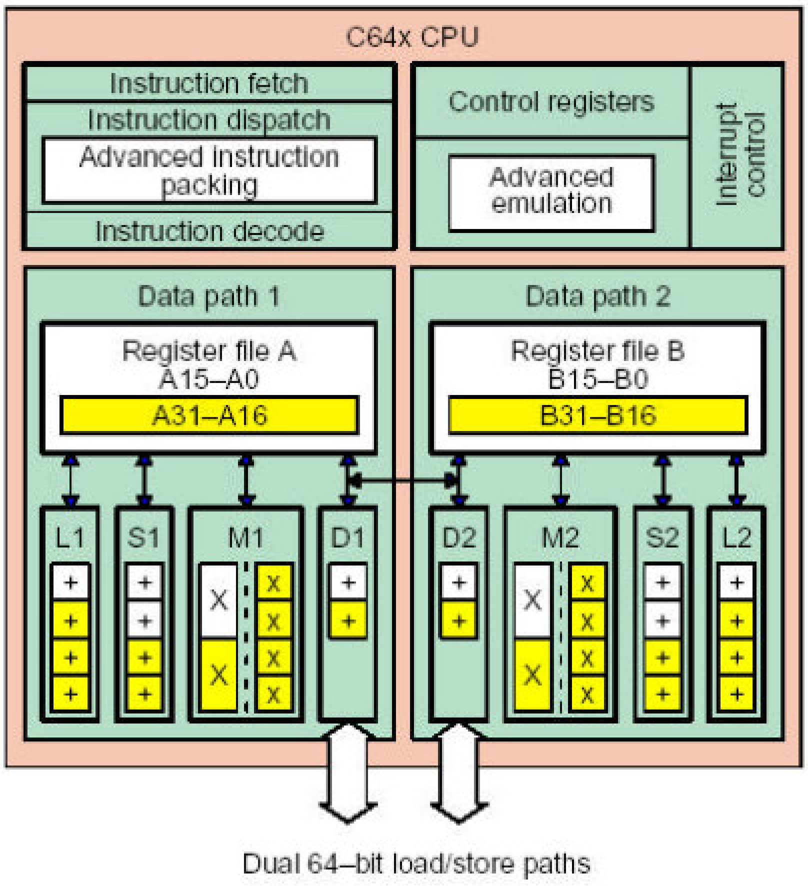


Figure 4.2 C6416 CPU Block diagram MPEG-4 real time encoding system [5]

4.2 Register Files

There are two general-purpose register files (A and B) in the C6000 data paths. For the C6416, each of these files contains 32 32-bit registers (A0–A31 for file A and B0–B31 for file B). The general-purpose registers can be used for data; data address pointers, or condition registers. On the C6416, registers A0, A1, A2, B0, B1, and B2 can be used as condition registers. In all C6000 devices, registers A4–A7 and B4–B7 can be used for circular addressing.

The C6416 register file supports data ranging in size from packed 8-bit data, packed 16-bit data, through 40-bit fixed-point, 64-bit fixed point, and 64-bit floating-point data. Values larger than 32 bits, such as 40-bit long and 64-bit float quantities are stored in register pairs, with the 32 LSBs of data placed in an even-numbered register and the remaining 8 or 32 MSBs in the next upper register (which is always an oddnumbered register). Packed data types store either four 8-bit values or two 16-bit values in a single 32-bit register or four 16-bit values in a 64-bit register pair.

4.3 Functional Units

The eight functional units in the C6000 data paths can be divided into two groups of four; each functional unit in one data path is almost identical to the corresponding unit in the other data path. The C6416 contains many 8-bit and 16-bit instructions to support video and imaging applications.

4.4 Register File Paths

Each functional unit reads directly from and writes directly to the register file within its own data path. That is, the .L1, .S1, .D1, and .M1 units write to register file A, and the .L2, .S2, .D2, and .M2 units write to register file B.
Most data lines in the CPU support 32-bit operands, and some support long (40-bit) and double word (64-bit) operands. Each functional unit has its own 32-bit write port into a general-purpose register file. Each functional unit has two 32-bit read ports for source operands *src1* and *src2*. Four units (.L1, .L2, .S1, and .S2) have an extra 8-bit-wide port for 40-bit long writes, as well as an 8-bit input for 40-bit long reads. Because each unit has its own 32-bit write port, all eight units can be used in parallel with every cycle when performing 32 bit operations. Since each C6416 multiplier can return up to a 64-bit result, an extra write port has been added from the multipliers to the register file. The register files are also connected to the opposite-side register file's functional units via the 1X and 2X cross paths have shown in Figure 4.3. These cross paths allow functional units from one data path to access a 32-bit operand from the opposite side's register file. The 1X cross path allows functional units from data path A to read its source from register file B. Similarly, the 2X cross path allows functional units from data path B to read its source from register file A.

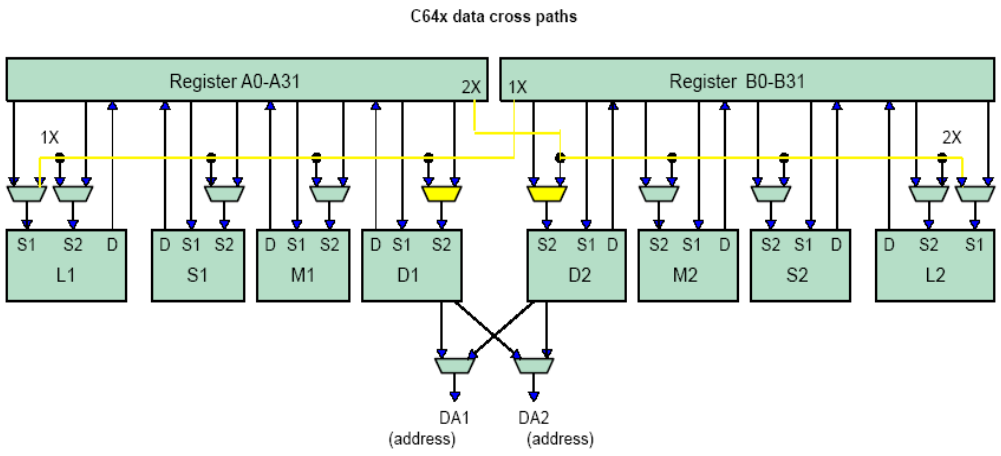


Figure 4.3 C6416 Data cross paths [7]

On the C6416, all eight of the functional units have access to the register file on the opposite side via a cross path. The .M1, .M2, .S1, .S2, .D1 and .D2 units' *src2* inputs are selectable between the cross path and the register file found on the same side. In the case of the .L1 and .L2, both *src1* and *src2* inputs are also selectable between the cross path and the same-side register file. Only two cross paths, 1X and 2X, exist in the C6000 architecture. Therefore, the limit is one source read from each data path's opposite register file per cycle, or a total of two cross-path source reads per cycle. The C64" pipelines data cross path accesses allow multiple units per side to read the same cross-path source simultaneously. The cross path operand for one side may be used by up to two functional units on that side in an execute packet.

4.5 Memory, Load and Store Paths

The data address paths named DA1 and DA2 are each connected to the .D units in both data paths. Load/store instructions can use an address register from one register file while loading to or storing from the other register file. Figure 4.6 illustrates the C6416 memory load and store paths.

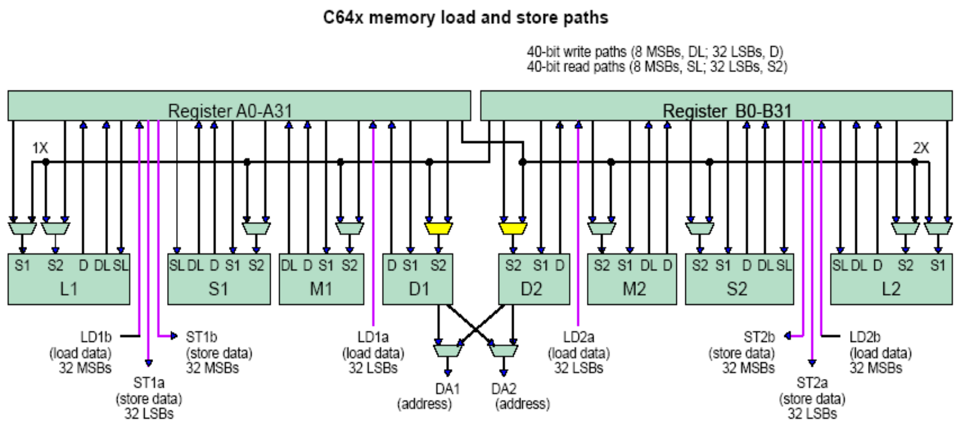


Figure 4.6 C6416 Memory load and store paths [9]

The C6416 device supports double-word loads and stores. There are four 32-bit paths for loading data for memory to the register file. For side A, LD1a is the load path for the 32 LSBs; LD1b is the load path for the 32 MSBs. For side B, LD2a is the load path for the 32 LSBs; LD2b is the load path for the 32 MSBs. There are also four 32-bit paths for storing register values to memory from each register file. ST1a is the write path for the 32 LSBs on side A; ST1b is the write path for the 32 MSBs for side A. For side B, ST2a is the write path for the 32 LSBs and ST2b is the write path for the 32 MSBs. Wide loads are essential in sustaining processing throughput. The C6416 device can also access words and double words at any byte boundary using non-aligned loads and stores. As a result, word and double-word data does not always need alignment to 32-bit or 64-bit boundaries. This feature is particularly useful in motion estimation and video filtering operations, where one may need access to data from any arbitrary byte boundary in memory. Non-aligned loads and stores combined with the pack and unpack instructions described earlier, mean that the compiler does not have to format the data to take advantage of the 8-bit and 16-bit hardware extensions. Without these operations, significant effort would be needed to leverage the parallelism. C6416 provides a complete set of data flow operations to sustain the maximum performance improvement made possible by the 8-bit and 16-bit extensions added to the C6000 architecture.

4.6 Additional Functional Unit Hardware

Additional hardware has been built into the eight functional units of the C6416. We have already discussed two important extensions. Each .M unit can perform two 16x16 bit multiplies or four 8x8 bit multiplies every clock cycle. Also, the .D units can access words and double words on any byte boundary by using non-aligned load and store instructions.

In addition, the .L units can perform byte shifts and the .M units can perform bi-directional variable shifts in addition to the .S unit's ability to do shifts. The .L units can perform quad 8-bit subtracts with absolute value. This absolute difference instruction greatly aids motion estimation algorithms.

Chapter 5
Hardware and Software Implementation

Code development is quite critical in this application; therefore code development flow is applied when writing and debugging the code. There are 3-steps to software development flows:
Develop C code, Refine C code, Write linear assembly.

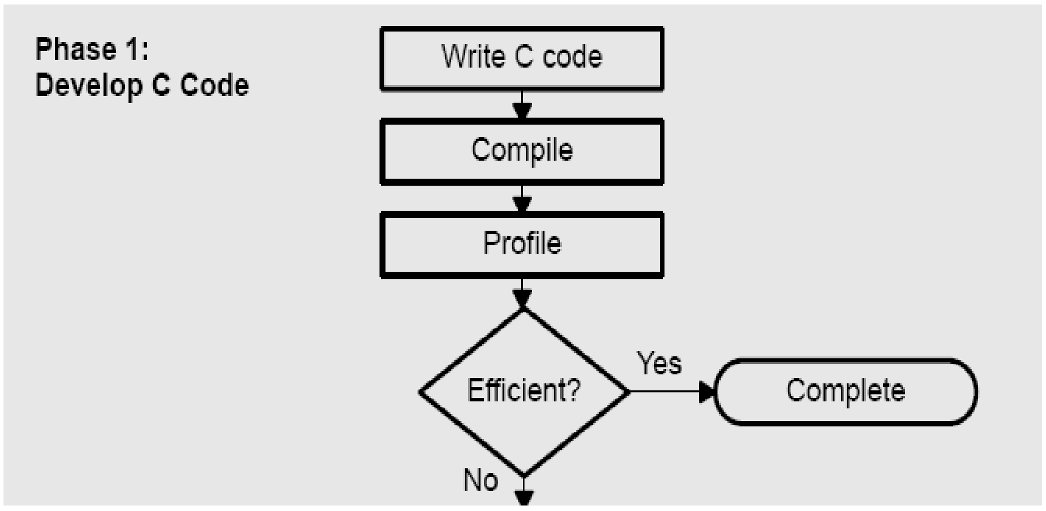


Figure 5.1 Develop C code flow chart [10]

Figure 5.1 shows the development of the C code for phase 1. It is written in the C code by using the C6416 profiling tools that are described in the Code Composer Studio User’s Guide. First of all, validation of the original C/C++ code is done and profiling is performed to determine which loops are most important in terms of millions of instructions per second requirements. Then the code is compiled and the performance is tested. Since it is not possible to improve the performance of this code any further, we should proceed to phase 2.

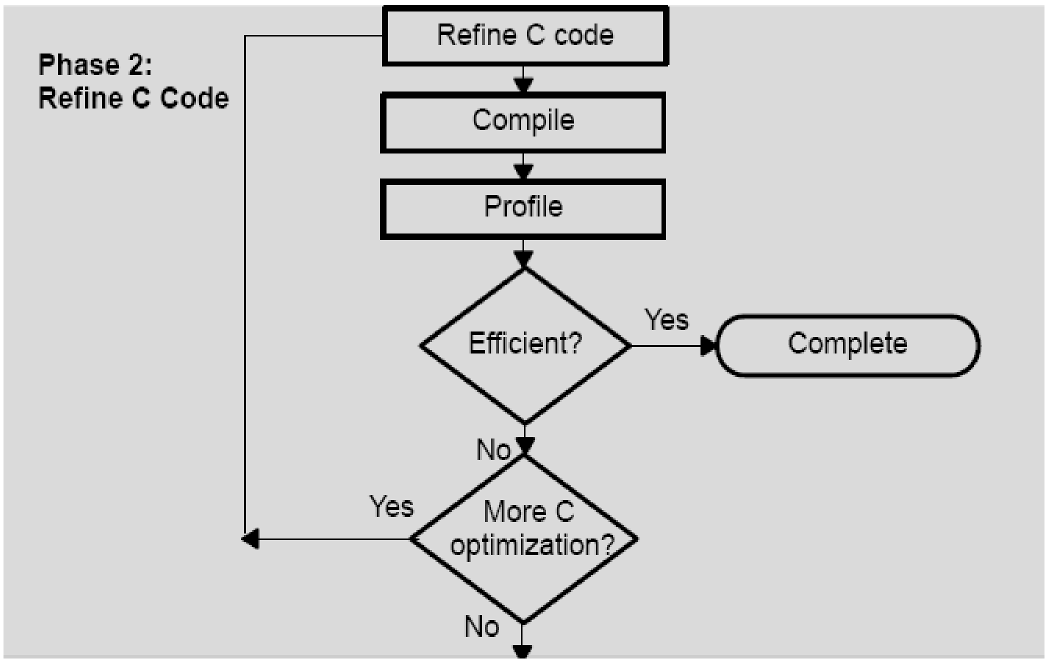


Figure 5.2 Refine C code flow chart [10]

Figure 5.2 shows development of rewritten C code for phase 2. First of all memory bank pragmas needs to be used to pass memory bank and alignment information to the compiler [10]. For the application of this process, the C6000 profiling tools are used to check its performance. If the code is still not as efficient as we would like to be, then it should proceed to phase 3.

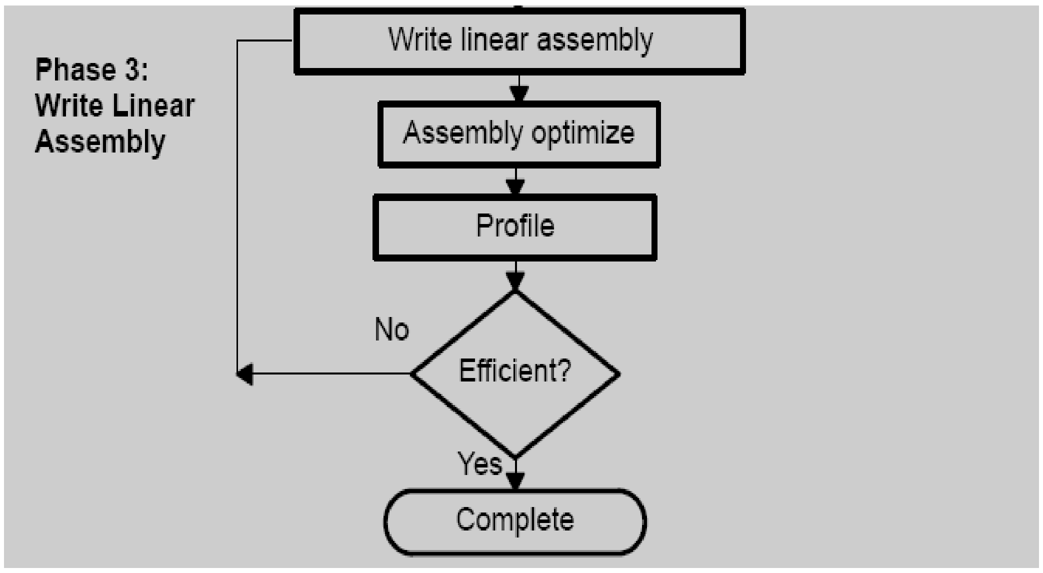


Figure 5.3 Linear assembly flow chart [10]

Figure 5.3 shows that C code is rewritten in linear assembly for phase 3. The linear assembly code allows us to determine exact C6416 instructions for best performance and it provides flexibility of hand-coded assembly without worry of pipelining, parallelism, or register allocation.

5.1 Compiling C/C++ Code

The C6416 compiler setting is very critical for performance in this video coding application.

5.2 Compiler Options

Options control the operations of the compiler which are for increasing the performance, optimizing the code, and decreasing code size.
In this thesis, optimization and speed are the most important issues. To get a start, the compiler optimization is set to Speed Most Critical (no -ms) for optimize for the code size and performance. Secondly the compiler is instructed by –o3, to perform file-level optimization. There are other ways to perform specific optimizations, although for general file-level this optimization level can be used alone. In addition to these, this compiler option can parallelize instructions, fill delay slots and maximize functional unit. Also, in compiler option the software pipelining can be turned on.

-pm option and –o3 option can be used together in the program level optimization as a part of Code Composer Studio. In addition to these there could be some necessaries to make some improvements in software pipelined loops. The compiler has the entire program; so it performs some additional optimizations which are usually applied during file-level optimization such as:

_ The compiler deletes the return code which is never used in the function.
_ The compiler removes the function which is not called directly or indirectly.

The –mt option eliminates memory dependencies, which allows the compiler to use assumptions that can eliminate memory dependency paths [10].

Table 5.1 C6416 Compiler options for performance

Options	**Description**
No debug	Exclude the debug info from the output file, so provides much more parallelized code
Speed Most Critical (no –ms)	The first strategy to determine the optimization type (code size vs. speed)
File(-o3)	Software pipelining, loop unrolling, SIMD. Different file level characteristics are increasing the performance.
Program Mode Compilation (-pm)	Program-level optimization option, which gives the combines source files to perform.
No Bad Alias Code (-mt)	This option, which allows the compiler to use assumptions that can eliminate memory dependency paths.

5.3 Software Pipelining

Software pipelining is a technique that is used to schedule instructions from a loop so that multiple iterations of the loop execute in parallel. When a compiler setting is chosen as the –o2 and –o3, the compilers apply to software pipeline code with information that it gathers from program.

5.4 Linear Assembly

An assembly language is a low-level language for programming computers. It implements a symbolic representation of the numeric machine codes and other constants needed to program a particular CPU architecture. The compiler sometimes does not fully exploit the potential of the C6000 architecture. Therefore, writing the loop/function in linear assembly obtains better performance.

As a result, linear assembly code will be the input for the assembly optimizer, to improve the performance of a video/image process on such DSP architecture. SIMD instructions can be calculated in linear assembly easily. The linear assembly is the key coding feature for using the pipeline parallelism with powerful Single Instruction, Multiple Data (SIMD) instructions as the inexpressible C program level operations.

Linear assembly is similar to regular C6416 assembly code in that we use C6416 instructions to write our code. With linear assembly we do not need to specify all of the information that we need to specify in regular C6416 assembly code. There is the option of specifying the information or letting the assembly optimizer specify it in linear assembly.

Here is the information that is not needed to specify in linear assembly code:

- Parallel instructions
- Pipeline latency
- Register usage
- Which functional unit is being used

If they are not specified, the assembly optimizer determines the information that is not included, based on the information that it has about the code. As with other code generation tools, linear assembly code might be needed to be modified up for a satisfactory performance. During linear assembly coding, much more details to assembly can be added, such as specifying which functional unit should be used. The important regulars of linear assembly code writing are:

A linear assembly file must be specified with a '**.**sa' extension.

'**.**cproc' and '**.**endproc' directives should be included by the code of linear assembly. A section of code that is optimized by assembly optimizer is bounded by the the .cproc and .endproc directives. .cproc directive has to be used at the beginning of the section and .endproc must be used at the end of the section.

A '.reg' directive which allows using descriptive names for values that will be stored in registers, may be included by the linear assembly code. The register agrees with the functional units chosen for the instructions. These instructions operate on the value. The register is chosen by the assembly optimizer when .reg directive is used.

A '.trip' directive may be included by linear assembly. The values which indicate how many times a loop will iterate, are specified by the .trip directive [9-10].

In the proposed encoder, for performance optimization, critical code segments and functions are written in linear assembly. Furthermore, with balanced side effect the SIMD instructions of the C6416 DSP are used for pipelining.

5.5 Linear Assembly of Critical Functions

Writing the linear assembly can provide the pipeline utilization and parallel video/image processing with SIMD instructions.
In H.264 encoder, the time consuming operations are written as linear assembly code, and specialized SIMD instructions are used for best performance.

The structure of the current and reference frames are used for motion estimation. In this thesis, CIF size frames are used. A CIF frame has 352x288 pixels. The motion estimation process is performed for each 16x16 MB in the current frame. For each 16x16 MB in the current frame, a 64x64 search window from the reference frame is used for motion estimation which means the MVs will be in the range [-24,24]. The current 16x16MB and the 64x64 search window are stored in block RAMs in the Field-Programmable Gate Array (FPGA).

In Table 5.2 there are the performance results of the SAD 4x4 function and Rec_192x160 is given. There are three different results for Phase1, Phase2 and Phase3.

In phase1, we write the C code by using the C6416 profiling tools that are described in the Code Composer Studio User’s Guide. The compliers settings are chosen as speed critical (-ms1) and optimize the level register (-o0).

In phase2, The C code which was prepared with “for loops” was rewritten by using the pointers and compilers settings are chosen as speed most critical (no-ms) and file level optimization (-o3). These options perform software pipelening, reduce potential pointer aliasing problems, allow loops with indeterminate iteration counts to execute epilogs; we also use memory bank and data alignment pragmas to pass memory bank and alignment information to the compiler.

In phase 3, the linear assembly code is used while rewriting the SAD 4x4, Rec_192x160, Rec_96x80, current MB 8x8 and 4x4. Also we used available linear assembly routines for SAD8x8 calculation.

For reference frame of size 352x288 (384x320 with padding for out-of-boundary motion vectors), the low-resolution frames of Level 1 and Level 2 are of size 192x160 and 96x80, respectively.

The REC_192x160 function is written in linear assembly and this function can be expressed as in Figure 5.4

```
LDNW        *org_ptrA++[ORG_WINA], win1
LDNW        *org_ptrB++[ORG_WINB], win2
AND         win1,org8,say1
AND         win1,org8_2,say2
SHR         say2,8,say2
ADD         say1,say2,say2
AND         win2,org8,say3
AND         win2,org8_2,say4
SHR         say4,8,say4
ADD         say3,say4,say4
ADD         say2,say4,say4
ADD         say4,2,say4
SHR         say4,2,say4
SUB         org_ptrA, 14, org_ptrC
SUB         org_ptrB, 14, org_ptrD
```

Figure 5.4 Linear assembly of REC192x160 function

For a video encoder in which the original MB is searched over a reference picture area, the motion estimation has the highest computational complexity. SAD based motion estimation scheme can be effectively implemented by a rich set of extensive video/image instructions. These instructions are provided by TMS320C6416 in the proposed encoder. The SAD 4x4 functions are written in linear assembly. These functions can be expressed as in Figure 5.5

```
LDNW        *win_ptr++[SRCH_WIN], win4
LDNDW       *win_ptr++[SRCH_WIN], win8:win4
LDNW        *win_ptr++[SRCH_WIN], win4_2
LDNW         *win_ptr++[SRCH_WIN], win8_2
SUBABS4      org4, win4, absdif4
SUBABS4      org8, win8, absdif8
DOTPU4       absdif4, dot_sad, SAD_tmp4
DOTPU4       absdif8, dot_sad, SAD_tmp8
ADD          SAD_tmp4, SAD1, SAD1
ADD          SAD_tmp8, SAD2, SAD2
```

Figure 5.5 Linear assembly of SAD4x4 function

The C6416 can access up to 64 bits per cycle at any byte boundary with non-aligned load instructions (LDNW, LDNDW).LDNW instruction loads four bits to memory. However, LDNDW loads eight bits.

Table 5.2 SAD 4x4 and REC 192x160 linear assembly results

Function	Phase1	Phase2	Phase3	Speed Ratio
SAD 4x4	194,5ms	124,3 ms	98,83	2x
REC 192x160	49,7 ms	25,6 ms	11,2 ms	4,45x

In this thesis, LDDW instruction (loads 4 bytes of memory) is used for loading and processing 8 pixels in parallel. Thus, our code automatically accelerates the data fetching from the MB especially in searching window in reference frame. On the other hand absolute value of pixels needs to be calculated. Also, eight bits needs to be loaded as pixel value. A SUBABS4 assembly instruction calculates the absolute value of the differences for each pair of packed 8–bit values. DOTPU4, an important video/image instruction, returns the dot product between four pairs of packed 8-bit values. For each pair of 8–bit values in number1 and number2, the 8–bit value from number1 is multiplied with the 8–bit value from number2. The four products are summed together. Since two DOPTPU4 can run in parallel in a single cycle, this instruction accelerates the sum of absolute difference (SAD) process significantly that is the core for motion estimation.

The main idea of SAD can be abstracted in the following steps:

1. LDDW and LDNDW fetch 8 pixels from current and reference frames
2. Two SUBABS4 calculate 8 absolute differences
3. Two DOTPU4 accumulate 8 result additions

```
        LDNW    .D1T1   *win_ptr++[SRCH_WIN],win4 ; |755|  win[7-4]:win[3-0]
|| [ loop_cnt] ADD .L2     0xffffffff,loop_cnt,loop_cnt ; |780|

        LDNW    .D1T1   *win_ptr++[SRCH_WIN],win8 ; |756|  win[7-4]:win[3-0]
        LDNW    .D1T1   *win_ptr++[SRCH_WIN],win4_2 ; |757|  win[7-4]:win[3-0]

        LDDW    .D1T1   *org_ptrA++[ORG_WINA],org8:org4 ; |747|  org[7-4]:org[3-0]
||      LDDW    .D2T2   *org_ptrB++[ORG_WINB],org8_2:org4_2 ; |748|  org[7-4]:org[3-0]

        LDNW    .D1T1   *win_ptr++[SRCH_WIN],win8_2 ; |758|  win[7-4]:win[3-0]
        NOP           3
        SUBABS4 .L1     org4,win4,absdif4 ; |762|

        SUBABS4 .L1     org8,win8,absdif8 ; |763|
||      SUBABS4 .L2X    org8_2,win8_2,absdif8_2 ; |772|

        DOTPU4  .M1X    absdif8,dot_sad,SAD_tmp8 ; |766|
||      DOTPU4  .M2     absdif8_2,dot_sad,SAD_tmp8_2 ; |775|
||      SUBABS4 .L2X    org4_2,win4_2,absdif4_2 ; |771|

   [ loop_cnt] B   .S1     L6              ; |781|
||      DOTPU4  .M2X    absdif4,dot_sad,SAD_tmp4 ; |765|

        DOTPU4  .M2     absdif4_2,dot_sad,SAD_tmp4_2 ; |774|
   [!loop_cnt] RET .S2     B3              ; |786|
        ADD     .L1     SAD_tmp8,SAD2,SAD2' ; |769|

        ADD     .L1X    SAD_tmp8_2,SAD2',SAD2 ; |778|
||      ADD     .L2     SAD_tmp4,SAD1,SAD1' ; |768|

        ADD     .L2     SAD_tmp4_2,SAD1',SAD1 ; |777|
        ; BRANCHCC OCCURS {L6}          ; |781|
```

Figure 5.6 Disassembly of SAD linear assembly's core loop

In this thesis we have use the basic 264 encoder that is being Vestek electronics [8]. This software has some functions in terms of the real-time implementation of the H.264/AVC encoder on DM642 DSP core. The standard H.264/AVC baseline profile coding tools is used except error resiliency tools and quarter-pel motion estimation in Vestek Electronics code. Integer and half pixel position motion estimation and compensation for all luminance and chrominance components are implemented instead of quarter-pel motion compensation.

The modifications described above are incorporated into the baseline H.264 encoder of VESTEK Electronics. To summarize, we have implemented the following steps during encoder optimization:

1- For a better comparison with our work, the code that was taken from Vestek Electronics was run on the board TMS320C6416T. The results are given in table 6.1 to compare with the new algorithm results.

2- Vestek Electronics used bihexagonal search algorithm in the step of motion estimation. According to the SAD reuse in hierarchical motion estimation in Hasan Fehmi Ateş's article [7], the motion estimation module of H.264 encoder was re-written. Hierarchical method speeds up motion estimation so it was the preferred method for our implementation.

3- After a stand-alone analysis of the code, Hierarchical motion estimation algorithm was incorporated into the available H.624 encoder software.

4- First of all, the reference frame was saved as a whole in external memory. The whole frame was reduced to low resolution. The low-resolution frames are stored in internal memory using the pragma directive. The current 16x16 MB that was located in internal memory was reduced first into 8x8 then 4x4 blocks. This is a pre-processing step to Hierarchal motion estimation module.

5- The motion vectors that are found in the Hierarchal step and the SAD values were calculated. After these we accepted the motion vectors as zero and again the SAD values were calculated. The results of these two different cases compared and the case which had the smallest SAD value showed us the motion vector.

6- Simulations have shown that the initial unoptimized version of the algorithm was slower than the VESTEK implementation. So, we decided to use the three phased system described above. Architectural details of the board were investigated. Useful settings of the optimizing compiler were chosen. Some reductions in execution times were seen based on these new settings.

7- According to Phase 2, the C written for “code loops” was restructured by using array pointers for more regular memory access. This change reduced data cache misses and therefore reduced the total execution times.

8- To get better results, the code was written in assembly language which is shown in Phase 3. First of all, the functions that produce low-resolution reference frames for Level 1 and Level 2 were written in assembly language. In addition to these, the function that reduces the current MB into low resolution and the functions that were used for calculating SADs during motion estimation were transformed into linear assembly.

9- At the end of these steps, the program code was placed in internal memory, to avoid any possible memory stalls due to program cache misses and long delays of accessing external memory.

10- After these applications, the code was run again and the results are shown in the next section.

Chapter 6

Simulation Results

In this thesis, C6416 with higher performance and lower cost hardware architecture is chosen in order to achieve H.264 encoder in real time implementation. There are three basic properties of TMS320C6416 board in terms of the processor rate is 1 GHz, the flash memory is 512 Kb and the ability of making 32-Bit Instructions/Cycle. The C6416T uses a two-level cache-based architecture and has a powerful and diverse set of peripherals. The Level 1 program cache (L1P) is a 128-Kbit direct mapped cache and the Level 1 data cache (L1D) is a 128-Kbit 2-way set-associative cache. The Level 2 memory/cache (L2) consists of a 1-Mbit memory space that is shared between program and data space. L2 memory can be configured as mapped memory or combinations of cache (up to 256K bytes) and mapped memory [11].

Overall, encode MB process of the H.264 baseline encoder that was obtained from Vestek electronics. It has been modified for real time implementation and optimization.

The data in the following tables are obtained by using CIF Foreman sequence.

Vestek electronics code which is without any modification run on TMS320C6416T board. The result is shown in table 6.1. Modified code is also run under the same circumstances and the corresponding values are given in table 6.2.

Table 6.1 Vestek's code optimization results for CIF [352x288] 'Foreman' sequence

Functional Block		Total Count @ 30 frames	Total Elapsed Time (ms)	Average Time (ms/frame)	Percentage (%)
Before & after encode (EDMA usage)		All MBs	143,2	4,77	14,68
Encode MB		All MBs	598,56	19,95	61,38
Encode MB	Motion Estimation	29 frames	373,65	12,88	62,42
	Residual Coding	All MBs	189,2	6,31	31,61
	Intra Predict & Others	All MBs	35,71	1,19	5,97
Deblocking Filter		30 frames	116,87	3,90	11,98
VLC (write NALU)		All MBs	50,47	1,68	5,18
Half-pel Interpolation		29 frames	66,09	2,28	6,78
Total		30 frames	975,19	32,51	100,00

Comparing Table 6.1 and Table 6.2, we see a speed-up in various parts of the encoding process. The reason of the speed increase in the after and before encode is the transform of 128 Mbyte cache memory into 256 Mbyte. According to tables, in the step of encode MB motion estimation, the execution time of 374 ms is decreased to 270 ms which is equal to %28 speed increase. This improvement is due to the software modifications that are explained in Chapter 5. However, if we run the Vestek code with 256 Mbyte cache memory, the motion estimation part becomes 355 ms. These brings the speed up of the motion estimation part as 24%.

The speed improvement for the overall encoding time of Foreman sequence is also 21%. Due to the algorithmic, memory and code optimizations performed in the encoder software, the resulting encoder can code Foreman at 40 frames per second (fps), compared to 30 fps of Vestek code.

Table 6.2 Optimization results for CIF [352x288] 'Foreman' sequence

Functional Block		Total Count @ 30 frames	Total Elapsed Time (ms)	Average Time (ms/frame)	Percentage (%)
Before & after encode (EDMA usage)		All MBs	89,35	2,98	11,57
Low Resulation Referance Frame		All MBs	11,37	0,38	1,47
Encode MB		All MBs	452,87	15,10	58,64
Encode MB	Motion Estimation	29 frames	270,75	9,03	59,79
	Residual Coding	All MBs	154,56	5,15	34,13
	Intra Predict & Others	All MBs	27,56	0,92	6,09
Deblocking Filter		30 frames	114,37	3,81	14,81
VLC (write NALU)		All MBs	42,2	1,41	5,46
Half-pel Interpolation		29 frames	62,09	2,07	8,04
Total		30 frames	772,25	25,74	100,00

Table 6.3 Vestek's code compression efficiency of the implemented H.264 encoder

Sequence CIF [352x288]	Encoder Speed (fps)	Y-PSNR (dB)	U-PSNR (dB)	V-PSNR (dB)	Compression Ratio
Akiyo	49,72	39,5	42,49	44,03	258
Paris	32,08	33,49	34,93	34,58	48
Foreman	30,07	36,03	40,68	43,24	67
Container	37,11	35,95	42,8	41,91	105
Mother and Daughter	47,9	38,87	43,66	44,5	281
News	48,3	37,68	40,03	41,629	128

Accelerated data processing speed would normally have a negative effect on PSNR and compression ratio; however the hierarchical algorithm we used has resulted positively. PSNR values on tables 6.3 and 6.4 indicate the acceleration without a loss of visual quality, while there is a slight increase in bit-rate Simulation results for different sequences are available at table 6.4.

In the Table 6.4, it can be seen that the same encoder used for different sequences result in different frame rates because some sequences include more action and details. Foreman and Container sequence which are used in video conferences and mobile applications, have significant roles in all this application. After observing values of these sequences, we can say that the system processes 40 frames in 1 sec. For a real-time application frame rate can be 25 to 30 fps. Therefore, The H.264 encoder is optimized enough for real-time purposes and the achieved encoding rate is over 25 fps.

Table 6.4 Compression efficiency of the implemented H.264 encoder

Sequence CIF [352x288]	Encoder Speed (fps)	Y-PSNR (dB)	U-PSNR (dB)	V-PSNR (dB)	Compression Ratio
Akiyo	63,38	39,5	42,49	44,03	258
Foreman	38,85	36,03	40,68	43,24	65
Container	46,8	35,95	42,8	41,91	106
Mother and Daughter	61,3	38,87	43,66	44,5	284
News	53,59	37,68	40,03	41,629	129

Chapter 7

Conclusion And Future Work

In this thesis, high performance and low cost hardware architecture is designed for real-time implementation of an SAD reuse based hierarchical motion estimation algorithm for H.264 / MPEG4 Part 10 video coding. The reference decoder verifies and implements a real-time H.264 baseline encoder on TI TMS320C6416 digital signal processor (DSP) at Common Intermediate Format (CIF) (352x288) resolution. The performance measurements over video sequences show that 40 to 65 fps encoding performance is possible and the PSNR measurements are sufficient for embedded applications such as video conferencing and mobile applications.

For future work, other parts of the encoder can be written in linear assembly for better optimization. Especially getting rid of all dynamic memory allocations and defining all variables as static can speed up the encoder. Placing most of the variables, such as the reference and current frames as a whole in the internal memory will avoid memory stalls of the CPU due to slow external memory access. However, this is not possible unless there is enough empty space in internal memorys the proposed encoder achieves real-time performance at CIF resolution, improvement of the encoder performance for higher resolution especially at D1 (720x576) (Standard TV resolution) is a challenging future study.

References

[1] Wiegand, T., Sullivan, G. J., Bjontegaard, G. and Luthra, A., "Overview of the H.264/AVC Video Coding Standard", *IEEE Transactions on Circuits and Systems for Video Technology*, 13, 7, 560-576, July 2003.

[2] Mehmet, G.. *H.264 Baseline Encoder Implementation and Optimization on TMS320DM642 Digital Signal Processor*, M: S. Thesis, Sabancı University, İstanbul, 2005.

[3] Richardson, I. E. G., H.*264 and MPEG-4 Video Compression*, The Robert Gorden University, Aberdeen, ENGLAND, 2003.

[4] Laleh, S., *Context-Based Complexity Reduction Applied to H.264 Video Compression*, M. S. Thesis, Simon Fraser University of Technology, CANADA, 2005.

[5] Wiegand, T., Sullivan, G. J., Bjontegaard, G. and Luthra, A., "Overview of the H.264/AVC Video Coding Standard", *IEEE Trans. on Circuits and Systems for Video Technology,* 13, 7, 560-576, July 2003.

[6] Sullivan, G. J., Pankaj, T. and Ajay, L.," The H.264/AVC Advanced Video Coding Standard:Overview and Introduction to the Fidelity Range Extensions", *Applications of Digital Image Processing XXVII. Edited by Tescher, Andrew G. Proceedings of the SPIE, 5558454-474*, August, 2004.

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

[8] Ender, M., *Implementation and Optimization of Real-time H.264Baseline Encoder on TMS320DM642 DSP*, M.S. Thesis, İstanbul Technical University İstanbul, June 2007.

[9] Kwon, S., Tamhankar A. and Rao, K. R., *Overview of H.264/MPEG-4 Part 10,* Dongeui University, T-Mobile, University of Texas at Arlington, USA, March 15.

[10] Schäfer, R., Wiegand, T. and Schwarz, H., *The Emerging H.264/AVC, Technical Review, Heinrich Hertz Institute,* Berlin, Germany, January 2003.

[11] Digital Spectrum, Inc., *TMS320C6416T Evaluation Module*, Technical Reference. Texas Instruments, 2004.

[12] *TMS320C6000 Programmer's Guide Document*, Texas Instruments, 2000.

[13] *TMS320C6000 CPU and Instruction Set*, Reference Guide, Texas Instruments, 2006.

[14] *TMS320C64x DSP Library, Programmer's Reference*, Texas Instruments, 2002

[15] http://focus.ti.com/docs/prod/folders/print/tms320c6416t.html.