# H.264 BASELINE VIDEO ENCODER IMPLEMENTATION AND OPTIMIZATION ON TMS320DM642 DIGITAL SIGNAL PROCESSOR

by

MEHMET GÜNEY

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

Sabancı University
Fall 2006

H.264 BASELINE VIDEO ENCODER IMPLEMENTATION AND OPTIMIZATION ON TMS320DM642 DIGITAL SIGNAL PROCESSOR

APPROVED BY:

Assist. Prof. Dr. Ayhan Bozkurt …….

(Thesis Supervisor)

Assist. Prof. Dr. İlker Hamzaoğlu …….

Assist. Prof. Dr. Hasan Ateş …….

Assist. Prof. Dr. Ahmet Onat …….

Assist. Prof. Dr. Mehmet Keskinöz …….

DATE OF APPROVAL: …….

## ABSTRACT

Digital video encoding plays an important role in many applications such as digital surveillance systems, video conference systems as well as digital TV. In this thesis, a H.264 baseline encoder is implemented on Texas Instruments TMS320DM642 digital signal processor.

The TMS320DM642 is a high-performance digital media processor with 2-level memory/cache hierarchy and very-long-instruction-word (VLIW) architecture. The proposed encoder system consists of almost all parts of standard H.264 baseline encoder except quarter-pel motion compensation and error resiliency tools such as Arbitrary Slice Ordering (ASO) and Flexible Macroblok Order (FMO). Instead of quarter-pel motion compensation, integer-pel motion estimation and compensation for both Luminance and Chrominance samples is implemented.

The complete H.264 encoder system is verified to work on both computer and DM642 EVM (Evaluation Module) platform. Basically, the encoder takes the input of a QCIF video sequence (YUV) and converts it to the standard compressed H.264 AnnexB file format. The encoder is fully compliant with the standard H.264 JM Decoder.

The reconstructed video, which is exactly the same with the output of the standard JM H.264 decoder, is being displayed on a TV screen. In addition, by making use of the TI development tools, performance of the complete encoder system is analyzed for real-time applications.

Finally, memory optimization, code optimizations and compiler optimizations are applied to the encoder for higher performance. The proposed H.264 encoder is able to encode, display and store 26.7 QCIF frames per second.

# ÖZET

Sayısal video kodlama sayısal gözetim, video konferans sistemleri ve sayısal televizyon gibi birçok uygulamalarda önemli rol oynar. Bu tezde, H.264 taban profili video kodlayıcı Texas Intruments TMS320DM642 Sayısal Sinyal İşleyici üzerinde gerçeklenmiştir.

TMS320DM642 2-seviyeli hafıza/önbellek hiyerarşisine ve çok uzun komut kelimesi (VLIW) mimarisine sahip yüksek-performanslı bir sayısal sinyal işleyicidir. Burada sunulan kodlayıcı sistem, çeyrek piksel hareket dengeleme, rastgele dilim sıralayıcı ve esnek makro-blok sıralayıcı gibi hata esnekliği araçları hariç standart H.264 taban video kodlayıcının içerdiği hemen hemen tüm kısımları içermektedir. Çeyrek piksel hareket dengeleme yerine, hem ışıklılık hem de renklilik değerleri için tam sayı piksel hareket dengeleme gerçeklenmiştir.

H.264 kodlayıcı sistemin tamamının hem bilgisayar hem de DM642 EVM (Değerlendirme Modülü) üzerinde çalıştığı doğrulanmıştır. Esas olarak, kodlayıcı QCIF video serisini (YUV) alır ve onu sıkıştırılmış standart H.264 AnnexB dosya şekline dönüştürür. Kodlayıcı, kabul edilen H.264 JM çözücü ile tamamen uyumludur.

Yeniden oluşturulmuş video, standart JM H.264 çözücüsünün çıktısıyla tamamen aynıdır ve bu video televizyon ekranında görüntülenir. Buna ek olarak, TI geliştirme araçları kullanılarak, kodlayıcı sistemin bütününün performansı gerçek-zamanlı uygulamalar için analiz edilmiştir.

Son olarak, daha yüksek performans için kodlayıcıya hafıza iyileştirmeleri, kod dönüşümleri ve derleyici iyileştirmeleri uygulanmıştır. Sunulan H.264 kodlayıcı saniyede 26.7 adet QCIF çerçevesini kodlayabilme, görüntüleme ve kaydetme yeteneğindedir.

*To my family…*

## ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor Dr. Ayhan Bozkurt who gave me the chance to join Sabancı University and participate in this project. I appreciate very much for his valuable interest, continuous support and guidance.

Also I would like to thank Dr. Ilker Hamzaoğlu and Dr. Hasan Ates for assisting me during the H.264 project.

I would like to thank Dr. Ahmet Onat and Dr. Mehmet Keskinoz for participating in my thesis jury.

I would also like to thank my partners in H.264 project: Sinan Yalcin, Ozgur Tasdizen, Esra Sahin, and Mustafa Parlak.

Lastly, I would like to thank all my friends and my students at Sabanci University.

# TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## ABBREVIATIONS

| | |
|---|---|
| JVT | Joint Video Team |
| DSP | Digital Signal Processor |
| JM | Joint Model |
| MPEG | Motion Picture Experts Group |
| ISO/IEC | International Organization of Standardization, International Electrotechnical Commission |
| AVC | Advanced Video Coding |
| VCEG | Video Coding Experts Group |
| ITU-T | International Telecommunication Union, Telecommunications Standardization Sector |
| NAL | Network Adaptation Layer |
| VCL | Video Coding Layer |
| MB | Macroblock |
| CIF | Common Intermediate Format |
| QCIF | Quarter Common Intermediate Format |
| EDMA | Enhanced Direct Memory Access |
| UVLC | Universal Variable Length Codeword |
| CABAC | Context Based Adaptive Binary Arithmetic Coding |
| CAVLC | Context Based Adaptive Variable Length Coding |
| RTP | Real Time Transport Protocol |
| UDP | User Datagram Protocol |
| IDR | Instantaneous Decoding Refresh |
| PSNR | Peak Signal to Noise Ratio |
| SAD | Sum of Absolute Differences |
| SATD | Sum of Total Absolute Differences |
| CBP | Coded Block Pattern |
| LSB | Least Significant Bit |
| VLIW | Very Long Instruction Word |
| SIMD | Single Instruction Multiple Data |
| PCI | Peripheral Communications Interface |
| JTAG | Joint Test Access Group |
| I/O | Input Output |
| SCOM | Synchronized Communication |
| CPU | Central Processing Unit |
| CCS IDE | Code Composer Studio Integrated Development Environment |
| TI | Texas Instruments |
| EMIF | External Memory Interface |
| RBSP | Raw Byte Sequence Payload |

# CHAPTER 1

# INTRODUCTION

The need for digital video compression is certain. An uncompressed CIF (352x288) 4:2:0 video at 30 frames/sec requires 36.5Mbits/s, and 23.3 Gbytes to store one 90-minute video[1]. Compression is inevitable in order to fit digital video into affordable storages capacities and network bandwidths. H.264 is such a digital video compression standard and is now the most popular and useful one. This emerging standard offers major improvements and produces excellent picture quality with high compression rates. The fact that H.264 is not backwards compatible with other coding standards makes it a milestone in video compression technology.

H.264 provides significant improvement in digital video compression, but it also requires much computation power. Also, implementation of digital compression standards such as H.264 is not an easy task. The trend in implementing video and image processing algorithms is towards using digital signal processors because DSPs are now fast enough and they offer reprogrammable designs. The programmability feature is especially important because it provides us with the ability to tailor video encoder for application needs. One can make innovations in video processing by making use of DSP's flexibility. Moreover, software reuse enables video market to limit design costs. Briefly, H.264 encoder and the design of H.264 especially on a programmable platform is so valuable.

In this thesis, an implementation of H.264 Baseline Encoder on TMS320DM642 digital signal processor is presented. The proposed encoder system includes almost all features of a standard H.264 baseline encoder and its conformance with the H.264 standard is verified by using the H.264 JM Reference Software[2]. For the motion search, full search and hierarchical three step search algorithms are both implemented. The performance results for QCIF video format have shown that real-time execution of

encoder with full search algorithm is possible. Therefore full search is selected for the motion search part.

Besides the software development of H.264, software optimization is also investigated in this thesis. Optimization techniques such as algorithm/system optimization, refinements in memory access pattern, code optimizations and compiler optimizations are experimented. Although the un-optimized encoder's performance is about 3.1 fps, it is shown that the performance can be increased to 26.7 fps for QCIF.

The work proposed in this thesis proved that the huge number of memory accesses is the bottleneck in a video coding system. Unless the memory accesses are optimized, it may not be possible to achieve a real time solution even other optimizations are applied.

The proposed encoder is capable of compressing a QCIF video input into H.264 AnnexB file format. The complete encoder system is realized by using TMS320DM642 Evaluation Module(EVM)[3], a standard definition TV for displaying the reconstructed frames and a desktop computer for storage of output bit-file.

## 1.1 Implementation Complexity

H.264/AVC is a new codec generation featuring an outstanding coding efficiency, but its cost-effective realization is a big challenge. H.264/AVC leads to an average 40% bit saving plus 1-2 PSNR gain compared to previous video coding standards. In this way, it represents the enabling technology for the widespread diffusion of multimedia communication over wired and wireless transmission networks such as xDSL, 3G mobile phones and WLAN. However, this outstanding performance comes with an implementation complexity increase of a factor of 2 for the decoder. At the encoder side, the cost increase is larger than one order of magnitude. This represents a design challenge for resource constrained multimedia systems such as wireless and/or wearable devices and high-volume consumer electronics, particularly for conversational applications (e.g., video telephony), where both the encoder and the decoder functionalities must be integrated in the user's terminal [4].

A single H.264/AVC configuration able to minimize algorithmic performance while minimizing memory and computational burdens does not exist. However, different configurations leading to several performance/cost trade-offs exist. To find these optimal configurations, and hence to highlight the bottenecks of H.264/AVC a good analysis is required. In this thesis, two motion search algorithms (full search and three step hierarchical search) are implemented in order to explore the computational complexity and performance of each. Performance results have shown that computational burden of full search is higher than three step hierarchical search. However, the performance of encoder with full search can be increased up to desired value by applying software optimization techniques.

Data transfer and storage have a dominant impact on the cost-effective realization of multimedia systems for both hardware and software-based platforms. Application specific hardware implementations have the freedom to match the memory and communication bus architectures to the application. An efficient hardware implementation exploits this to reduce area and power. On the other hand, programmable processors rely on the memory architecture that come with them. Efficient use of these resources is crucial to obtain the required speeds as the performance gap between the CPU and DRAM is growing every year [4].

The proposed implementation in this thesis achieves real time execution for QCIF format. Therefore this implementation can be used in a real world application but at low resolution applications. During this implementation at QCIF resolution, a deep exploration of H.264 implementation is done. Therefore this exploration will help us to implement H.264 at higher resolutions. Moreover, because of the flexibility and programmability of dsp implementations this design may be easily adapted for higher performance solutions. Based on this implementation, improvements to H.264 is also possible.

## 1.2 Literature Survey

In the literature, there are examples of real time implementations of video coding algorithms. In [5], optimization of a baseline H.263 encoder on TMS320C6000 is presented. The work presented in that paper focuses on the optimization issue. They do

not write the software code but they use the University of Columbia's (UBC) H.263 encoder software. Starting from a software code written for a desktop application they obtain a real time implementation of H.263 encoder on TMS320C6701. They demonstrated that memory accesses to external memory are a significant bottleneck in the implementation of real-time embedded video systems. It is shown by simulation that most of the time is spent in data access to external memory. Their optimizations resulted in an overall speedup of 61 times over the un-optimized version. They performed optimization in two steps: efficient use of on-chip data and program memory (memory optimizations), and code optimizations of computationally intensive routines in C as well as in assembly language. The total speedup obtained with memory optimizations alone is about 29 times. With the code optimizations alone, a speedup of only 4 is achieved. The affect of code optimizations without memory optimizations is low because the effect of slow off-chip memory accesses becomes the dominant bottleneck as the code becomes more efficient in performing computations. Combination of the memory and code optimizations gives an improvement of 61 times. For the motion estimation, the macroblock and the corresponding search window are copied into internal data memory before the routine was called. In this thesis, I applied the same approach and obtained very good performance. I also copied each macroblock and neighboring pixels of the macroblock into internal data memory before the mode decision, transform, quant routines are called.

The paper called "Parallelization of a H.263 Encoder for the TMS320C80 MVP" [6] describes a real-time implementation on a multiprocessor system. Texas Instruments' TMS320C80 MVP system contains four signal processors and one RISC-processor. The main aspect of the work is parallelization of the encoder in order to exploit the computational power of the multiprocessor system. They try to increase the utilization of the processors to obtain real-time execution however their implementation is below real-time. They concluded that assembler optimizations are necessary. Usually, Implementation on parallel processor architecture gives better results but the implementation becomes so difficult. Especially, the algorithm should be divided into parts for parallel execution. Since the H.264 algorithm is a very complex one, dividing H.264 into parallely executable parts is so difficult. Moreover, the technical paper [7] shows that real-time realization of H.264 on TMS320DM642 digital signal processor is possible.

There is another study related to memory optimizations called “Memory Centric Design of an MPEG-4 Video Encoder” [8]. In this study, high-level memory optimizations are presented. They observed that motion estimation routine repeatedly access the same set of neighboring pixels. To reduce these accesses to frame size memories, a memory hierarchy is introduced for the motion estimation. Heavily used data is copied from large (frame size) to minimal intermediate memories. The solution is more efficient as soon as the cost of extra memory transfers (due to copies) is balanced by the advantage of using smaller memories.

The study called “Code Transformations for Data Transfer and Storage Exploration Preprocessing in Multimedia Processors” [9] says that platform-independent source code transformations can greatly help alleviate the data-transfer and storage bottleneck. This article also covers code rewriting techniques to improve data reuse. They claim that the code should expose maximal data reuse possibilities in order to optimize data transfer and storage. This idea can be implemented in the implementation of H.264 encoder because there is much data reuse in H.264 software. Especially, during the prediction and filtering some pixels are used repeatedly. Instead of accessing the same pixel from the frame memory, this pixel can be stored in a buffer. In other words, we can create data reuse buffers for frequently accessed data. In my H.264 implementation, I introduce buffers for intra 4x4 prediction, intra 16x16 prediction.

Finally, “Video Encoding Optimization on TMS320DM64x/C64x” summarizes the optimization techniques for video encoders on TMS32320DM64x/C64x processors [10]. These techniques include algorithm/system optimization, memory buffering optimization, enhanced direct memory access (EDMA) and cache utilization optimization. The algorithm/system optimization is performed by breaking the algorithm into loops/modules that fits into L1P cache. In this way, they avoid the huge cache miss penalty and CPU stalling. M macroblocks (MB strip) are processed at a time in each loop instead of a single macroblock. Memory buffering optimization is realized by transferring macroblock from the external memory to internal buffer. In this thesis, a similar approach is used. The proposed encoder in this thesis introduces buffers not only for macroblock data but also for intra prediction samples. The third optimization technique that is experimented in this article is the EDMA usage. EDMA is preferred to transfer code/data between L2 SRAM and off-chip memory. The last optimization technique is improving the cache performance for video coding. With the help of TI’s

cache analysis tools, cache efficiency problem areas can be identified, visualized and optimized.

These articles show that video coding applications includes much parallelism. Optimization is possible with techniques such as code transformations, memory optimizations, EDMA and optimizing compiler. Most of the studies show that the bottleneck in video coding is the high number of memory accesses. In this thesis, I experienced the same problem. First, I attempted to start directly with code optimizations. However, code optimizations alone bring up to 10.4fps performance. In order to achieve better performance, algorithmic optimizations and especially improvements in memory access pattern is crucial. Thus, in the second attempt I started with improvements in algorithm and created buffers to optimize memory access pattern. After that, code optimizations and compiler optimizations gave better performance and I achieved performance up to 26.7 fps.

## 1.3 Organization of the Thesis

The organization of the thesis is as follows:

Chapter 2 starts with an overview of H.264 algorithm and specific building blocks of H.264. It also gives a comparison of H.264 with other video coding standards. The description of the algorithm is also briefly explained in chapter 2.

Chapter 3 explains the DM642 digital signal processor core. Understanding the architecture and features of this digital signal processor is necessary for achieving high performance designs.

In chapter 4, the implementation details are explained. Software development phase and software architecture is explained in this chapter. The proposed encoder is presented with software flow graphs. The developed software is transported to DSP environment. This process and the hardware setup for the encoder is also described here.

Chapter 5 gives the performance results of the proposed encoder. Memory optimizations, algorithmic transformations, code optimizations and compiler optimizations for high performance are discussed in this chapter. The compression

efficiency of the proposed encoder with respect to the PSNR is also found in this chapter.

Chapter 6 presents the conclusions and future work.

# CHAPTER 2

# H.264 OVERVIEW

In 1997, the ITU-T Video Coding Experts Group (VCEG) initiated the work on the H.264 standard (formerly known as the H.26L standard). The main objective behind the H.264 project was to develop a high-performance video coding standard by adopting a "back to basics" approach using simple and straightforward design with well-known building blocks. After observing the superiority of video quality offered by H.264-based software over that achieved by the existing most optimized MPEG-4 based software, ISO/IEC MPEG joined ITU-T VCEG by forming a Joint Video Team (JVT) that took over the H.264 project of the ITU-T. The JVT objective was to create a single video coding standard that would simultaneously result in a new part of the MPEG-4 standard (MPEG-4 Part 10 Advanced Video Coding (AVC)) and a new ITU-T Recommendation (H.264) [1]. To this date, 4 major profiles of H.264 have been released. These are named as baseline, main, extended and high profiles. H.264 can be used in many application areas ranging from video conferencing to digital cinema.

Table 2.1: H.264 profiles and their application areas

| **Profile** | **Typical Applications** |
|---|---|
| Baseline | Video Conferencing and Mobile Applications |
| Main | Digital Storage Media and Television Broadcasting |
| Extended | Streaming and Mobile Video Applications |
| High Profile<br>(Fidelity Range Extensions) | Studio Editing, Post Processing, Digital Cinema |

## 2.1 The Specific Coding Parts of H.264

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

1. I slice (Intra-coded slice): The coded slice by using prediction only from decoded samples within the same slice.
2. P slice (Predictive-coded slice): The coded slice by using inter prediction from previously-decoded reference pictures, using at most one motion vector and reference index to predict the sample values of each block.
3. Context-based Adaptive Variable Length Coding (CAVLC) for entropy coding.

The baseline profile consists of these common coding parts plus some others. The complete list of coding parts for baseline profile and their explanations are:

1. Common Parts: I slice, P slice, CAVLC
2. FMO (Flexible macroblock order): Macroblocks may not necessarily be in the raster scan order. The map assigns macroblocks to a slice group.
3. ASO (Arbitrary Slice Order): The macroblock address of the first macroblock of a slice of a picture may be smaller than the macroblock address of the first macroblocks some other preceding slice of the same coded picture.
4. RS (Redundant Slice): This slice belongs to the redundant coded data obtained by same or different coding rate, in comparison with previous coded data of same slice.

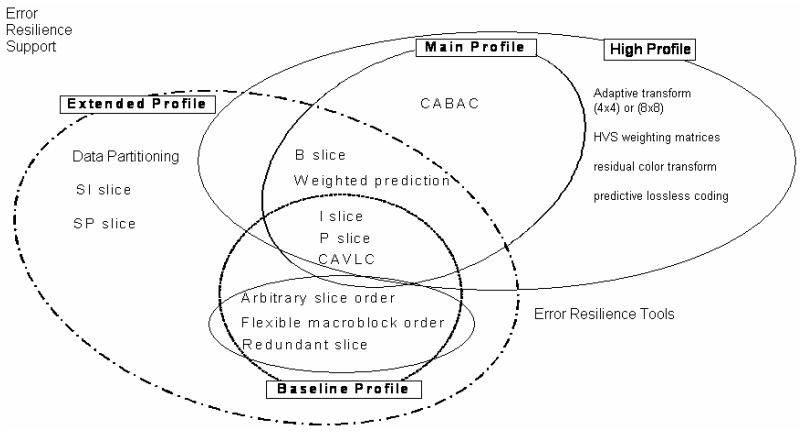


Figure 2.1: Coding parts of H.264 with respect to the profiles.[1]

If a comparison between the profiles is done, baseline is the simplest one. The main profile allows an additional reduction in bandwidth over the Baseline profile through mainly Bi-directional prediction (B-pictures), Context Adaptive Binary Arithmetic Coding (CABAC) and weighted prediction.

B-pictures provide a compression advantage as compared to P-pictures by allowing a larger number of prediction modes for each macroblock. Specifically, bi-predictive coding modes are available for each partition of the macroblock. Here, the partition is formed by averaging the sample values in two reference blocks, generally, but not necessarily using one reference block that is forward in time and one that is backward in time with respect to the current picture. In addition, "Direct Mode" prediction is supported, in which the motion vectors for the macroblock are interpolated based on the motion vectors used for coding the co-located macroblock in a nearby reference frame. Thus, no motion information is transmitted. By allowing so many prediction modes, the prediction accuracy is improved, often reducing the bit rate by 5-10%.

Weighted Prediction allows the modification of motion compensated sample intensities using a global multiplier and a global offset. The multiplier and offset may be explicitly send, or implicitly inferred. The use of the multiplier and the offset aims at reducing the prediction residuals due, for example, to global changes in brightness, and consequently, leads to enhanced coding efficiency for sequences with fades, lighting changes, and other special effects.

Context Adaptive Binary Arithmetic Coding (CABAC) makes use of a probability model at both the encoder and decoder for all the syntax elements (transform coefficients, motion vectors, etc.). To increase the coding efficiency of arithmetic coding, the underlying probability model is adapted to the changing statistics within a video frame, though a process called context modeling.

The context modeling provides estimates of conditional probabilities of the coding symbols. Utilizing suitable context models, given inter-symbol redundancy can be exploited by switching between different probability models according to already coded symbols in the neighborhood of the current symbol to encode. The context modeling is responsible for most of CABAC's 10% savings in bit rate over the Baseline entropy coding method (universal and context adaptive VLC) [1].

The coding parts according to the profiles can be summarized as in table 2.2. The baseline encoder proposed in this thesis includes all coding parts except error resiliency

tools and quarter-pel motion compensation. Quarter-pel motion compensation is replaced with integer-pel. In the future, quarter-pel support will be added.

Table 2.2: H.264 profiles with the coding parts they include.

| | Main | Extended | High | Baseline | Proposed Baseline Encoder |
|---|---|---|---|---|---|
| I Slices | X | X | X | X | X |
| P Slices | X | X | X | X | X |
| Deblocking Filter | X | X | X | X | X |
| Variable Block Size | X | X | X | X | X |
| ¼ Pel Motion Compensation | X | X | X | X | |
| CAVLC/UVLC | X | X | X | X | X |
| Error Resilience Tools (Flexible MB Order, ASO, Redundant Slices) | | X | | X | |
| SP/SI Slices | | X | X | | |
| B Slice | X | X | X | | |
| Interlaced Coding | X | X | X | | |
| CABAC | X | | | | |
| Data Partitioning | | X | | | |
| Weighted Prediction | X | X | | | |

## 2.2 Encoder Complexity

The H.264 standard is significantly more complex than any of the previous coding standards [23]. Motion estimation, for instance, makes use of 7 block sizes from 16x16 to 4x4. Consequently, the H.264 encoder is expected to be significantly more demanding in terms of computations and memory requirements. Moreover, the development of an embedded encoder/decoder where the internal memory size is limited is a challenging task. [10]

According to [7], in order to implement an H.264 Main profile encoder, multiple DSP encoder architectures needed due to the high complexity. But designing a multiple DSP encoder is not an easy task. Multiple processor architecture brings a couple of problems such as the partitioning of the encoder. Partitioning may also have impact on the quality of the encoder. Another issue to be considered in a multiple DSP design is the handling of inter-DSP communication [7].

To summarize, the high complexity of the H.264 video coding standard makes it difficult to implement but provides us with the video quality at low bit rates especially when compared with previous compression standards.

## 2.3 Comparison of H.264 With Other Video Coding Standards

H.264 is not backward compatible with previous standards. The new compession techniques used in H.264 bring great compression efficiency to it. H.264 offers up to 2x compression compared with MPEG-4 simple profile.

An important concept of H.264 is the separation of the system into two layers: a video coding layer (VCL), providing the high-compressed representation of data, and a network abstraction layer(NAL), packaging the coded data in an appropriate manner based on the characteristics of the transmission network [4]. H.264 gives superior error resilience due to VCL and NAL layer enhancements and error resiliency tools. VCL enables efficient transmission of video data on network by representing video content in integer number of byte units. H.264 is very adaptive to different applications from digital TV to video conferencing. It may operate with low delay constraints for real-time applications or higher delay constraints for applications which require more processing power, such as video content storage

The introduction of the switching mechanism enables fast random access for video decoders. SP and SI frames may be used for switching from low bit rate to high bit rate in the same video stream according to the available bandwidth.

The basic building blocks of the H.264 encoder are similar with the previous standards'. The transform, quantization, motion estimation, motion compensation and entropy coding are found on almost all video coding standards. However, there are major improvements done inside these basic building blocks. For instance, H.264 uses 16x16 or 4x4 block sizes in the intra coding part. Another improvement in H.264 is the addition of new intra prediction modes. For the inter prediction, there are 7 block types changing from 16x16 to 4x4. Moreover, luma motion vectors have quarter pel resolution whereas chrominance vectors have 1/8 pixel. This results in increased precision of motion vectors. Context Adaptive Variable Length Coding (CAVLC) and

Context Based Adaptive Arithmetic Coding (CABAC) are also new. Lastly, in H.264 loop filter is introduced to reduce the blocking artifacts.

## 2.4 Description of H.264

H.264 has a layered structure. It has two different layers one of which is Network Abstraction Layer (NAL) and the other is Video Coding Layer (VCL). The NAL layer abstracts the VCL data. It includes the header information about the VCL format. NAL is appropriate for conveyance by the transport layers or store media. NAL unit (NALU) is a generic format which is used in both packet based and bit-streaming systems. Second layer which is VCL is the core coding layer. VCL concentrates on attaining maximum coding efficiency rather than the transportation.[21]

### 2.4.1 Network Abstraction Layer (NAL)

A coded H.264 video sequence consists of a series of Network Abstraction Units (NALUs), each containing an RBSP (Raw Byte Sequence Payload). An example of a typical sequence of RBSP units is shown in figure 2.2. Each of these units is transmitted in a separate NAL unit. The header of the NAL unit (one byte) signals the type of RBSP unit and the RBSP data makes up the rest of the NAL unit.

H.264 introduces the concept of parameter sets which contain information that can be applied to a large number of coded pictures. A sequence parameter set contains parameters which are applied to a complete video sequence. Similarly, a picture parameter set contains parameters which are applied to one or more decoded pictures within a sequence.

Parameters in the Sequence Parameter Set include an identifier, limits on frame numbers and picture order count, the number of reference frames that may be used in decoding, the decoded picture height and width and the choice of progressive or interlaced (frame or field) coding.

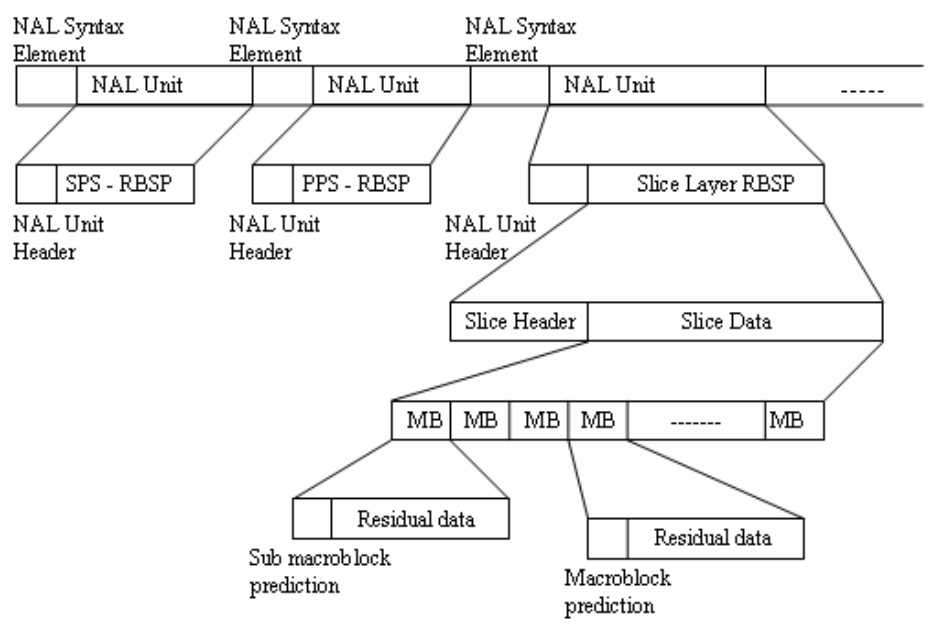


Figure 2.2: The structure of a H.264 coded video sequence.

Each Picture Parameter Set includes an identifier, a selected sequence parameter set id, a flag to select VLC or CABAC entropy coding mode, the of slice groups in use, the number of reference pictures in list0 and list1 that may be used for prediction, initial quantizer parameters and a flag indicating whether the default deblocking filter parameters are to be modified.

In a typical application, coded video is required to be transmitted or stored together with associated audio tracks and side information. It is possible to use a range of transport mechanisms to achieve this, such as Real Time Protocol and User Datagram Protocol (RTP/UDP).

Earlier video coding standards such as MPEG-1, MPEG-2 and H.263 did not explicitly define a format for storing compressed audiovisual data in a file. The MPEG-4 file format and AVC File Format are designed to store MPEG-4 Audio Visual and H.264 Video data respectively. Both formats are derived from the ISO Base Media File Format, which in turn is based on Apple Computer's Quick Time Format. In my implementation, the encoder produces an output bitstream in Annex B byte stream file format which is described in the ITU-T H.264 Recommendation [11].

### 2.4.2 Video Coding Layer (VCL)

Basically, the coding of a macroblock is obtained with the flow shown in the figure 2.3.

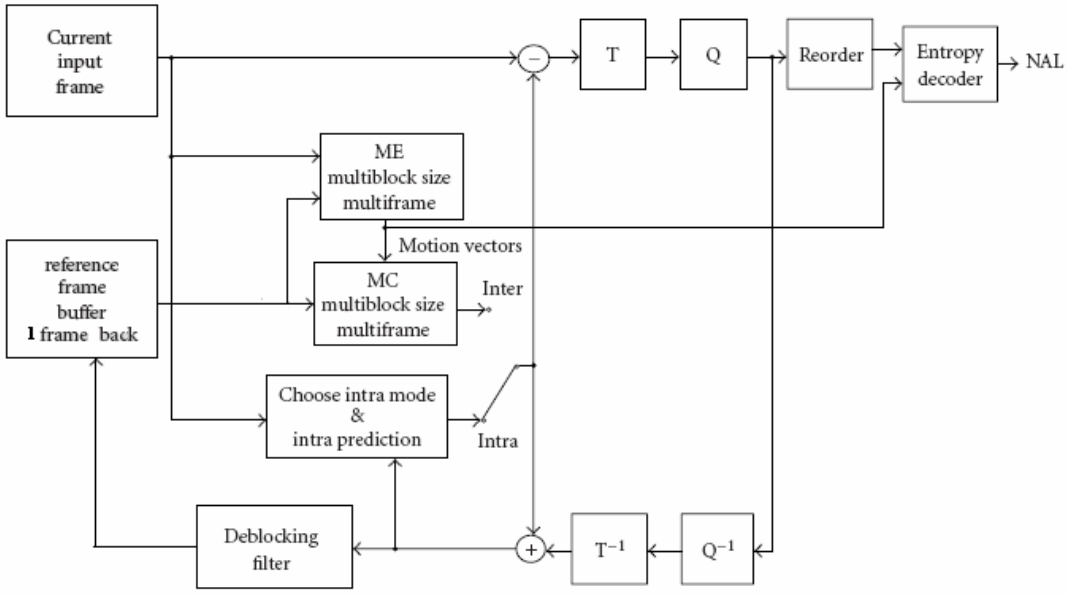


Figure 2.3: Block diagram of H.264 encoder

#### 2.4.2.1 Intra Prediction

Intra prediction process exploits the spatial redundancy between adjacent macroblocks in a frame. Intra predicted frames usually have better PSNR than inter coded frames; however they require much more bits to encode and decrease compression rate. Therefore less number of frames of a video sequence is coded in intra that in inter. During the encoding process, propagation errors occur due to the successive inter-coded pictures. At that time, an intra-coded picture should be inserted in order to reduce the propagation of errors to the next predictions. Subsequent inter prediction is done based on this new intra picture. In other words, this intra-coded picture refreshes the prediction and is therefore called Instantaneous Decoding Refresh (IDR) picture. In my implementation, the IDR period can be determined by a parameter inside the software. During the tests it is fixed at 20; meaning that every one frame out of 20 is intra-coded.

The inputs of intra prediction process are previously reconstructed samples prior to the deblocking filter process from neighboring macroblocks. A prediction block is formed in this process and it is subtracted from the current block before encoding. For the luma samples, prediction is formed for each 4x4 block or for a 16x16 macroblock.

The intra_4x4 mode is based on predicting each 4x4 luma block separately and is well suited for coding parts of a picture with significant detail. The intra_16x16 mode on the other hand, performs prediction of the whole 16x16 luma block and is more suited for coding very smooth areas of a picture.

There are a total of nine optional prediction modes for intra_4x4 luma block, four modes for intra_16x16 luma block.

Table 2.3: 4x4 Luma block intra prediction modes

| | |
|---|---|
| **Mode 0 : Vertical** | The upper samples are extrapolated vertically |
| **Mode 1 : Horizontal** | The left samples are extrapolated horizontally |
| **Mode 2 : DC** | All samples are predicted by the mean of A, B, C, D, I, J, K, L |
| **Mode 3 : Diagonal Down Left** | The samples are interpolated at a 45° angle between lower left and upper samples. |
| **Mode 4 : Diagonal Down Right** | The samples are extrapolated at a 45° angle down and to the right. |
| **Mode 5 : Vertical Right** | Extrapolation at an angle of approximately 26.6° to the left of vertical. |
| **Mode 6 : Horizontal Down** | Extrapolation at an angle of approximately 26.6° below horizontal. |
| **Mode 7 : Vertical Left** | Extrapolation (or intrapolation) at an angle of approximately 26.6° to the right of vertical. |
| **Mode 8 : Horizontal Up** | Interpolation at an angle of approximately 26.6° above horizontal |

Table 2.4: 16x16 Luma block intra prediction modes

| | |
|---|---|
| **Mode 0 : Vertical** | The upper samples are extrapolated vertically. |
| **Mode 1 : Horizontal** | The left samples are extrapolated horizontally. |
| **Mode 2 : DC** | All samples are predicted by the mean of A, B, C, D, I, J, K, L. |
| **Mode 3 : Planar** | A linear 'plane' function is fitted to the upper and left-hand samples. This works well in smoothly varying luminance. |

The four prediction modes of 8x8 intra chroma blocks are similar to the 16x16 luma block prediction modes, but only the numbering of the modes is different.(Mode

0: DC, Mode 1 : Horizontal, Mode 2 : Vertical, Mode 3 : Plane. Also it is noticeable that both chroma components always use the same prediction mode.

In order to reduce the number of bits that comes from coding the choice of intra prediction modes for each 4x4 blocks, a predictive coding mechanism is developed. This mechanism exploits the correlation between intra 4x4 modes of neighboring blocks. It takes the prediction modes of previously coded 4x4 blocks and finds a "most probable mode" for the current block. If the current block's mode is same as the "most probable mode", then the encoder send a flag with a value of 1 instead of sending the prediction mode. Oppositely, if current block's mode is different from "most probable mode" then the flag is sent with a value of 0. The prediction mode is also sent but with the following change:

- If the current mode is smaller than "most probable mode" it is sent without any change.
- If the current mode is larger than "most probable mode" it is sent after being decreased by one.

Table 2.5 Coding of intra 4x4 prediction modes

| (most probable mode=2) | | |
|---|---|---|
| Current mode | Description | Code |
| 1 | Flag and mode is coded. Mode value is sent directly without any change (i.e. 1 is sent) | 0-001 |
| 2 | Only flag is coded since this mode is equal to the most probable mode. | 1 |
| 3 | Flag and mode information is coded. Mode value is sent after being decreased by 1 (i.e. 2 is sent) | 0-010 |

### 2.4.2.2 Inter Prediction

Inter Prediction process exploits temporal redundancies in the video stream. In other words, it uses the similarity between successive frames for the compression. Inter prediction creates a prediction model from one or more previously encoded video frames of fields at variable block sizes. H.264 supports a range of block sizes from

16x16 down to 4x4 and fine 1/4 sample motion vectors for luma as well as 1/8 sample motion vectors for chroma component. Using multiple reference frames in inter prediction results in better compression efficiency. In the proposed encoder implementation only one reference frame is used but the design can be easily adapted to support multiple number of reference frames.

A macroblock can be partitioned into 16x8, 8x16, 8x8 blocks or remain as 16x16. If it is partitioned into 8x8 blocks, then these 8x8 blocks can be further partitioned into sub-blocks of 8x4, 4x8, 4x4 or remain as 8x8. A macroblock partition can not be mixed with sub-macroblock partition. That is to say, wen cannot have 16x8 and 4x8 partitions inside a macroblock.

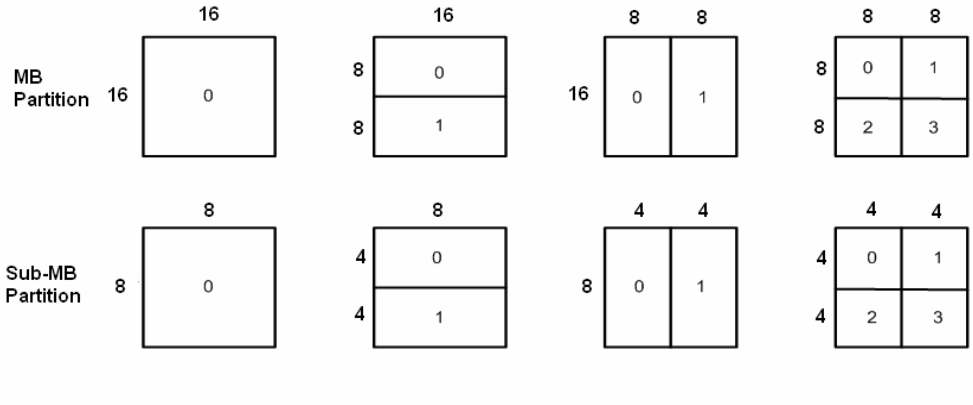


Figure 2.4: Macroblock partitioning in inter prediction.

The macroblock partitions and sub-macroblock partitions gives rise to a large number of possible combinations within each macroblock. In an ideal encoder, a large partition size should be selected for a homogeneous area whereas a small partition size should be selected for a detailed area. Moreover, choosing a small partition size may give better prediction but it results in increased number of motion vectors and reference indexes since each block has its own vector and reference frame. Thus, small partitioning does not only brings better prediction but also larger number of bits. Finding the optimal partition size is one of the challenging tasks in an encoder.

A chroma motion vector is derived from the corresponding luma motion vector. Since the accuracy of luma motion vectors is one-quarter sample and chroma has half resolution compared to luma, the accuracy of chroma motion vectors is one-eight sample. That is, a value of 1 for the chroma motion vector refers to one-eight sample displacement. When the luma vector applies to 8x16 luma samples, the corresponding

chroma vector applies to 4x8 chroma samples and when the luma vector applies to 4x4 luma samples, the corresponding chroma vector applies to 2x2 chroma samples. The horizontal and vertical components of each luma motion vector are halved when applied to the chroma blocks.

The encoder developed in this thesis does not support quarter-pel motion compensation. Instead, it is based on integer motion search and compensation. In order to make the output bitstream decodable by the JM reference decoder[2], the resultant motion vectors are multiplied by 4. In this way, the JM reference decoder[2] uses pixel locations whose indexes are multiples of 4 (i.e. integer pixel locations) during the motion compensation. The same is true for chroma motion compensation. In the near future, I will make it to support quarter pel motion compensation.

#### 2.4.2.2.1 Hierarchical Three Step Search

There are several motion search algorithms used for video compression. Among these, the full search algorithm gives best PSNR but it is not an efficient solution since it requires much computation. For real world implementations much intelligent algorithms with lower computation requirements are desired. Hierarchical three step search is such an algorithm which decreases the number of computations by 10 compared with full search [12].

According to hierarchical search, the search is done in three steps:

1-) Level2: At the first step, current frame and reference frame are averaged and down-sampled by 4 and a full search with a search range of 4 is performed. For this full search, we only use SAD of the current and reference blocks.

2-) Level1: The resultant vectors from level2 are passed to this step. The current frame and reference frame are now averaged and down sampled by 2 and again a full search with a search range of 4 is performed at the location shown by the vectors passed from level2. For this full search, we only use SAD of the current and reference blocks.

3-) Level0: At this final step, a final full search with a range of 4 is performed on the original current and reference frames. However, for this search we do not only use the SAD values but also $\lambda$*R. More specifically, a cost is calculated for each search mode and macroblock partition or sub-macroblock partition.

$$\cos t = SAD + \lambda \bullet R\{mv\_cand - mv\_pred\}$$

The R function in the cost formula returns us the number of bits required to transmit a vector with the given value. mv_cand is the candidate vector for that search location and mv_pred is the vector predicted using motion vector prediction.

#### 2.4.2.2.2 Motion Vector Prediction

Luminance motion vectors of neighboring blocks are highly correlated, so that each motion vector is predicted from early previously coded blocks. After finding a prediction vector, the difference between the current vector and the predicted vector is transmitted to the decoder. In other words, not the original vector but its difference from the prediction vector is coded in order to reduce the number of transmitted bits. At the decoder side, the predicted vector is formed in the same way and added to the transmitted motion vector difference in order to find current vector.

#### 2.4.2.2.3 Transform and Quantisation

H.264 uses three transforms depending on the type of the data to be coded.

1. Hadamard transform for the 4x4 array of luma DC coefficients in macroblocks with type intra 16x16.
2. Hadamard transform for the 2x2 array of chroma DC coefficients.
3. DCT based transform for all other 4x4 blocks in the residual data.

The transformation matrixes and other detailed information about transform and quantization process can be found in [11].

#### 2.4.2.2.4 Coded Block Pattern (CBP)

Coded block pattern is a parameter sent by the encoder to the decoder. It specifies which 8x8 blocks in a macroblock are coded and which are not. According to the coefficients coming out of transform and quant, 8x8 blocks which do not contain any non-zero 4x4 blocks are determined and not coded. CBP is a value that tells the decoder

how many and which blocks are coded and transmitted. This information is hidden in the binary equivalent of cbp value. The least significant bit of cbp represents the zeroth 8x8 luma block. If that block is coded than LSB is 1, else it is 0. Likewise, the next three bits corresponds to other three 8x8 blocks. For instance, if all 8x8 luma blocks are coded the least significant four bits of cbp is 1111. If second 8x8 luma block is not coded then the least significant four bits of cbp is 1011. The leading 2 bits of cbp are for chroma DC and chroma AC blocks. If only chroma DC is found in the bitstream then leading 2 bits is 01; if both are found it is 10 and it is 00 when neither chroma DC nor chroma AC is found. Some of the possible values of cbp and its explanation are given in Table 2.6.

Table 2.6: Illustration of calculating cbp values for some coded blocks.

| **CBP Value (binary equivalent)** | **Coded Blocks** |
|---|---|
| 47 (101111) | All Luma 8x8 Blocks + Chroma DC + Chroma AC |
| 43 (101011) | 0th, 1st, 3rd Luma 8x8 Blocks + Chroma DC + Chroma AC |
| 31 (011111) | All Luma 8x8 Blocks + Chroma DC |
| 15 (001111) | All Luma 8x8 Blocks |
| 7 (000111) | 0th, 1st, 2nd Luma 8x8 Blocks |

#### 2.4.2.2.5 Entropy Coding

Entropy coding aims at compressing the generated bitstream so that fewer bits are used for coding. It is uniquely decodable or in other words does not provide any error. The generated syntax elements and residual data are entropy coded. Context Adaptive Variable Length Coding (CAVLC) and Context Adaptive Binary Arithmetic Coding (CABAC) are two entropy coding methods of H.264. In the proposed baseline H.264 encoder CAVLC is developed. Context Adaptive Variable Length Coding (CAVLC) is found in all profiles whereas Context Based Adaptive Binary Arithmetic Coding (CABAC) is found in main profile. Other details about entropy coding are listed as follows:

All syntax elements other than residual transform coefficients are encoded by the Exp-Golomb code (UVLC)

Zig-zag ordered, 4x4 (and 2x2) blocks of transform coefficients are encoded by CAVLC.

Coefficients of residual data are scanned in zig-zag order.

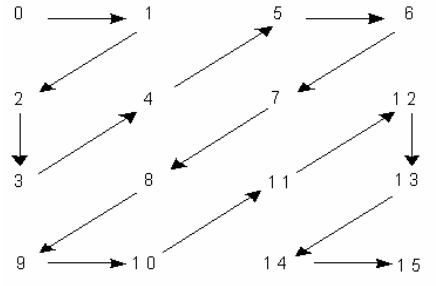


Figure 2.5: Zig Zag scan order.

**2.4.2.2.6 Deblocking Filter**

This adaptive filter is designed to reduce the blocking artifacts in the block boundary and prevent propagation of accumulated coded noise. Filtering is applied to horizontal or vertical edges of 4x4 blocks in a macroblock adaptively. The filter smoothens block edges, improving the appearance of decoded frames. The filtered image is then used for motion-compensated prediction of future frames. The inter prediction for a P-Slice following an I-Slice is carried out using the filtered version of the I-Slice. However, the intra prediction inside the I-Slice is done using the previously reconstructed but unfiltered macroblocks.

The inclusion of deblocking filter before motion compensated predition stage is beneficial in terms of compression efficiency. Because the filtered image is much more resembles the original image than a blocky, unfiltered image.

The main principle of this filter is that it adjusts the amount of filtering adaptively according to the coding modes of neighbouring blocks and the gradient of image samples across the boundary. More detail about deblocking filter can be found in [11].

# CHAPTER 3

# TEXAS INSTRUMENTS TMS320DM642 DSP

Programmable digital signal processors (DSPs) are increasingly important in a wide range of video and imaging applications, such as machine vision, medical imaging, security monitoring, digital cameras and printers, and a large number of consumer applications driven by digital video processing including DVDs, digital TV, video telephony and many others. The importance of multimedia technology, services and applications is widely recognized by microprocessor designers. The number of special-purpose multimedia processors such as the Trimedia processor from Philips, Mitsubishi's multimedia processor and digital media processors of Texas Instruments are becoming more popular. These special purpose multimedia processors are being used in low-cost embedded applications such as set-top boxes, wireless terminals, digital TVs, DVDs and mobile applications.

Multimedia applications are characterized by requirements for processing flexibility, sophisticated algorithms and high data rates. One of the processor architectures to exploit parallelism of multimedia applications is the very long instruction word (VLIW) architecture. VLIW processors can exploit instruction level parallelism (ILP) in programs [13]. TMS320DM642 device is based on VLIW architecture and it seems to be a perfect choice for H.264 encoder implementation.

## 3.1 Overview of DM642 DSP Core

The TMS320DM642 device is based on the high-performance, very-long-instruction-word (VLIW) architecture VelociTI.2 [14] developed by Texas Instruments. The key features of this device such as VLIW architecture, 2-level memory/cache

hierarchy, and EDMA engine makes it an excellent choice for computationally intensive video/image applications such as video coding and analysis.

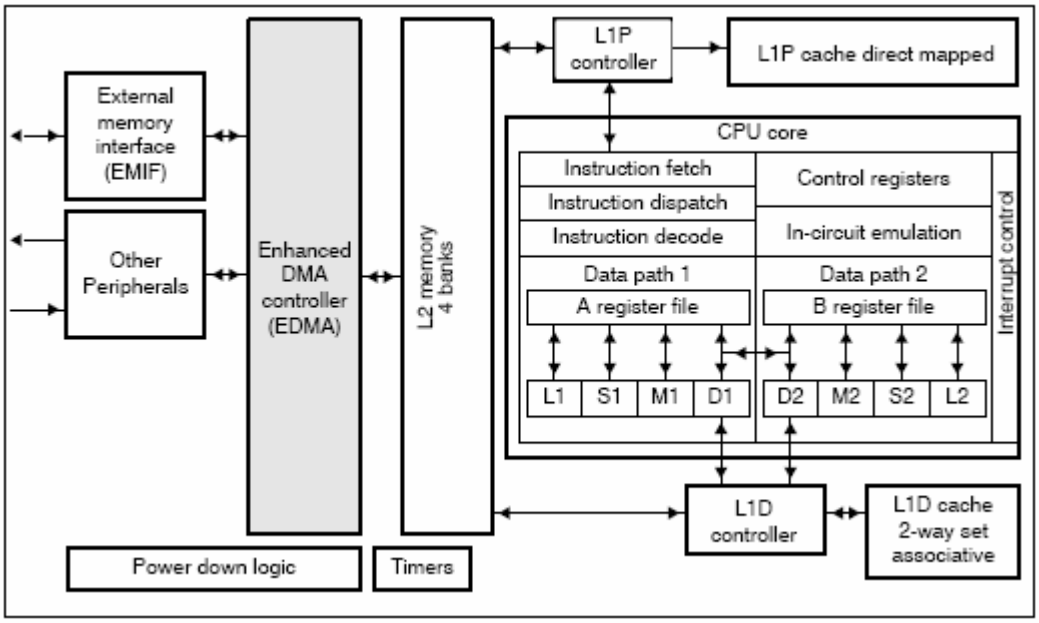


Figure 3.1: TMS320C64x DSP Block Diagram [14].

DM642 DSP core and essential features are listed as follows:

- The VelociTI.2 extensions in the eight functional units of DM642 include new instructions which accelerate performance in video and imaging applications.
- Two general-purpose register files (A and B)
- Eight functional units (.L1, .L2, .S1, .S2, .M1, .M2, .D1 and .D2)
- Two load-from-memory data paths (LD1 and LD2)
- Two store-to-memory data paths (ST1 and ST2)
- Two data address paths (DA1 and DA2)
- Two register file data cross paths (1X and 2X)
- It has a 16Kbytes direct mapped L1P program cache with 32-byte cache line (8-cycle L1P cache miss penalty). The L1D cache is 16Kbytes 2-way set-associative and has a 64-byte cache line. (6-cycle L1D cache miss penalty).
- 256Kbytes of internal memory can be mapped either RAM or cache (flexible RAM/cache allocation, 8-cycle L2 cache miss penalty). L2 4-way set associative cache has 128 byte cache line.

### 3.1.1 Register Files

There are two general purpose register files (A and B) in the C6000 data paths. For the C64x 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 C64x, 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 C64x 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 long quantities, are stored in register pairs, with the 32LSBs of data placed in an even-numbered register and the remaining 8 or 32 MSBs in the next upper register (which is always an odd-numbered 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.

### 3.1.2 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 s almost identical to the corresponding unit in the other data path. The C64x contain many 8-bit and 16-bit instructions to support video and imaging applications.

### 3.1.3 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 the register file B.

Most data lines in the CPU support 32-bit operands, and some supporting 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 C64x 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. 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.

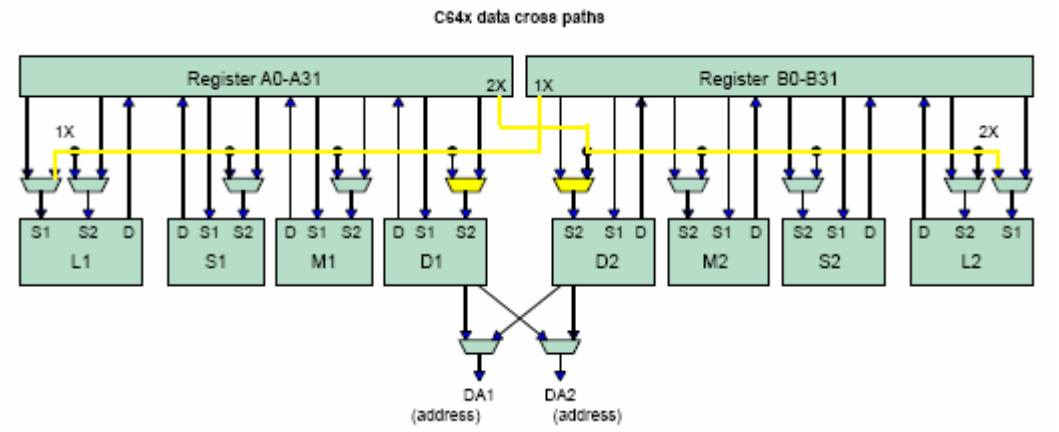


Figure 3.2: C64x Data Cross Paths [14]

On the C64x, 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.

The C64x 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.

### 3.1.4 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.

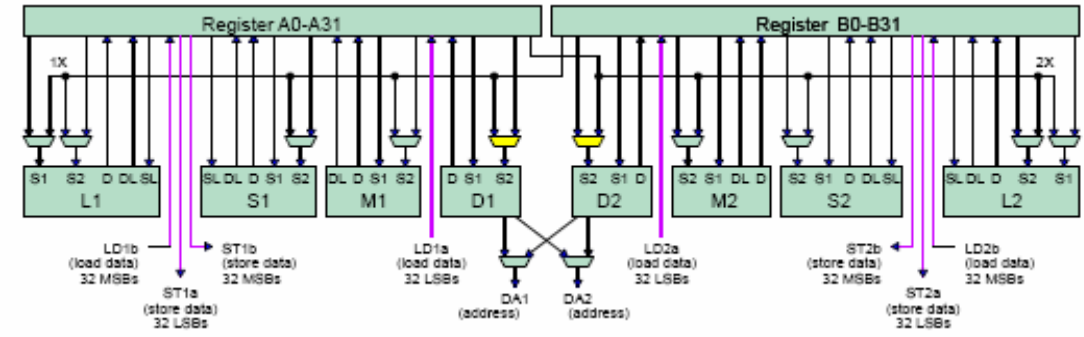


Figure 3.3: C64x Memory Load and Store Paths [14]

The C64x 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. 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 32MSBs 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 C64x 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 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. C64x 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.

### 3.1.5 Additional Functional Unit Hardware

Additional hardware has been built into the eight functional units of the C64x. 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 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.

It is important to note that the C64x provides a comprehensive set of data packing and unpacking operations to allow sustained high performance for the quad 8-bit and dual 16-bit hardware extensions. Unpack instructions prepare 8-bit data for parallel 16-bit operations. Pack instructions return parallel results to output precision including saturation support.

### 3.1.6 DM642 Cache Architecture

On DM642 devices, the CPU interfaces directly to dedicated level-one program (L1P) and data (L1D) caches of 16Kbytes each. These caches operate at the full speed of CPU access. A second level unified L2 program/data memory provides flexible storage. Figure 3.5 depicts an example for different configurations of L2 cache with a size of 256Kbytes. One configuration for L2 is entirely mapped SRAM. The other configurations have both SRAM and a 4-way set associative cache of various sizes. Mapped SRAM can be used for streaming video data and critical sections of code such as interrupt service routines. Cache is useful for most of the program and data structures.

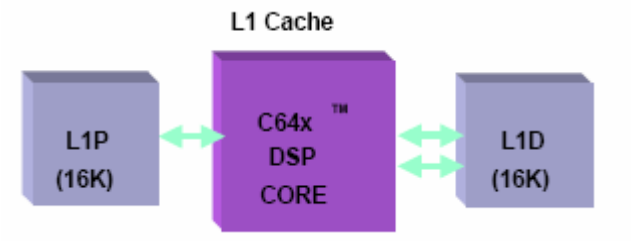


Figure 3.4: DM642 L1/L2 Cache [14].

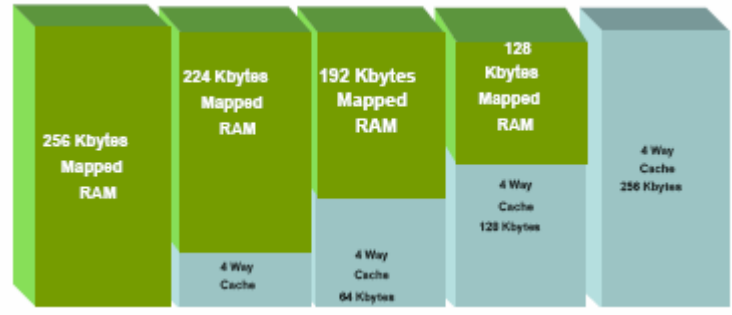


Figure 3.5: Partitioning internal memory into L2 cache/ram

The C64xx family of DSPs has a large byte addressable address space. Program code and data can be placed anywhere in the unified address space. Addresses are always 32-bits wide. Portions of internal memory can be remapped in software as L2 cache rather than fixed RAM.

There are two methods for transferring data from one part of the memory to another, these methods are:

1. By using CPU
2. By using DMA

If a DMA is used then the CPU only needs to configure the DMA. While the transfer is taking place, the CPU is free to perform other operations. The EDMA controller handles all data transfers between the level-two (L2) cache/memory controller and the device peripherals on the DM642 DSP. These data transfers include cache servicing, user-programmed data transfers, and host accesses.

The EDMA provides us with the ability to transfer data with zero overhead. It is clear that the EDMA and CPU operations can be independent. However, if the CPU and EDMA both try to access the same memory location, arbitration will be performed by the program memory controller. We should also take into account that the following conditions may limit the performance:

1. EDMA stalls when there are multiple transfer requests on the same priority level.
2. EDMA accesses to L2 SRAM with lower priority than the CPU.

Some basic concepts concerning memory/cache hierarchy and EDMA engine need to be considered for an algorithm implementation. When code size is bigger than the size of L1P, L1P cache misses can occur, CPU stalls until the required code is fetched. Similarly, L1D cache misses and CPU stalls occur when the data do not fit in the L1D. All L1P and L1D misses are serviced by L2 cache/SRAM. L2 cache misses occur if the code and data size is bigger than the size of L2 cache. Cache friendly program partitioning and data transfer handling (e.g. reducing L1/L2 misses) are two critical factors to guarantee video encoder optimal performance [10].

To sum up, the DM642 digital signal processor addresses the needs of video and imaging application developers. DM642 is based on the C64x CPU which includes special instructions to accelerate the performance of video and imaging processing. Also, the RISC-like instruction set and extensive use of pipelining in C64x allow many instructions to be scheduled and executed in parallel. A high performance two-level cache design allows the CPU to operate at the maximum rate. There are a number of available C intrinsic functions that can be used to increase the efficiency of the code. Finally, the existence of EDMA provides us to transfer data with zero overhead.

# CHAPTER 4

# SOFTWARE DEVELOPMENT AND DSP REALIZATION OF ENCODER

The design flow of complex multimedia systems such as video codecs typically starts with an algorithmic development. Algorithmic development focuses on algorithmic performance (peak signal-to-noise ratio (PSNR), visual appearance, and bit rate). The algorithmic specification is typically released as a paper description plus a software verification mode (as the H.264 JM Reference software [2]). Usually, the software model is not optimized for a cost-effective realization since its scope is mainly a functional algorithmic verification and the target platform is unknown. Moreover, in the case of multimedia standards such as ITU-T and ISO/IEC video codecs, the verification software models (up to 100.000 C-code lines) are written in different code styles since they are the results of combined effort of multiple teams. Therefore it is a must to rewrite the software code in order to realize an actual system [4].

At first, I started with analyzing the JM Reference Software [2]. Because the JM Software is not written for a real application purpose, its performance is very poor. Even at the desktop computer its performance is low as 0.5 frames per second. The source code is very huge and memory consuming so that it is almost impossible to run it on an embedded processor. But JM software is very useful for studying the details of H.264 algorithm. It is also very practical for the test and verification of a new developed encoder. At many times, I referred to the JM software in order to extract the algorithm. I also solved many bugs in my encoder software thanks to the JM reference software. In the end, I obtained an encoder which is fully conformant with the JM. I observed this by comparing the reconstructed frames of my encoder with the ones of the JM decoder which is decoding the bit-stream formed by my encoder. I checked that these frames are perfectly matched.

Most of the software is coded by me, but I also copied some parts of the JM. For instance, the deblocking filter and boundary strength calculation functions are taken from JM.

## 4.1 Software Development

All the source files are written in C programming language. The complete software development process can be divided into two phases. In the first phase, the encoder software is developed on a desktop computer using the Microsoft Visual Studio development environment. In the second phase, this software is transferred from the desktop computer to the embedded DSP platform and some additional coding for embedded parts is done. Embedded software development to run the Philips SAA7105 encoder device which displays video output on TV, is such an additional coding. Texas Instruments Code Composer IDE is used as the embedded development environment. The embedded phase is much more difficult compared with the first one. At the desktop computer, we have huge resources such as memory and computation power whereas those are limited for the embedded devices. The DM642 digital signal processor's clock speed (i.e. 720MHz) is less than a Pentium processor, but it has high enough computation power because of its parallel architecture. However, we have to make sure that we make use of its parallel architecture and additional features efficiently. The DM642 DSP is capable of executing eight instructions in parallel at the clock speed of 720 MHz. If we are able to utilize these execution units then we can benefit from its processing power and obtain a high performance solution. Otherwise, the performance may not be satisfactory. Providing a parallel execution for H.264 encoder at an embedded DSP is not an easy task. To achieve this, one has to know both the H.264 encoder algorithm and the target processor architecture very well. Moreover, both the algorithm and the software implementation of the algorithm should be analyzed in depth.

### 4.1.1 Software Flow Graphs

In this section, I will briefly explain the software modules with their flow graphs to understand the software architecture of the proposed H.264 encoder.

#### 4.1.1.1 Main

The whole software starts with the function “main” and ends with the function “terminate sequence”. I will explain all of the functions shown in the flow graph in figure 4.1

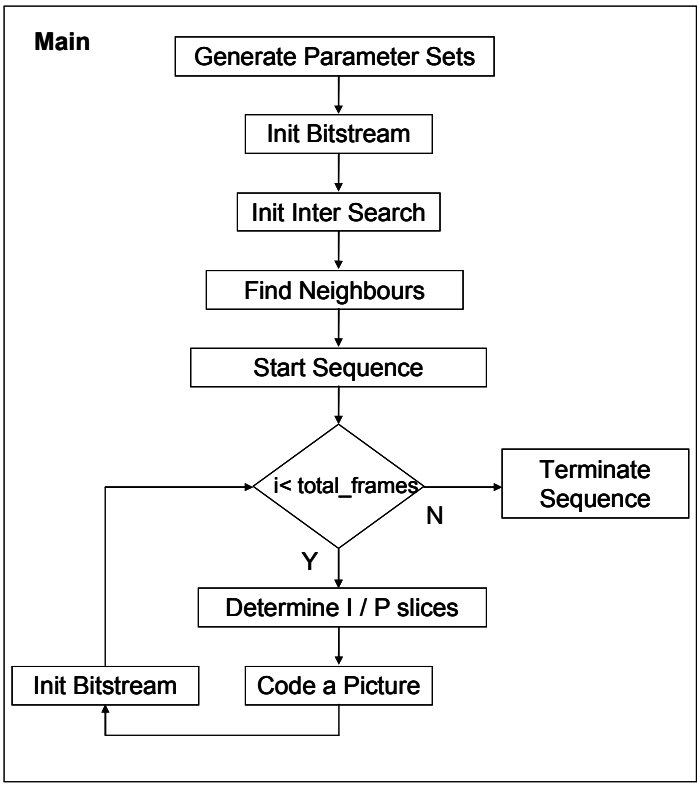


Figure 4.1: Flow graph of main

**Generate Parameter Sets:** This routine creates a sequence and a picture parameter set and fill these structures with the necessary values. When calculating these values, input variables such as the “profile type” or “number of frames to be encoded” are used.

**Init Bitstream:** This function simply creates a stream to write the output of the encoder. Also the first byte of the stream array is set to zero.

**Init Inter Search:** It simply sets the motion vector arrays to zero. This initialization is necessary because the vector values are used by the vector predictor.

**Find Neighbors:** During the intra prediction and deblocking filter the neighboring samples or blocks are needed. Calculation of the index of these samples or blocks is very frequent and repeating. Therefore, redundant calculations should be eliminated. This function is called for each macroblock and calculates the neighboring relations for once. Other functions uses this information and do not recalculate it. This is a very good example of algorithm/system level optimization. It decreases down the number of operations and really improves the performance.

**Start Sequence:** This function is going to described in detail in section 4.1.2.2.

In the main function, there is loop which is iterated for total number of frames times. The "number of frames" information is given by the user as an input.

**Determination of I-P Slices:** I defined a intra period which determines the current slice as I or P. During the experiments shown in this thesis the intra period is given as 20. This means that each $20^{th}$ frame is assigned as intra, others as inter.

**Code a Picture: :** This function is going to described in detail in section 4.1.2.3.

**Terminate Sequence:** It simply closes the AnnexB output bit stream file.

#### 4.1.1.2 Start Sequence

This section generates the sequence parameter set NALU and picture parameter set NALU. These NALU's are written to AnnexB byte stream. The output file is also opened here. The flow graph of start sequence is given in figure 4.2

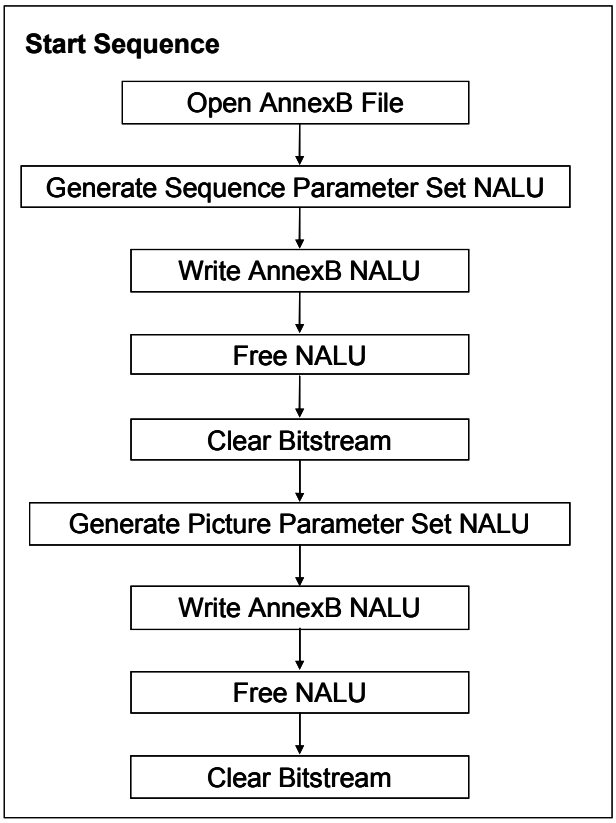


Figure 4.2: Flow graphs of start sequence

#### 4.1.1.3 Code a Picture

Code a picture function is the core of the program. Basically, this function takes a frame, codes reconstructs the frame and filters it. The details of this function and the explanation for its sub-functions are given in the flow graph in figure 4.3.

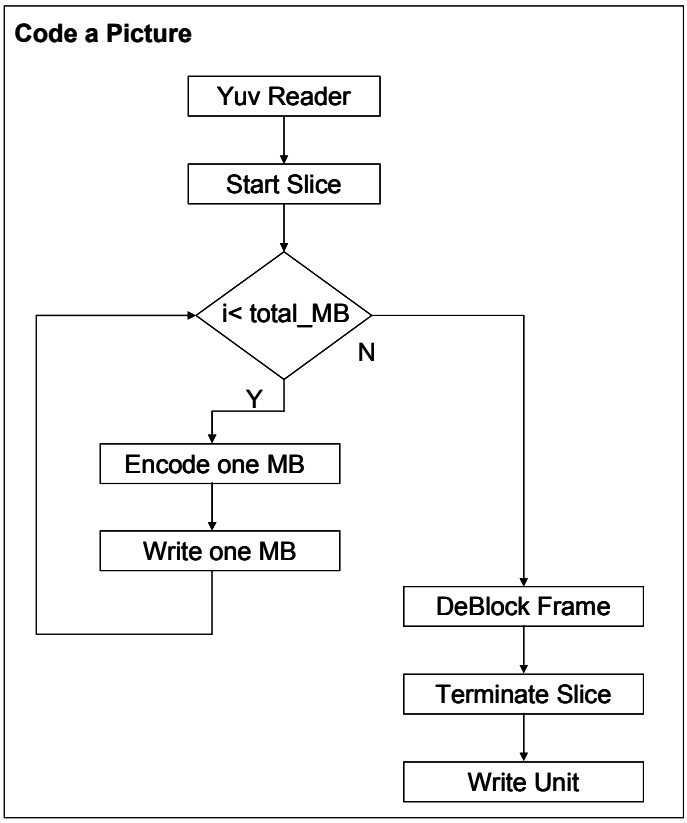


Figure 4.3: Flow graph code a picture

**Yuv Reader:** It reads and stores a new frame as the input. Additionally, it adjusts the pointers reference frame pointer and current frame pointer as required. When a new frame is read, the current frame pointer is set to point to this frame. The reference frame pointer is set to point to the previous frame. Instead of copying the frames to the memory locations, only the pointers that points to these locations are swapped. This approach is very important, because it avoids copying memory from one location to antoher location and decreases the number of operations.

**Start Slice:** It calculates and writes a slice header.

In the code a picture function, there is a loop which is iterated for the total number of macroblock times. Each macroblock in the frame is processed in raster scan order.

**Encode One Macroblock:** This function will be explained in section 4.1.1.4.

**Write One Macroblock:** The macroblock header, motion information, CBP, luma coefficients and chroma coefficients are written to the output bit stream.

**Deblock Frame:** This function is copied from JM Reference Software and adapted to my encoder after some modifications.

**Terminate Slice:** It puts a "1" bit after the end of a slice. The output bit stream must be byte aligned. In other words, it must end at the byte boundary. If it does not end at the byte boundary, then we fill the stream with zeros until the end of the byte.

**Write Unit:** Writes a NAL Unit of a slice to AnnexB byte stream.

#### 4.1.1.4 Encode one Macroblock

When a new macroblock comes first of all we check whether it belongs to an I-SLICE or a P-SLICE.

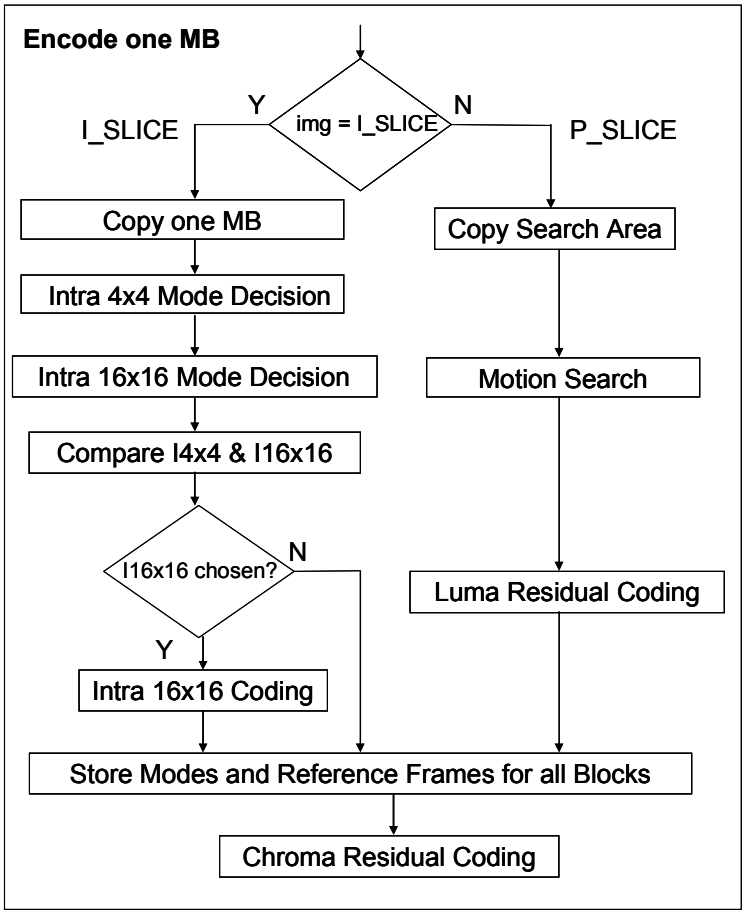


Figure 4.4: Flow graph of encode one macroblock

**Copy One MB:** This function is for copying the original macroblock from the current frame. Instead of accessing the whole frame, I copy only a macroblock and process that macroblock. Beforehand, I was not doing this and accessing the whole frame to read and write macroblock data. However, I observed that this implementation

results in poor performance. If I copy the macroblock, then the data accesses to the cache will result in more number of hits according to the principle of locality. Furthermore, accessing the whole frame data requires complex calculations for addressing.

**Intra 4x4 Mode Decision:** This function will be explained in section 4.1.1.5.

**Intra 16x16 Mode Decision:** This function will be explained in section 4.1.1.6.

**Compare I4x4 and I16x16:** After calculating the costs of both intra modes, one has to make a choice between the two. This is done by simply comparing the cost of intra 4x4 with that of intra 16x16. The 4x4 cost is calculated inside the Intra 4x4 Mode Decision part. This cost is added with "24.λ" before the comparison. The 16x16 cost remains as it is found in the Intra 16x16 Mode Decision part.

**Intra 16x16 Coding:** If the 16x16 mode is chosen then luma residual coding is executed. If intra 4x4 mode is chosen, luma residual coding is not called because luma samples are already coded and reconstructed during the intra 4x4 mode decision. During the intra 4x4 mode decision one has to do the luma residual coding for each 4x4 block because the prediction of the next block depends on the reconstructed block of the previous one. If that reconstructed block is stored at somewhere, there is no need to repeat the luma residual coding. This implementation idea is important in terms of performance.

**Store Modes and Reference Frames:** The selected mode for each 8x8 block is stored in an array for future use. During the motion search the reference frame and mode of the previous blocks and frames are used.

**Chroma Residual Coding:** This routine includes both intra chroma prediction and chroma residual coding.

If the macroblock belongs to an I-SLICE, then 8x8 chroma prediction is done. For each chroma prediction mode, the predicted block is subtracted from the original block and hadamard based SATD is applied o find the cost. The costs of each prediction mode are compared with each other and finally the best chroma intra prediction mode is determined.

If the macroblock belongs to a P-SLICE, chroma motion vectors are extracted from the luma motion vectors. Then chroma motion prediction is done by using those vectors.

After finding the prediction for the chroma, residual is formed. Transform, quantization, vlc and also the inverses of these are applied to the chroma residual in this

function. At the end of the function we get the reordered array for vlc and the reconstructed chroma block.

**Copy Search Area:** For the inter-predicted frames, the motion search is called. The search algorithm searches for the best location over a search window. The size of the search window varies according to the given search step. This function creates a 2D array as the search window and copies data to this array from the reference frame. Instead of accessing the whole reference frame, I copy only a part of it and access only to that portion during the motion search. The dimensions of the search window is as follows:

$$search\_window[16 + 2 \bullet (search\_step)] \cdot [16 + 2 \bullet (search\_step)]$$

In the experiments presented here, the search step is taken as 4 so that a search window of size 24 by 24 is used.

**Motion Search:** This function will be explained in section 4.1.1.7.

**Luma Residual Coding:** After the inter prediction is done and residual is formed, this residual is coded using transform, quant, reorder. Inverse transform, inverse quant processes are also performed and reconstructed macroblock is formed in this function.

#### 4.1.1.5 Intra 4x4 Mode Decision

In the flow graph shown in figure XXX b8 stands for 8x8 blocks and b4 stands for 4x4 blocks inside the 8x8 ones. This means that Intra 4x4 mode decision is called for each 4x4 block of a macroblock.

**Initialize Intra Prediction Buffer:** The intra prediction of a block is done by looking at the neighboring samples of that block. For example, the vertical prediction mode uses the samples above the block. Checking the availability of the neighboring samples and reading the sample values is very time consuming. Because the intra 4x4 mode decision function is called so many times, this part of the software is a critical part. Even a small improvement here may affect the overall system seriously. Therefore, I introduce the prediction buffer which has one more column and row than the macroblock has. That is to say, it is a two dimensional array of size 17 by 17. The required neighboring samples are copied on these extra rows and columns, so that the accesses to them become easier. If the neighboring samples are not available, then the value 128 is written for those samples as is usual. This method results in the same

prediction and fortunately it eliminates the need for repeated checks for the availabilities of neighbors. In other words, it does the same thing but makes it in an efficient and shorter way.

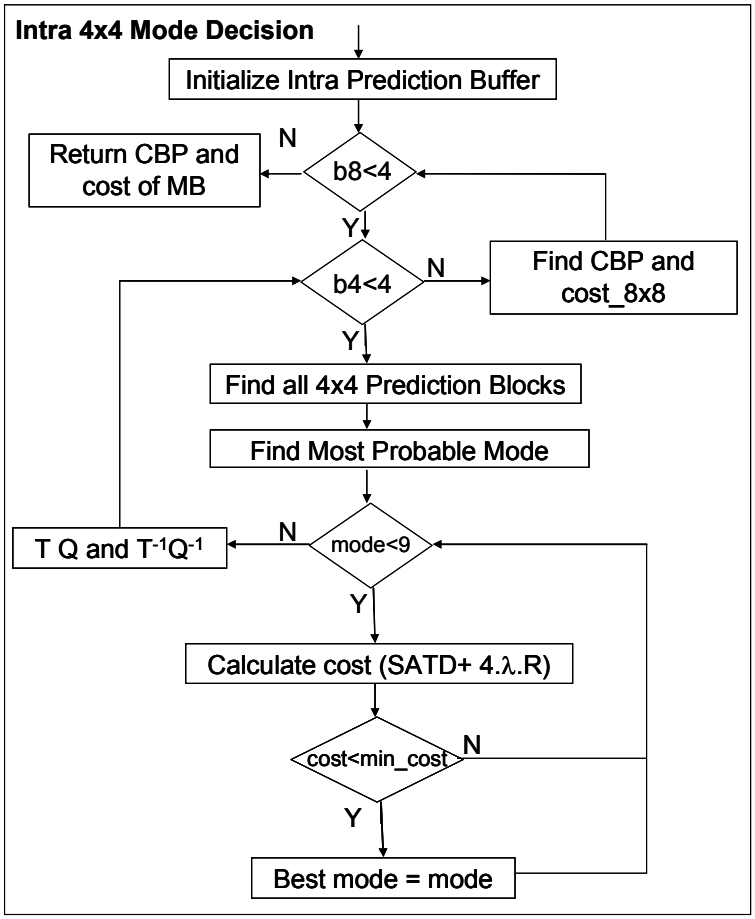


Figure 4.5: Flow graph of intra 4x4 mode decision

**Find all 4x4 Prediction Modes:** Firstly, all 9 prediction modes for each 4x4 block are calculated. Also the most probable mode is predicted from the modes of neighboring blocks. After that, the costs of all 9 modes are compared and the mode with the least cost is assigned as the best mode. The cost calculation is done by using SATD function which uses Hadamard transform. The details of the cost calculation are as follows:

The SATD function takes the residual of the original block and the predicted block.

For each 4x4 residual, hadamard ransform is applied and the absolute values of transformed coefficients are summed up. This total is the SATD value.

SATD is added with 4.λ.R where λ is a constant and R is either 0 or 1. If the current mode is most probable mode R parameter becomes 0, otherwise it is 1. This total is the total cost for that mode.

By comparing the costs of all 9 modes, the mode with the minimum cost is chosen as the best mode for one 4x4 block.

This procedure is repeated for all 4x4 blocks and the minimum costs of each 4x4 blocks are added to find the total minimum cost of the intra4x4 mode. This final cost is going to be compared with the cost of intra16x16 mode.

**T-Q and $T^{-1}$-$Q^{-1}$:** When the comparison of the costs of 9 modes is finished, best mode and the best prediction are determined. The best prediction is used to form the residual. The residual goes through the transform and quantization steps. Also the inverse transform and inverse quantization are applied to reconstruct the block. This reconstructed block is going to be used for the next predictions of neighboring blocks. When the intra 4x4 prediction is performed for all 4x4 blocks and they are reconstructed the macroblock is also reconstructed.

**Find CBP and cost_8x8:** The coded block pattern for each 8x8 block is found in this part. Also the 8x8 costs are calculated by adding up the costs of 4x4 sub-blocks of that 8x8 block.

**Return CBP and Cost of MB:** The cost of intra 4x4 prediction is calculated at the end of this function. This cost is passed to the function which will compare the intra 4x4 with intra 16x16. Also the CBP value is passed because it may be used if intra 4x4 mode is selected for the macroblock.

#### 4.1.1.6 Intra 16x16 Mode Decision

Similar to the intra 4x4 mode decision, intra 16x16 starts with the initialization of the buffers that store the neighboring samples.

**Initialize Prediction Buffer:** In the 16x16 mode, we use the left and up samples. Before making the prediction, two buffers (one buffer for left samples and one buffer for upper samples) are created and initialized with the necessary values. This is for avoiding the repeated availability checking and address calculation for neighboring samples. Neighbor samples are copied to these buffers if they are available. Then we do

not need to calculate the array index of a neighboring sample over the whole frame array.

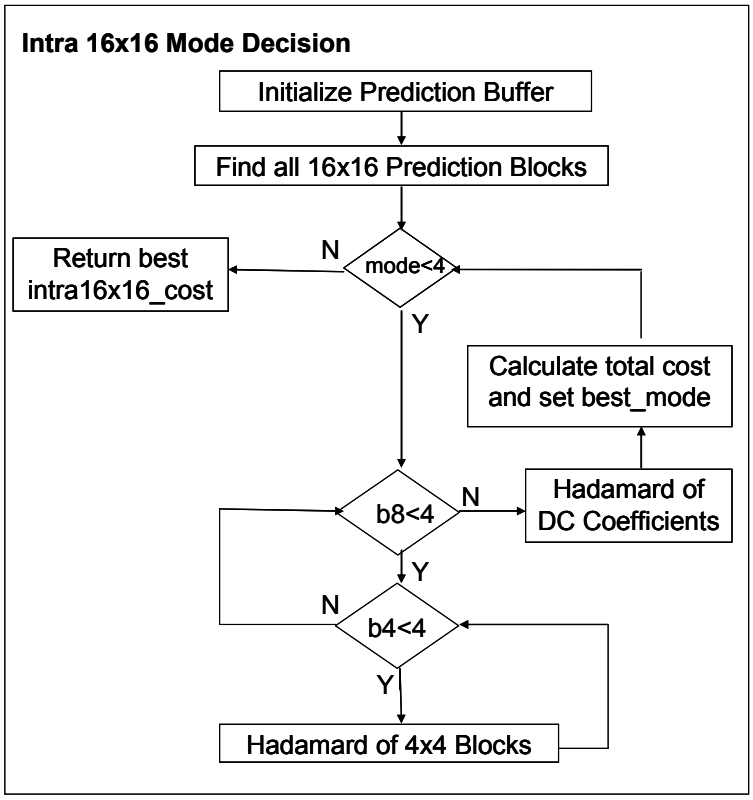


Figure 4.6: Flow graph of intra 16x16 mode decision

After the initialization of the buffers, there is loop which runs for all intra16x16 modes (i.e. 4 times). For each prediction mode, a cost is calculated in the following manner:

1. The macroblock is divided into 4x4 blocks and the hadamard transform is applied to these 4x4 blocks.
2. The DC coefficients of each transformed block is extracted and divided by 2. These coefficients constitute a new 4x4 block.
3. The hadamard transform is applied to this DC block.
4. After all hadamard transformations, the absolute values of all AC and DC coefficients are summed up. This total is the cost of that intra16x16 mode.
5. This procedure is repeated for all modes and the mode with the minimum cost is selected as the best mode. The cost of the best mode is also equal to the cost of the intra16x16 mode.

#### 4.1.1.7 Motion Search

This module runs the motion search algorithm, finds the partition mode and motion vector with the minimum cost. The motion search algorithm is implemented in this function. I implemented two algorithms for this part. First algorithm is three step hierarchical motion search, other algorithm is the full search. I compared the results and calculated performance results for both. Full search algorithm always gives the best result in terms of picture quality. But it requires so many search and therefore computation power. However, the performance of full search is not very low at DM642 DSP platform, because there are fast library functions at Texas Instruments' Image library [15]. These functions are fast enough to catch real time performance at small frame sizes. Hierarchical three step search requires less number of search, but it requires averaging and down sampling operations. At first, I implemented three step search on the embedded DSP platform. During the optimization part, I noticed the library functions of TI's image library which can be used for full search. After that I implemented the full search algorithm. As a result, the implementation of full search is easy and satisfactory in terms of execution time if I use the TI's image library. The detailed performance results will be given in performance analysis section.

Due to time constraint, I am able to implement full search only for the 16x16, 16x8 and 8x16 partitions and not for sub partitions (8x8, 8x4, 4x8, 4x4). However, the other implementation which is three step search supports all partitions. The support for all sub-partitions will be added in the future. But I can say that the performance of the whole system will not be much affected after adding this feature, because the search algorithm is based on SAD reuse. In other words, it does not calculate the SAD for every partition mode but reuse the SAD data that is already calculated. Before checking the inter prediction modes the SAD array is calculated once for all search locations. The motion search for each partition is done by making use of these SAD values. For instance, in order to find the SAD of a 16x16 block the SAD's of four 8x8 blocks are added. The flow graph of the full search algorithm is show in figure 4.7.

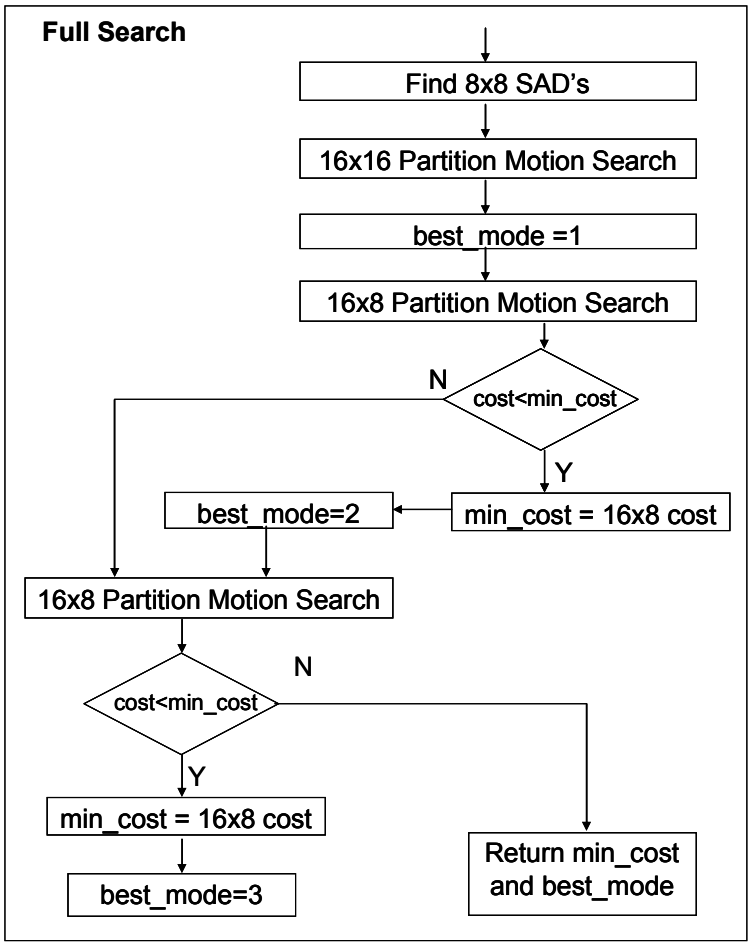


Figure 4.7: Flow graph of full search

**Find 8x8 SAD's:** This function is called once for each macroblock. It calculates 8x8 SAD's for all search locations over a search window. For instance, if the search step size is 4 there are totally 91 ( $(2*step\_size+1)^2$) search locations. The SAD for each 8x8 block at these 91 search locations are calculated one by one and stored in an array for further use.

#### 4.1.1.8 Partition Motion Search

This function is called for each partition and it returns the best motion vector and the minimum cost for that partition. Function starts with finding a prediction for the motion vector (MVP). This prediction is performed by using the motion vector of neighboring blocks. Detailed information about this process can be obtained from from [11].

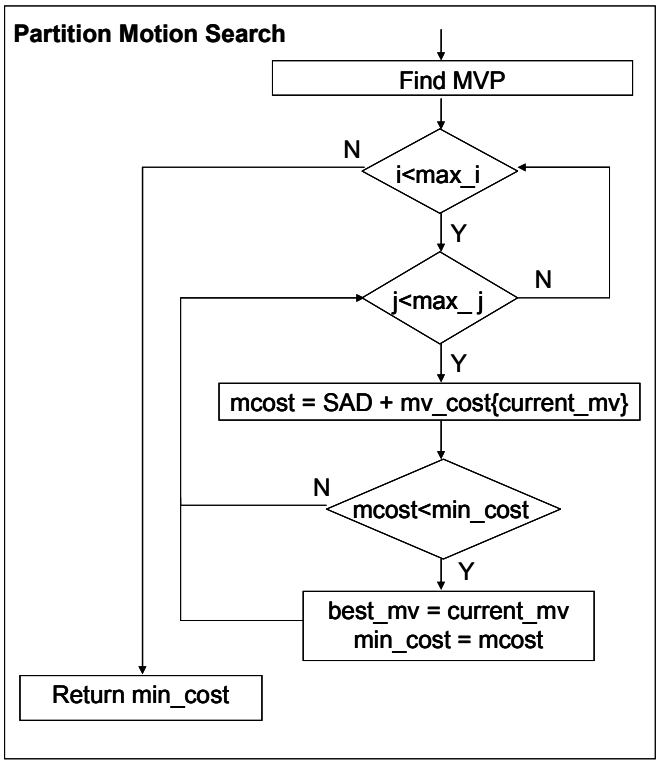


Figure 4.8: Flow graph of partition motion search

In the flow graph (figure 4.8) max_i is the maximum index of the search location in the y-direction and max_j is the maximum index of the search location in the x-direction. For example if search step size is 4, there are 9 locations in the x and y directions and max_i and max_j are equal to 9. That is to say, each loop is executed for 9 times and motion search performed over 81 search locations.

**Calculation of mcost:** mcost is the cost of a partition for a specific search location. At each location, the SAD is added to the motion cost. The motion cost is calculated as follows:

$$mv\_\cos t\{current\_mv\_x, current\_mv\_y\} = mv\_x\_\cos t + mv\_y\_\cos t$$

$$mv\_x\_\cos t = mv\_bits\{current\_mv\_x - pred\_mv\_x\} * lambda\_motion$$
$$mv\_y\_\cos t = mv\_bits\{current\_mv\_y - pred\_mv\_y\} * lambda\_motion$$

Where mv_bits{z} function returns the number of bits to code a motion vector of z. pred_mv_x and pred_mv_y are the predicted motion vectors and lambda_motion parameter is a constant.

After comparing all search locations, the best motion vector for that specific partition is found and the partition motion search function returns this motion vector together with the cost.

## 4.2 DSP Realization

A typical embedded development project starts with the design of the hardware setup and ends with tuning/optimization. Between these two, there are code generation and debugging parts. Code Composer Studio Development environment provides us with many tools for code generation, debugging and tuning. The project development cycle using Code Composer Studio can be summarized with the graph in figure 4.9.

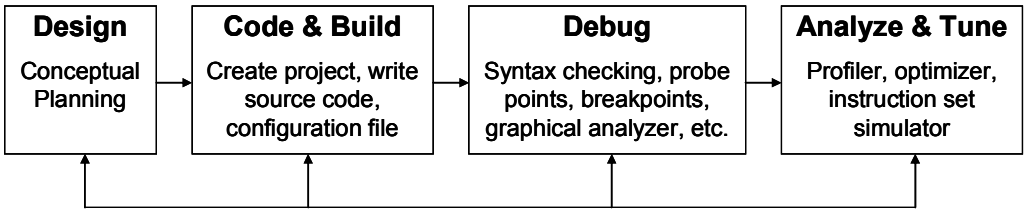


Figure 4.9: A project development cycle using Code Composer Studio

### 4.2.1. Design of Experimental Setup

I tried to simulate a real world embedded application for a H.264 encoder. A real encoder basically takes a video input and outputs a bitstream after compressing the video data. Moreover, the input video can be displayed on a display device for visualization. I formed a hardware setup in order to realize this encoder system.

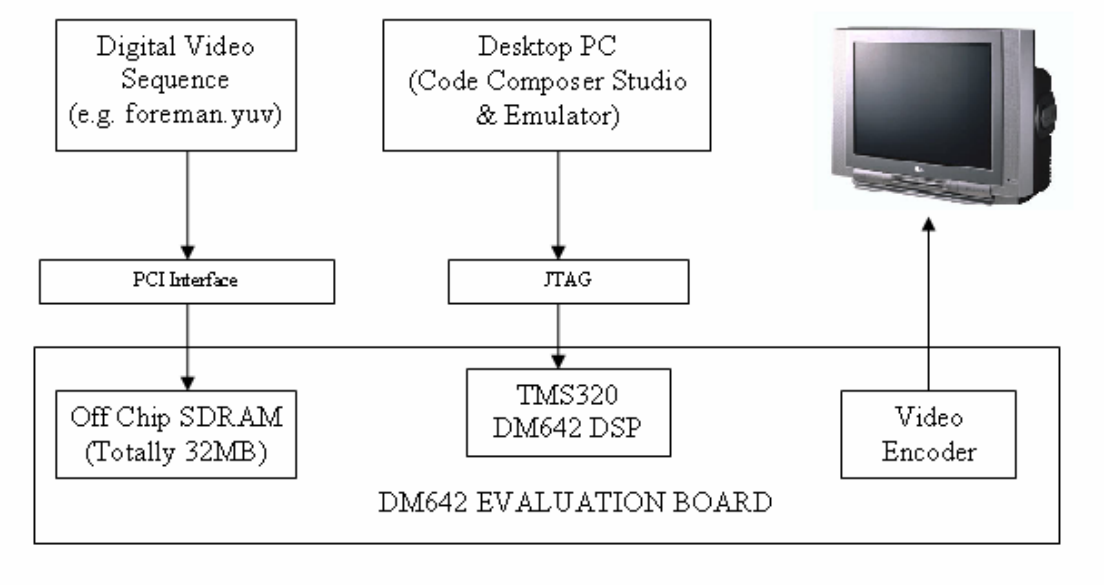


Figure 4.10: The experimental setup

The hardware setup consists of the DM642 Evaluation Board, a desktop computer for Code Composer tool and XDS 560 PCI emulator, a second computer for the transfer of an input video sequence over PCI to SDRAM on the board and finally a television for the display. DM642 board uses a Philips SAA7105 video encoder in order to convert digital video signal to analog video signal.

In a real video application the video data comes packet by packet from a hard-disk, a video capture device or over the ethernet. To simulate this, I copy a small part of a video sequence on the SDRAM of the DM642 EVM using the PCI interface. In this way, there is no need to run file-read operation inside the program. A file I/O operation spend computation power and is also slow because it requires communication between the computer (where the file is stored) and target DSP over the JTAG connection. The proposed implementation in this thesis avoids reading file but writes a file. The file written is the compressed H.264 bitstream file which is the output of the encoder. I put this file-write operation intentionally, because a real encoder must have this ability to store the output.

#### 4.2.1.1 TMS320DM642 Evaluation Module (EVM)

The DM642 EVM is an evaluation board that is designed by the company Spectrum Digital. The EVM is designed to work with TI's Code Composer Studio

development environment. Code Composer communicates with the board through an external JTAG emulator. The EVM board comes with a variety of on board devices that suit many application environments: [3]

1. A Texas Instruments TMS320DM642 DSP operating at 720 MHz.
2. Standalone or standard PCI computer slot operation
3. 3 video ports with 2 on board decoders and 1 on board encoder
4. 32 Mbytes of synchronous DRAM
5. On Screen display (OSD) via FPGA
6. 4 Mbytes of non-volatile Flash memory
7. Ethernet interface
8. Software board configuration through registers implemented in FPGA
9. Configurable boot load options
10. JTAG emulation through on-board external emulator interface
11. Expansion connectors for daughter card use

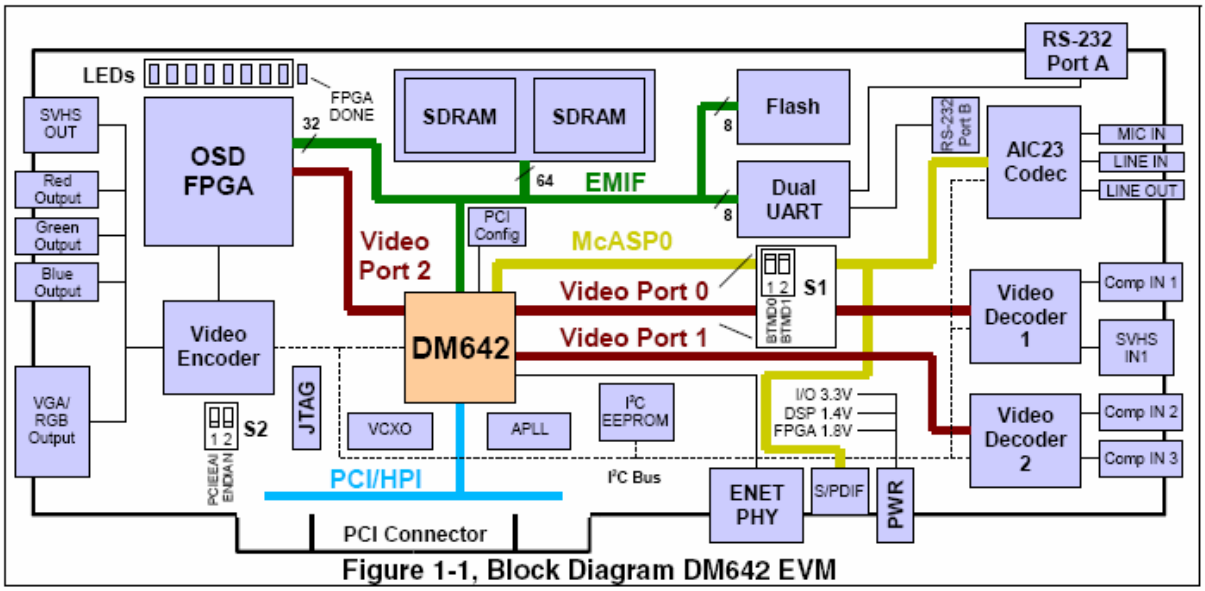


Figure 4.11: Block Diagram DM642 EVM

### 4.2.2 Code Generation using Code Composer Studio

During the embedded code generation, I worked on Code Composer Studio. Code Composer Studio is designed for the Texas Instruments high performance TMS320C6000 digital signal processor (DSP) platforms. It integrates all host and target

tools in a unified environment. It is a development environment that tightly integrates the following components:

- Integrated development environment with editor, debugger, project manager, profiler, probe points, break points.
- C Compiler, assembly optimizer and linker (Code Generation Tools)
- Instruction set simulator
- Real-Time kernel (DSP/BIOS)
- Real-Time data exchange between host and target (RTDX)

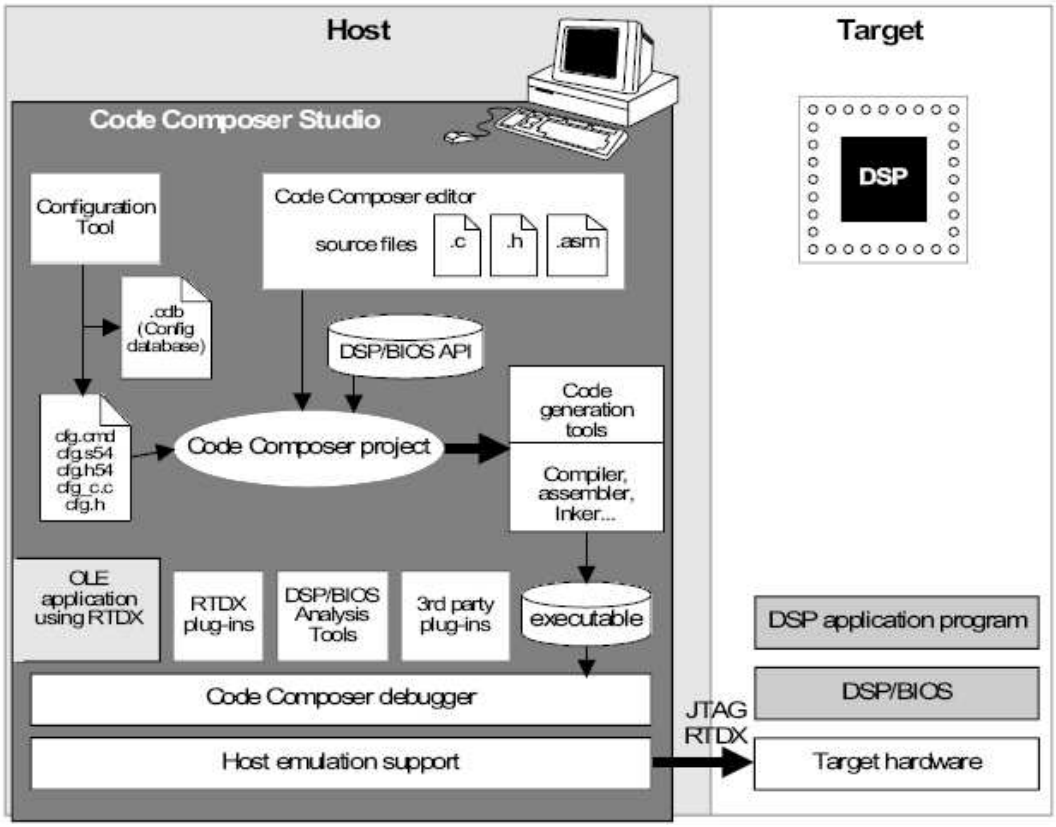


Figure 4.12: Project development on Code Compose Studio

#### 4.2.2.1 DSP/BIOS Real Time Kernel

DSP/BIOS is a real time operating system designed for applications that require real time scheduling and synchronization, host-to-target communication or real time instrumentation. DSP/BIOS provides preemptive multi-threading, hardware abstraction, real-time analysis and configuration tools [24][25].

The threading model provides types of situations. Hardware interrupts, software interrupts, tasks, idle functions are all supported. One can control the priorities and blocking characteristics of threads. Additionally, structures to support communication and synchronization between threads are provided. These include semaphores, mailboxes and resource locks. Using the configuration tool, DSP/BIOS objects can be pre-configured and bound into an executable program image.

To be able to display video on Television using Philips SAA7105 video encoder chip, the device driver for that hardware unit must be included in the DSP/BIOS. Also the tasks in the program must be created with their priorities in DSP/BIOS. The encoder and display tasks are created with equal priorities. The memory space of the target board is also specified in the DSP/BIOS configuration. The target board has 32Mbytes SDRAM. However I configured the DSP/BIOS for 30Mbytes of memory. This is because I download the input video sequence using PCI interface to this 2Mbytes memory area.

#### 4.2.2.2 Sychronized Communication (SCOM) Module

I add the SCOM module to achieve synchronous communication between the encoder task and the display task. SCOM is a module for passing data-related messages among threads. It is a generic inter-task message-passing system. Tasks exchange data among themselves by making use of SCOM messages. A task can pass around an SCOM message by placing it on SCOM queue, or taking it from SCOM queue.

I used the SCOM module in the following way: There are two tasks executing one of which is the encoder task and other is the display task. As the program starts, both tasks are created and started by the DSP/BIOS real time operating system. The display task waits until the picture is ready for display operation. In other words, it waits for an SCOM message from SCOM queue. Whenever the encoder task finishes the reconstruction of a new frame, it sends a SCOM message which includes pointers to the reconstructed frame. After that, display task gets out of the wait state and starts the display operation. When display operation is finished, it sends a message back to encoder task and starts to wait for the next frame. This SCOM usage is crucial because both tasks are accessing the same memory area and without such a semaphore-based module errors would occur.

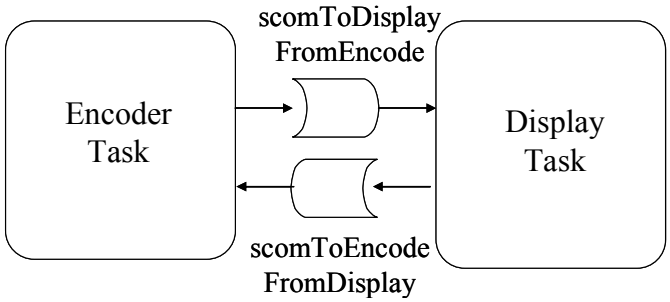


Figure 4.13: Synchronized communication between encoder task and display task

### 4.2.3 Testing and Verification

Testing and verification is one of the most time consuming parts of such a big software project. Code Composer tool provides us with very useful debugging features such as software probe points and graphical analyzers. Because of the real time data exchange feature of TI DSP platforms, one can debug the program without interrupting the program execution. Another tool which is very useful is Elecard Stream Eye tool [16]. This tool takes a H.264 coded bitstream as the input and visualizes the coding parameters such as motion vectors, macroblock partitions, macroblock types, bit count of each frame. In figure 4.14 we see the partitions and motion vectors of each partition of a video sequence encoded by the proposed H.264 encoder. The output of the proposed encoder is also decoded by JM Reference Decoder for verification purpose. The reconstructed frames of JM Decoder is exactly same with the encoder so that it is proven that the proposed encoder is compliant with the standard.

Figure 4.14: Elecard stream eye's output

### 4.2.4 Performance Analysis and Tuning

At the end of the software development, one needs to analyze and tune the program according to the application specifications. Tuning is almost as difficult as the software development. For a H.264 encoder system at least 25-30 fps coding rate is required. Hence, I aimed at developing an encoder with a speed between 25-30fps. For performance measurements, Code Composer Studio has very accurate simulators and analysis tools such as code coverage and exclusive profiler tool. The details of performance analysis and optimization will be described in chapter 5.

# CHAPTER 5

# PERFORMANCE ANALYSIS AND OPTIMIZATION

## 5.1 Test Environment

The performance measurements and profiling are done using the Code Composer 3.1.0 DM642 device simulator. In order to obtain exclusive profile data, the Code Coverage and Exclusive Profiler [17] feature is used. For the inclusive profile data, I setup the CCS profiler for that specific inclusive profile data. The simulator runs on Pentium-4 computer which has 4.2GHz clock speed and 2GBytes of SDRAM. The simulator uses the memory of the computer and this memory size can be changed by modifying the configuration file of the simulator. I assigned 1.6 GBytes memory of the computer to the CCS simulator so that the simulations run faster. The performance analysis and measurements are done using the "news.qcif" QCIF video sequence which is 2 frames long. During the simulations the encoder is set to encode 2 frames, the first frame as I and the second frame as P. The execution time per 1 frame is calculated by dividing the execution time of 2 frames by 2. The quantization parameter of the encoder is fixed at 28, search step size is 4, number of reference frames is 1. Two motion search algorithms are implemented in the proposed encoder. One is hierarchical motion search and other is full search. The performance analysis and optimization belongs to the encoder which implements full search. The clock speed of the processor is 720Mhz. The encoder speed (frames per second) is calculated by dividing the 720MHz by the total cycles consumed for the execution of the program.

$$encoder\_speed(fps) = \frac{Clock\_Speed(720MHz)}{Total\_Cycles}$$

## 5.2 Software Optimization

Software optimization is the process of manipulating software code to achieve faster execution time and smaller code size. Before starting the optimization process we should first decide which parts of the encoder needs optimization and what kind of optimization is needed. Trying to optimize the whole source code is not an efficient way. Instead, we must concentrate on the frequently executed and time consuming parts of the code. The small modifications in important code sections will result in high performance increase whereas modifications in unimportant sections may even not affect the overall performance. As a result, before directly starting with software optimization, one must first profile the software program and make a good analysis for optimizations.

When I first run the program on the DSP, the performance was very poor. Without any optimization and tuning, the initial code can encode only 3.31 frames per second (table 5.1). This performance is very far from our target performance which is 25 fps.

Table 5.1 Performance of the un-optimized encoder.

| Type of Frame | Total Cycles | Speed(fps) |
|---|---|---|
| Coding an I-Frame | $1.75 \times 10^8$ | 4.11 |
| Coding a P-Frame | $2.65 \times 10^8$ | 2.71 |
| Average Coding for one frame | $2.17 \times 10^8$ | 3.31 |

Also it is found out that the write_one_macroblock function has little effect on the overall performance. Because the dominant function is the encode_one_macroblock function, the optimization operations should focus on this function. The execution times of encoding and writing functions are shown in table 5.2.

Table 5.2: Comparison of encode_one_macroblock with write_one_macroblock

| Sub-functions Inside “code a picture” Function | Percentage |
|---|---|
| encode_one_macroblock | 96% |
| write_one_macroblock | 3% |
| Others | 1% |

The optimization process that is applied in this thesis can be divided into two:

- Applying optimization techniques before improving the algorithm and memory access pattern.
- Applying optimization techniques after improving the algorithm and memory access pattern.

### 5.2.1 Optimization without Algorithm/Memory Optimizations

After getting the initial performance results and the profile data, I directly started to apply software optimization techniques. In this approach, only software optimizations are applied but algorithm or memory access optimizations are not any considered. The applied software optimization techniques are:

1. Using functions from TI’s Image and DSP libraries.
2. Using compiler intrinsics.
3. Utilizing optimizing compiler.
4. Partitioning the on-chip memory into different L2 Ram/Cache sizes and allocating program data on the on-chip memory.

The details of these optimization techniques are not going to be discussed here but a more detailed explanation about these techniques will be given in the next section. The important thing in this optimization approach is that it could not achieve the real-time performance. The total execution cycle count is about $6.92 \times 10^7$ so the encoder speed is about 10.4 frames per second. Even though these optimizations are very useful, the performance is still beyond the desired value of 25fps.

Table 5.3: Performance increase with software optimizations only

| | Total Cycles | Speed(fps) |
|---|---|---|
| Before Optimization | $2{,}17 \times 10^8$ | 3.31 |
| After Optimization | $6{,}92 \times 10^7$ | 10.4 |

Table 5.4: Number of NOPs and CPU stalls after applying software optimizations

| NOPs | CPU Stall Cycles Due to Memory | Total Cycles |
|---|---|---|
| 21.280.882 | 11.330.520 | 69.200.866 |

In order to find out the reason, a further analysis on the program is performed. The results have shown that the number of NOPs and CPU stalls are very high with respect to the total cycles. Almost the half of the total cycles is spent by NOPs and CPU stalls due to memory accesses. It is obvious that CPU stalls because of cache misses. When a cache miss occurs the missed data is read from or written to the upper memory level. The CPU has to wait until this data transfer between memory levels is finished. The characteristic of the H.264 encoder is that there are many memory accesses inside the program. These memory accesses cause CPU to stall. When I investigated why there are many NOPs I found out that NOPs are also due to the memory accesses. Because of the high number of load and store instructions, other instructions can not be scheduled efficiently. There are operations which use the data that is loaded by the load operation. Such operations have to wait for the completion of the load operation. Therefore NOPs are introduced between load operations and those operations that use the loaded data. The same is also true for store operations. As a result, there are many NOPs due to the high number of loads and stores.

It is remarkable that after software optimizations the total cycle value is decreased to $6.92 \times 10^7$ but the percentage of NOPs and stalls with respect to the total cycles becomes very high. Nearly half of the execution time is consumed by NOPs and CPU stalls due to memory. In other words, after the optimization memory accesses become the bottleneck and limit the performance. Even though we applied very useful optimization techniques, without improving the memory accesses we always get a limited performance. This result has shown that before applying software optimization

one has to first improve the memory access pattern. Therefore I will go to the next step which studies optimizations with memory optimizations.

### 5.2.2 Optimization with Algorithm/Memory Optimizations

The results obtained in the previous section have shown that we have to first optimize the memory accesses and then apply the software optimizations. Memory optimization can be achieved by either modifying the algorithm in order to reduce the number of memory accesses or by creating buffers to obtain fast memory accesses. Modifications in the algorithm can reduce the number of loads and stores but is can also eliminate the redundant functions that cause memory accesses. Creating buffers obviously increases the cache hit rate, but it can also reduce the number of computations necessary for memory addressing. In this section, the optimization starts with memory optimizations and software optimizations similar to the previous section are applied afterwards.

The whole optimization process can be divided into six steps:

1. First of all, I simulated different L2 ram/cache partitioning and chose the best one.
2. Secondly, the memory access pattern is improved. This is achieved by the modifications in the algorithm and the creation of buffers for data storage and reuse. This approach also simplified the address calculation mechanism.
3. Third, the allocation of compiler output sections are analyzed and the best allocation is determined.
4. Fourth, code optimizations such as replacing some parts of the software with library functions, using intrinsics, function inlining and changing variable types are investigated.
5. Fifth, the optimizing compiler is utilized with different optimization options.
6. Lastly, the frequently accessed data arrays such as macroblock array or search window array are allocated on the on-chip memory.

The order of these optimizations is important since each method affects the results of other methods. These optimization steps are going to be discussed in the following section:

#### 5.2.2.1 L2 Cache / Ram Partitioning

First of all, the memory configuration of the system should be designed. All other simulations will run on this configuration. The DM642 DSP is composed of a ram/cache flexible on-chip memory of 256Kbytes. 256 Kbytes of on-chip memory can be partitioned either as L2 cache or as internal memory ISRAM. The L2 cache is a 4 way set associative cache and its size can be configured as 0, 32, 64, 128 and 256 Kbytes. In the DSP/BIOS configuration file, the DSP's memory model is configured. In order to obtain the best configuration, I profile the program with different L2 ram/cache partitioning. The summary of these profiling is shown in table 5.5. These results are obtained for the full execution of the program.

Table 5.5: Simulation results for different ram/cache partitioning

| Parameter | L2 Cache size (4 way set associative) | | |
|---|---|---|---|
| | 32Kbytes | 128Kbytes | 256KBytes |
| L2 cache access (Total) | 2.219.732 | 2.220.257 | 2.219.775 |
| L2 cache hit summary (Total) | 2.078.786 | 2.079.203 | 2.078.821 |

First of all, the existence of L2 cache is important because the number of L2 cache accesses is very high. This is due to the fact that the number of loads and stores in a typical video processing application is very high. Without L2 cache we have to pay for larger miss penalty for L1D and L1P misses. Whenever a L1D or L1P cache miss occurs the data requested will be transferred from external memory. The access time of the external memory is obviously larger than L2 cache so that the performance of the system is degraded much. Therefore using a two level cache hierarchy seems better than just using L1 cache. Secondly, results show that the values are very close to each other. This means that all three configurations give nearly the same performance. Since all three results are close to each other, choosing the smallest L2 size of 32Kbytes does not degrade the performance. If we choose the smallest L2 cache size then we have larger

space for the internal memory (ISRAM). As a result, I decide that L2 cache with a size of 32KBytes is optimal for this system. The remaining part of the internal memory which is 224Kbytes is mapped as ISRAM. By creating such a ISRAM memory, we gain the flexibility to allocate some critical code or data sections on the on-chip memory. All following simulations are done based on this cache/ram configuration. (i.e. 32Kbytes L2 cache and 224Kbytes ISRAM).

#### 5.2.2.2 Improvements in Memory Access Pattern and Encoder Algorithm

Time consuming and memory accessing parts of the program are determined and improved. Mainly, macroblock reading/writing functions, intra 4x4, intra 16x16 and motion search sections are improved.

##### 5.2.2.2.1 Buffering Macroblock Data

In the initial program, the macroblock data and neighboring samples were all read from the array that stores the whole frame. Since the frame array is large, we cannot store it on the on-chip memory. Therefore accesses to the frame may result in L1 and L2 data misses. Moreover, reading from the whole frame makes the address calculation very complex. In order to calculate the address of a neighboring sample, one has to first calculate the current macroblock's address and then add an offset in order to reach the neighbor sample. As a result, reading the samples from the whole frame is not efficient. Instead, those samples can be copied into a buffer which is stored on internal memory. This improves the performance in two ways:

- Address calculation becomes much easier.
- Because of locality, the cache misses will decrease.

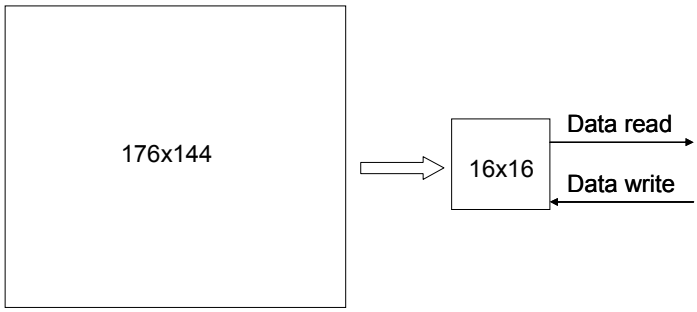


Figure 5.1: MB Data read/write is performed on the 16x16 macroblock buffer.

Copying the macroblock data into a smaller array requires extra operations. However, we copy the MB only for once at the beginning of the encode_one_MB function and we copy back the reconstructed MB to the frame array at the end of same function.

#### 5.2.2.2.2 Improvements for Intra 4x4 Prediction

Intra 4x4 prediction algorithm uses the samples that are neighbors of the 4x4 block. The samples that are at the top, left and up-right of a macroblock are used during the intra4x4 prediction. The first version of the software was searching for each of these samples over the frame. It was first checking the availability of that sample. If it is available then the address of that sample was being calculated. In the new version, an array of size 17x17 is created and neighbor samples are copied to the first row and column. For the up-right neighbors another array of size 1x4 is created. In case of non-availability of a sample, the value 128 is copied to the array. After that intra4x4 prediction and reconstruction is started. When a 4x4 block is reconstructed it is also copied into this 17x17 array so that the following predictions for neighbor blocks can make use of these reconstructed samples. This method not only improves the memory access but also it decreases the number of operations. In this method, the repeated checks for the availability of the samples are eliminated. Also the address calculation for all samples is very easy. It gives exactly the same result with the previous implementation, but the performance is obviously better.

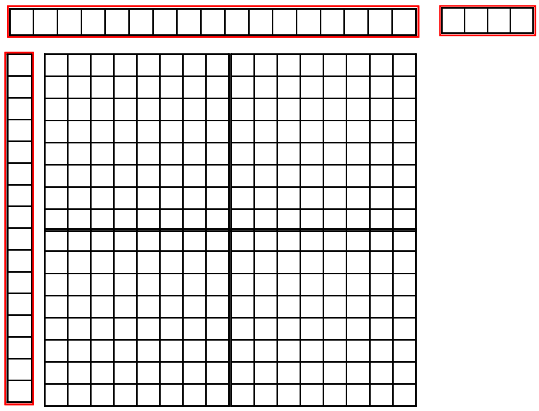

Figure 5.2: Buffers for intra 4x4 prediction

#### 5.2.2.2.3 Improvements for Intra 16x16 Prediction

Similar to the intra 4x4 prediction, intra 16x16 prediction also uses the neighbor samples of the macroblock. The same idea therefore can be applied to intra 16x16 prediction. However, it is noticeable that we do not need the macroblock array itself since we have already created a buffer for the macroblock. What we need for the intra 16x16 prediction is only the upper and left samples. Therefore, we only need two arrays of size 1x16.

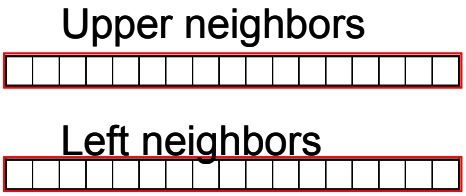


Figure 5.3: Buffer for intra 16x16 prediction

#### 5.2.2.2.4 Improvements for Motion Search

For the motion search two algorithms are implemented. One of them is full search and other three step hierarchical full search. Both of them are based on calculation of SAD between the current macroblock partition and the reference frame. In the initial software implementation, the SAD was being calculated by reading the reference blocks directly from the reference frame. This method is not efficient since it increases the cache misses and makes the address calculation complex. Therefore, before SAD

calculation a 2D array is created as the search window. The size of this array depends on the search_size. The samples of the search window are copied from the reference frame. Afterwards, the SAD calculation between the macroblock partition and search window is performed. It is important to say that the algorithm is based on SAD reuse. In other words, the SADs of larger blocks are found by adding up the SADs of small blocks. SAD reuse method eliminates unnecessary SAD calculations.

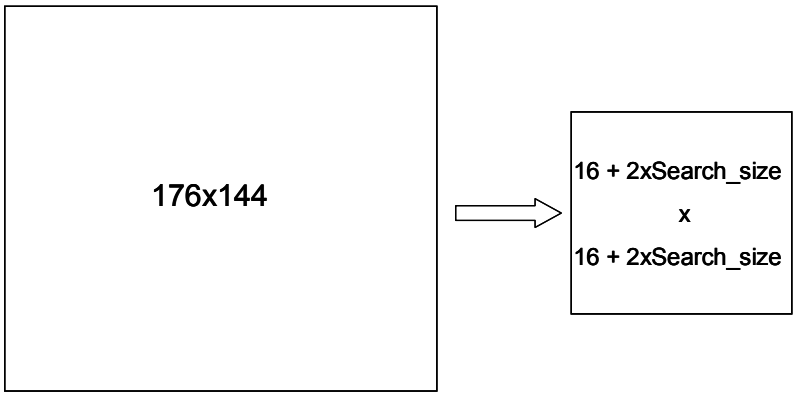


Figure 5.4: Search window array is created for motion search

Table 5.6: Performance increase with algorithm/memory access improvements only

| | Total Cycles | Speed(fps) |
|---|---|---|
| Before Memory Optimizations | $2.17 \times 10^8$ | 3.31 |
| After Memory Optimizations only | $1.94 \times 10^7$ | 3.71 |

The introduction of these improvements in memory access pattern increases the performance. After these improvements, code optimizations and compiler optimizations can achieve better results because the address generation becomes simple. Also the memory loads and stores can execute in parallel since the memory accesses are organized in a better way.

#### 5.2.2.3 Allocation of Compiler Output Sections

The compiler creates output sections after the compilation operation. The allocation of these output sections in memory is important in terms of performance. The

sections that are frequently accessed should be allocated in fast memory. For example, ".text" section is the program code and should be allocated in fast on-chip memory (ISRAM) if possible. On the other hand, table of constructors to be called at start-up is necessary only at the beginning of the program and is better allocated in slow off-chip memory (SDRAM).

Table 5.7: Output sections of compiler

| Name | Contents |
|---|---|
| .cinit | Tables for explicitly initialized global and static variables |
| .const | Global and static const variables that are explicitly initialized and contain string literals |
| .pinit | Table of constructors to be called at start-up |
| .switch | Jump tables for large switch statements |
| .text | Executable code and constraints |
| .bss | Global and static variables |
| .far | Global and static variables declared far |
| .stack | Stack |
| .sysmem | Memory for malloc functions (heap) |

I have tried to allocate executable code (i.e. ".text" section) in SRAM. At first trial, it does not fit into the SRAM because of its large size. Initially, when I transferred the source code from desktop computer to the embedded platform it was very long and inefficient. Hence the executable code was also large. But allocating program memory on the on-chip memory is very important because the processor will read each instruction from the program memory. If we allocate it on the off-chip memory the CPU will access the external memory at each instruction execution. In order to reduce the executable code I modified the source code much. I eliminated all redundant functions, redundant variables and routines. At the end I was able to fit the program code into the on-chip memory (ISRAM). Other output sections of the compiler such as ".switch", ".bss", ".const", ".data" and ".cio" are all allocated in external memory (SDRAM).

#### 5.2.2.4 Code Optimizations

During the software development phase, I considered that this piece of software will run on embedded platform and wrote the code accordingly. For instance, I avoided dynamic memory allocations because memory operations in embedded platform slow down the system. Especially, memory allocations on the off-chip memory takes much time and should be avoided at time critical applications. Additionally, I declared some variables as global in order to avoid excessive use of parameter passing. I also tried to limit the number of functions as far as possible in order to decrease the number of function calls.

Static variables are created at the beginning of a program and remains until the end. If constant variables such as matrix elements are declared as static they are initialized with first values at the start-up and remains forever. If we use static global variables we eliminate recreation and re-initialization of constant variables at each function call.

Time consuming code sections can be further analyzed using the mixed source/assembly view feature of CCS IDE. The assembly instructions that are counterpart of C source code are shown in this view. With the help of this feature one can decide upon what kind of optimization is suitable for a critical code section. In figure 5.5 both C and assembly code of the SATD critical code section is shown. The instructions that are tied with the pipe symbols "| |" are executed in the same execution packet. Up to 8 instructions can be found in an execution packet since there are 8 functional units in the DM642 architecture. From figure 5.5 it is seen that the biggest execution packet includes only two instructions. This tells that this part of the code is not parallelized enough. Moreover the assembly code includes NOPs.

```
/*===== hadamard transform =====*/
m[ 0] = d[ 0] + d[12];
81306934 019018F0            OR.D1X      0,B4,A3
81306938 018C0265            LDW.D1T1    *+A3[0x0],A3
8130693C 021016A0  ||        OR.S1X      0,B4,A4
81306940 02118266            LDW.D1T2    *+A4[0xC],B4
81306944 00006000            NOP         4
81306948 020C9AB2            ADD.D2X     B4,A3,B4
8130694C 023CC2F6            STW.D2T2    B4,*+SP[0x6]
81306950 00002000            NOP         2
m[ 4] = d[ 4] + d[ 8];
81306954 019008F1            OR.D1       0,A4,A3
81306958 021018F2  ||        OR.D2X      0,A4,B4
8130695C 018C8265            LDW.D1T1    *+A3[0x4],A3
81306960 021102E6  ||        LDW.D2T2    *+B4[0x8],B4
81306964 00006000            NOP         4
81306968 020C9AB2            ADD.D2X     B4,A3,B4
8130696C 023D42F6            STW.D2T2    B4,*+SP[0xA]
81306970 00002000            NOP         2
```

Figure 5.5: Mixed Source/Assembly view of the function SATD

The critical parts of the code must be parallelized as much as possible, so that we can benefit from the dsp's parallel processing capability and obtain high enough performance. As we analyze the critical functions of the encoder, we see that four type of code optimizations are suitable for these critical code sections of the encoder software. These optimizations are "library functions", "intrinsics", "function inlining" and "type conversion of variables".

#### 5.2.2.4.1 Fast Library Functions

Texas Instruments libraries include C-callable functions (ANSI-C language compatible) for general-purpose imaging functions that include compression, video processing or general mathematic operations. DSP library and Image library are such useful libraries for video and imaging applications and they are optimized for TI's C6000 DSP architecture. In the H.264 encoder algorithm there are a lot of matrix multiplication operations. The integer transform, hadamard transform and hadamard based SATD calculation are examples of functions that use matrix operations. These can be replaced with a matrix multiplication function from TI's DSP library[18], called "DSP_mat_mul".

**void DSP_mat_mul(short *x, int r1, int c1, short *y, int c2, short *r, int qs):**

- x [r1*c1]: Pointer to input matrix of size r1*c1.
- r1: Number of rows in matrix x.
- c1: Number of columns in matrix x. Also number of rows in y.

– y [c1*c2]: Pointer to input matrix of size c1*c2.
– c2: Number of columns in matrix y.
– r [r1*c2]: Pointer to output matrix of size r1*c2.
– qs: Final right–shift to apply to the result.

```
/*= hadamard transform =*/
m[ 0] = diff[ 0] + diff[12];
m[ 4] = diff[ 4] + diff[ 8];
...
...
...
diff[14] = m[14] + m[15];
diff[15] = m[15]  - m[14];
```

```
// had is the hadamard matrix

// res is the result matrix

short res[16];

DSP_mat_mul(diff,4,4,had,4,res,0);
```

Figure 5.6: Original code and its transformation into library function

The sad calculation function from TI's Image library is also very usefull for the motion search. This function return the SAD between a reference frame and a original block.

**IMG_sad_8x8(unsigned char *srclmg, unsigned char *reflmg, int pitch):**

– srcImg[64]: 8x8 source block. Must be double-word aligned.
– refImg[]: Reference image.
– pitch: Width of reference image.

```
for(i=0; i<8; i++)
  for(j=0; j<8; j++)
    sad +=abs(org[i][j]-ref[i][j]);
```

```
sad=IMG_sad_8x8(org, ref,size);
```

Figure 5.7: Original code and its transformation into library function

The source block "org" must be double word aligned. If it is not aligned to double word then the library function IMG_sad_8x8 does not function correctly. Data alignment can be done by using the pragma directive "DATA_ALIGN".

```
//double word alignment of orgEE array
#pragma DATA_ALIGN(org,sizeof(long));
unsigned char org[64];
```

Figure 5.8 The array is aligned to double word boundary

In this section, the whole source code is analyzed and all of the matrix operations are replaced with the library function DSP_mat_mul and sad calculation is replaced with IMG_sad_8x8.

#### 5.2.2.4.2 Compiler Intrinsics

The C64x compiler provides intrinsics, special functions that map directly to inlined C64x instructions to optimize C code quickly[19][20]. Intrinsics are specified with a leading underscore ( _ ) and are accessed by calling them as you call a function inside the C source code. Intrinsics allows us the use of C variables instead of hardware registers and also schedules the instructions to maximize performance.

```
for(k=0; k<16; k++)
{
  satd += (res[k] < 0 ? –res[k] : res[k]);
}
```

```
for(k=0; k<16 ; k++)
{
  satd += _abs(res[k]);
}
```

Figure 5.9: Original code and its transformation by using intrinsics

```
x = min(a,b);
y = max(a,b);
```

```
x = _min2(a,b);
y = _max2(a,b);
```

Figure 5.10: Original code and its transformation by using intrinsics

#### 5.2.2.4.3 Function Inlining

The function calls may decrease the performance since each function call requires passing some parameters between functions. If a function is called so many times then many cycles may be consumed during these parameter passing operations. For instance, the function called “sign” in the proposed encoder is such a function. It is called inside the transform-quant routines for great many times. Therefore even a small gain in cycle count may improve the overall performance [5]. The “sign” function is composed of

only an “if” loop and an absolute value calculation routine. Since it is a small function, inlining that function may not increase the code size much. As a result, “sign” function is inlined so that the calls to the function are eliminated.

#### 5.2.2.4.4 Changing Variable Types

In software project developers usually use integer variables. However, in an embedded software project the type of variables used may affect the performance. On the TMS320 platform, integer type is 4 bytes, short is 2 bytes and char is 1 one byte. Using short instead of integer may speed up the program execution because load/store of 2 byte variables is faster than load/store of 4 bytes. If the execution of the program is not affected with such type changes then we should better replace large variables with smaller ones. The change of “int” variables to “short” improves the performance because read/write operations on short variables are faster. When the read/write operations becomes faster, the CPU stalls due to memory are also decreased.

In the proposed encoder the integer data arrays are replaced with short arrays if possible. Especially, this conversion should be done for frequently used data array. Some examples of the frequently used arrays in the proposed encoder are orgSS(the original macroblock), recSS(the reconstructed macroblock), predFF(prediction for a 4x4 block), diffF(4x4 residual). All these arrays’ types are converted from integer to short. The simulation results show that this conversion is very useful in terms of performance. Therefore the conversion is done at several part of the source code.

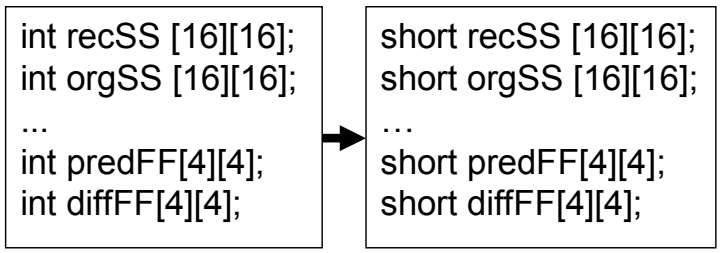


Figure 5.11: The integer variables are replaced with short variables

#### 5.2.2.5 Utilizing Compiler Options for Optimization

The C64x optimizing C compiler can perform optimization currently up-to about 80% compared with a hand-scheduled assembly [9]. If someone has the knowledge of all optimization methods and makes a good analysis on the program, achieving this level of optimization is possible.

The optimizing compiler can be invoked from the CCS screen with many optimizing options. Table 5.8 explains some of the important compiler options for optimizations. Among all these options, I found out that –o3 (file level optimization), -mt(no bad memory alias occurs) options are useful for the proposed encoder. I tried almost all options during the optimization phase, but most of them did not affect the performance. Therefore I mention only the three options that increase the performance.

Table 5.8: Compiler Options for Higher Performance

| Option | Description |
|---|---|
| -mh | Allows speculative execution. But the appropriate amount of padding must be available in data memory to insure correct execution. |
| -o3 | Represents the highest level of optimization available Various loop optimizations are performed, such as software pipelining, unrolling, and SIMD. |
| -pm | Combines source files to perform program-level optimization |
| -mii | Describes the interrupt threshold to the compiler. If you know that NO interrupts will occur in your code, the compiler can avoid enabling and disabling interrupts before and after software pipelined loops for a code size and performance improvement. In addition, there is potential for performance imprompment where interrupt registers may be utilized in high register pressure loops. |
| -mt | Enables the compiler to use assumptions that allow it to be more aggressive with certain optimizations. For example it assumes that no memory ambiguation will occur and makes optimizations accordingly. However, if this assumption is wrong the program will not function properly. |

##### 5.2.2.5.1 File-Level Optimization ( −o3 Option)

The −o3 option instructs the compiler to perform file-level optimization [20]. This option can be used alone to perform general file-level optimization, it can be combined with other options to perform more specific optimizations. Various kind of optimizations are performed with this compiler option. For instance, software pipelining

is implemented by this compiler option.[22] Software pipelining creates highly-optimized loop-code by:

- Putting several instructions in parallel.
- Filling delay slots with useful code.
- Maximizes functional units.

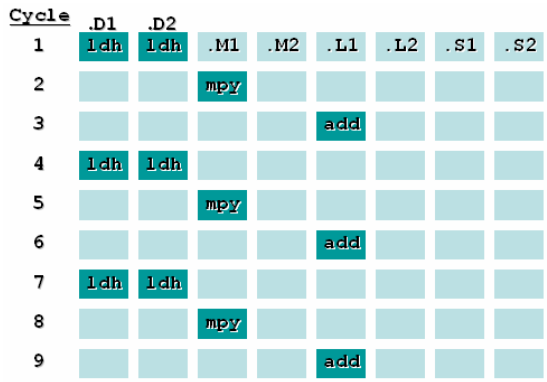


Figure 5.12: Before software pipelining

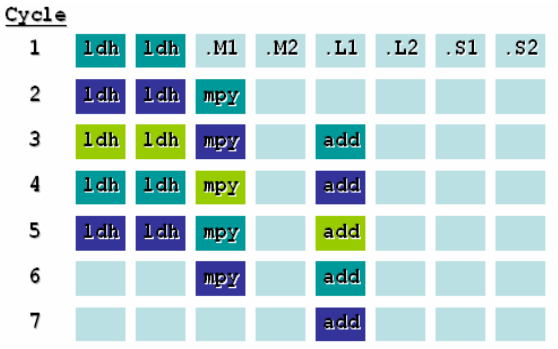


Figure 5.13: After software pipelining

#### 5.2.2.5.2 Assuming No Bad Memory Alias Occurs (–mt option)

This option allows the compiler to use assumptions that can eliminate memory dependency paths. To maximize the efficiency of the code, the C64x compiler schedules as many instructions as possible in parallel. To schedule instructions in parallel, the compiler must determine the relationships, or dependencies between instructions. Because only independent instructions can execute in parallel, dependencies inhibit parallelism.

- If the compiler can not determine that two instructions are independent, it assumes a dependency and schedules the two instructions sequentially.
- If the compiler can determine that two instructions are independent of one another, it can schedule them in parallel.

To analyze the memory dependencies, the C code and its dependency graph for a basic vector sum is given in figures 5.14 and 5.15 respectively.

```
void vecsum( short *sum, short *in1, short *in2, unsigned int N)
{
    int i;
    for(i=0 ; i < N ; i++)
    {
            sum[i] = in1[i] + in2[i];
    }
}
```

Figure 5.14: A basic vector sum function

The dependency graph of this source code (figure 5.15) says that:

- The paths from sum[i] back to in1[i] and in2[i] indicate that writing to sum may have affect on the memory pointed to by either in1 or in2.
- A read from in1 or in2 cannot begin until the write to sum finishes, which creates an aliasing problem. Aliasing occurs when two pointers can point to the same location.

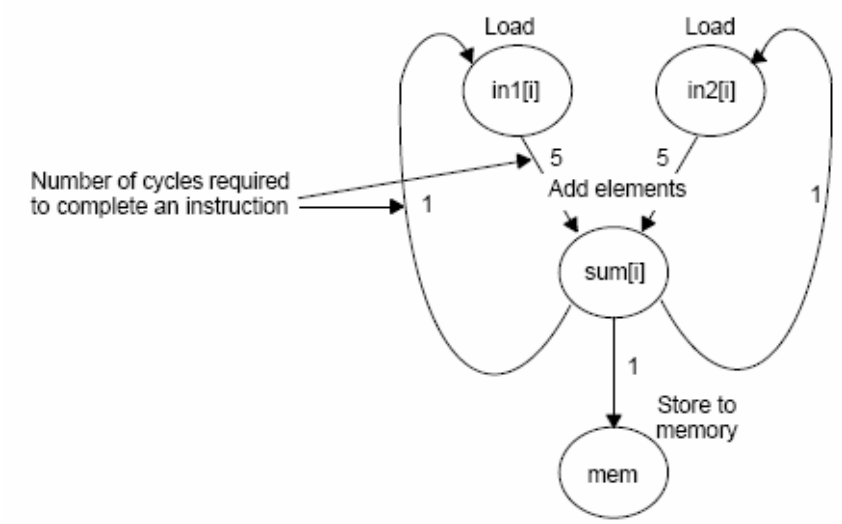


Figure 5.15: Dependency graph of basic vector sum.

If –mt option is used, then the compiler uses the assumption that in1 and in2 do not alias memory pointed to by sum. Therefore it eliminates memory dependencies among the instructions. However, if your code does not satisfy this assumption, you can get incorrect results.

#### 5.2.2.6 Allocating Frequently Used Data in the Internal Memory

The pragma directive “DATA_SECTION” allocates space for a symbol in the memory section named “mysection”. This directive can be used for allocating frequently used symbols such as macroblock data in the fast memory partition. After allocating such symbols in the on-chip memory (SRAM), an access request to that data results in a L2 hit. In this way, the program does not have to read this data from external memory.

```
#pragma DATA_SECTION(recSS,".mysect");
short recSS [16][16];

#pragma DATA_SECTION(orgSS,".mysect");
short orgSS [16][16];

#pragma DATA_SECTION(orgFFA,".mysect");
short orgFFA[16][4][4];

#pragma DATA_SECTION(predFF,".mysect");
short predFF[9][4][4];

#pragma DATA_SECTION(diffFF,".mysect");
short diffFF[4][4];

#pragma DATA_SECTION(all_pred_mv,".mysect");
char all_pred_mv[9][36][44][2];

#pragma DATA_SECTION(all_mv,".mysect");
char all_mv[9][36][44][2];

#pragma DATA_SECTION(sadEEA,".mysect");
short sadEEA[2][2][SRCH_9][SRCH_9];

#pragma DATA_SECTION(spadFEFE,".mysect");
unsigned char spadFEFE[SRCH_24][SRCH_24];
```

Figure 5.16: Allocation of frequently accessed arrays to internal memory section

The proposed H.264 encoder makes use of global variables that are used several times during the execution of the program. For instance, macroblock data, prediction data and residual data are frequently used parts of the source code. When I applied this allocation method to such critical data arrays, I obtained a slight increase in the performance. This increase is not very satisfactory in terms of overall performance.

As a result, using memory space in an efficient and effective way in video encoder design is crucial. Allocating the frequently accessed data on the on chip

memory increases the data read/write hits. Therefore, the CPU does not stall because of memory read/write operations.

## 5.3 Summary of Software Optimization

Starting with the un-optimized program which can process only 3.31 frames per second, I obtained an optimized encoder which can process 26.7 frames per second. As it is stated at the beginning of this chapter, these values are measured for the news.qcif video sequence which is 2 frames long. The first frame is coded as I-Frame and second one as P-Frame. The total cycle count to process 1 frame is calculated by averaging the cycle count for 2-frames. The speed of the encoder (in fps units) is calculated using the clock speed of the DSP (i.e.720 MHz).

Table 5.9: Total CPU cycle counts according to the performed optimization.

| **STEPS** | **1 frame (average)** | **Speed (fps)** |
|---|---|---|
| **1-)** Unoptimized Code | 217.914.368 | 3.31 |
| **2-)** L2 Ram/Cache Partitioning | 217.904.555 | 3.31 |
| **3-)** Improvement in Memory Access Pattern | 194.314.544 | 3.71 |
| **4-)** Code Optimizations(library functions, intrinsics, function inlining) and Compiler Optimizations | 29.679.631 | 24.25 |
| **5-)** Change of variables' types (integer to short) | 27.365.153 | 26.31 |
| **6-)** Frequently accessed data arrays are allocated in internal memory | 27.282.428 | 26.40 |
| **Average speed of the final encoder over 20 frames** | | **26.70** |

The unoptimized code performs very poor but it is proved that optimization on this code is possible. Several kinds of optimizations can be applied on the algorithm but the most important ones for this program are the Improvements in Memory Access Pattern, Code Optimizations and Compiler Optimizations. Memory Access Pattern refinement decreases the number memory accesses and simplifies the memory address

generation. In this way, it reduces the number of CPU stalls due to memory. Before implementing the memory access pattern refinement, code optimizations and compiler optimization can not increase the performance much because those optimizations also depend on memory accesses. Software optimizations alone are not able to increase the speed beyond 10.4 fps since the memory access pattern becomes the bottleneck and limit the performance. After experimenting this, I decided to refine the memory accesses and I obtained a real time solution. Thanks to code optimizations, the software is transformed into a much efficient and parallel executable code. Compiler optimizations are inevitable since they provide a software program to better utilize the hardware resources and execution units of the DSP. During the memory access pattern improvement, the software structure changed much. Both the number of memory accesses and the number of operations are decreased.

The final performance of the encoder is good enough for a real time application. Encoder speeds above 25 fps are considered as real time so this implementation can be used for a real world application. The average speed of the encoder is about 26.7 fps. This value is measured by encoding a 20 frames long video sequence.

## 5.4 PSNR and Compression Rate Measurements

The PSNR values are average values for a video sequence of 30 frames, first frame being I-Frame, others P-Frame. Quantization parameter is fixed at 28, number of reference frames is one, and compression ratio is defined as the ratio of the original ".yuv "file to the compressed ".264" file. Average PSNR shows that the picture quality of the encoder is good enough. Compression ratios change with the video sequences. Compression ratio of above 100 is possible; however this ratio also decreases to 30. Depending on the video sequence the algorithm decides which block size, mode and vector to use. The fact that mother & daughter video sequence is encoded with more bits is because the residual luma/chroma coefficients contain many non-zero components and they have to be coded. For a better compressing encoder a rate control mechanism should be added. Such a rate control mechanism may change the quantization parameter to decrease the number of coded blocks. In the future such a mechanism may be added for rate distortion optimization.

Table 5.10: Compression efficieny of the encoder

| | **Average PSNR** | **Compression Ratio** |
|---|---|---|
| News (QCIF) | 39.739 | 63.3 |
| Claire (QCIF) | 41.481 | 102.2 |
| Container (QCIF) | 40.380 | 73.4 |
| Highway (QCIF) | 38.210 | 52.3 |
| Carphone (QCIF) | 40.278 | 32.2 |
| Mother & Daughter (QCIF) | 39.382 | 47.85 |
| Waterfall (CIF) | 36.288 | 35.4 |
| Foreman (CIF) | 38.899 | 41.0 |

# CHAPTER 6

# CONCLUSION AND FUTURE WORK

The proposed H.264 encoder is verified to work on both computer and TI DM642 EVM platform. The fact that the encoder is fully compliant with the standard H.264 decoder is very important. The output bit stream file can be decoded by the JM standard decoder. PSNR measurements of the encoded and decoded videos show that the picture quality is good enough for QCIF and CIF frame sizes.

For the optimization of this encoder on DM642 platform many optimization steps are experienced. Different L2 ram/cache partitioning is experimented to find the optimal partitioning. The memory access pattern is improved. This is achieved by the modifications in the algorithm and the creation of buffers for data storage and reuse. The allocation of compiler output sections are analyzed program data is allocated in the on-chip memory. Code optimizations such as replacing function with fast library functions, using intrinsics, function inlining and changing variable types are investigated. Optimizing compiler is utilized with different optimization options to achieve the best compiler optimizations. Lastly, the frequently accessed data arrays such as macroblock array or search window array are allocated on the fast on-chip memory.

After all optimizations the performance of the complete system increased by almost 8. This proves that there is much potential or parallelism in a video processing system. It is understood that code optimizations provide us with increased performance but memory management is a key issue in the optimization of a video encoder. It is shown that the source code can be parallelized with some modifications. However, the memory accesses limit the performance. We can overcome the computation complexity of the baseline encoder by applying suitable software and compiler optimization methods because the processor is fast enough to make fast computations. But in order to reach the performance of a real-time application the memory access pattern of encoder

must be improved. This improvement may include modifications in the flow of the encoder algorithm or memory access pattern of the complete encoder system.

The proposed encoder system is verified to work at 26.7 fps for a QCIF video sequence after optimizations. Before the optimizations the performance was about 3.31 fps. The applied optimizations enhanced the speed of the whole encoder system by 8.

For the future work, the proposed encoder can be improved by adding quarter-pel motion compensation support, error resilience tools. Moreover, other motion estimation algorithms can be integrated into the system. The current encoder achieves real-time execution for QCIF. In the future, I will implement a real-time encoder for CIF and higher resolution video formats. Encoding at higher resolution video formats requires extra optimization effort. For this purpose, EDMA can be used. Also using parallel instructions with packing/unpacking operations will increase the performance.

## REFERENCES


[1] Overview of H.264 / MPEG-4 Part10, Soon-kak Kwon, A. Tamhankar, K. R. Rao. Dongeui University, T-Mobile, University of Texas at Arlington, 2005.

[2] Joint Video Team (JVT) of ITU-T VCEG and ISO/IEC MPEG, Joint Model (JM) Reference Software Version 8.4, http://iphome.hhi.de/suehring/tml

[3] Spectrum Digital, "TMS320DM642 Evaluation Module Technical Reference", August 2003.

[4] Saponara,Denolf,Lafruit,Blanch,Bormans, "Performance and Complexity Co-evaluation of the Advanced Video Coding Standard for Cost-Effective Multimedia Communications", EURASIP Journal on Applied Signal Processing 2004:2, 220–235, 2004.

[5] Hamid R. Sheikh, Serene Banerjee, Brian L. Evans, and Alan C. Bovik, Optimization of a Baseline H.263 Video Encoder on the TMS320C6000".

[6] Texas Instruments, "Parallelization of a H.263 Encoder for the TMS320C80 MVP",literature number:spra339

[7] Darek Blasiak, Broadcast Quality H.264 Encoding and Transcoding, TI Developer Conference, 2005.

[8] Denolf, Vleeschouwer, Turney, Lafruit, Bormans, "Memory Centric Design of an MPEG-4 Video Encoder", IEEE Transactions on Circuits and Systems for Video Technology, Vol.15 No.5, May 2005.

[9] Cathoor, Dutt, Danckaert, Wuytack, "Code Transformations for Data Transfer and Storage Exploration Preprocessing in Multimedia Processors, IEEE Design & Test of Computers, 2001.

[10] Cheng Peng, "Video Encoding Optimization on TMS320DM64x/C64x", Texas Instruments Application Report, Literature Number: SPRAA63, October 2004.

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

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

[13] Deependra Talla, Lizy K. John, Viktor Lapinskii, and Brian L. Evans, "Evaluating Signal Processing and Multimedia Applications on SIMD, VLIW and Superscalar Architectures".

[14] Vishal Markandey, Dipa Rao, Texas Instruments, "TMS320DM642 Technical Overview", Application Report LiteratureNumber: SPRU615, September 2002.

[15] Texas Instruments, "TMS320C64x Image/Video Processing Library Programmer's Reference", Literature Number: SPRU023A, April 2002.

[16] www.elecard.com

[17] John Stevenson, Texas Instruments Code Composer Studio IDE v3 White Paper, Application Report, Literature Number: SPRAA08 - July 2004.

[18] Texas Instruments, "TMS320C64x DSP Library Programmer's Reference", Literature Number: SPRU565A, April 2002.

[19] Texas Instruments, "TMS320C6000 CPU and Instruction Set". Literature Number: SPRU189F, October 2000.

[20] Texas Instruments, "TMS320C6000Optimizing Compiler User's Guide , Literature Number: SPRU187L", May 2004.

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

[22] Rulph Chassaing , DSP Applications using C and the TMS320C6x DSK, Wiley, 2002.

[23] UbVideo, "UBLive-264MP: An H.264-Based Solution on the DM642 for Video Boadcast Applications", www.ubvideo.com, whitepaper, 2002.

[24] Texas Instruments,TMS320C6000 DSP/BIOS Application Programming Interface (API) Reference Guide, Literature Number: SPRU403G, April 2004.

[25] Texas Instruments,TMS320 DSP/BIOS User's Guide, Literature Number: SPRU423D, April 2004.