MAPPING AND OPTIMIZING A SOFTWARE-ONLY REAL-TIME MPGE-2 VIDEO ENCODER ON VLIW ARCHITECTURES

by

HENRY LEE

B.A.Sc (Electrical and Computer Engineering)

University of British Columbia, 1997

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQURIEMENT FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April 2000

Abstract

Due to its high computational demand, MPEG-2 video coding solutions have been based mainly on custom hardware (ASIC) systems. Such systems lack the flexibility and adaptability of software-based solutions. Achieving real-time MPEG-2 video encoding in software remains to be a major challenge. A typical MPEG-2 encoder performs 20 to 30 GOPS (giga operations per second), which exceeds the capabilities of the most advanced contemporary processors.

In this thesis, we have developed and tested a highly optimized, low complexity, high-quality MPEG-2 video encoder software based on Texas Instruments' fixed-point TMS320C6201 VLIW (Very Long Instruction Word) processor. First, we developed MPEG-2 video encoder software written in C for the C62x processor platform, however, due to the difference in the processor architecture, optimization and modification are done on the software to ensure the MPEG-2 video encoder runs efficiently in the VLIW architecture. The optimization are done at the assembly language level to maximize the attainable instruction-level parallelism (ILP) of the C62x VLIW architecture. In our experience, optimizations done alone by the optimizing C-compiler of the C62x could not meet the real-time requirements of MPEG-2. After code re-mapping and optimization, the resulting MPEG-2 video encoder implementation runs approximately 32 times faster than the original unoptimized MPEG-2 video encoder. Moreover, the current version of the encoder can handle SIF(320x240) video format at 16 frames per second with both I and P pictures, and CCIR-601 (720x480) at 15 frames per second for the I pictures only. Our real-time MPEG-2 encoder has been implemented and tested on the C62x Evaluation Model (EVM) board from TI.

Table of Contents

List of Tables

List of Figures

Acknowledgments

First of all, I would like to thank my parents and my three elder brothers, for their great love and confidence in me, and for making my life easier during my heavy courses load and research work.

My research would not have become a reality without the very valuable supervision of Dr. Hussein Alnuweiri. The guidance and support are greatly appreciated.

I would like to thank my colleagues Don Hoi Ng and Joseph Chu for sharing their valuable technical knowledge with me. I also wish to express my gratitude to Dr. Faouzi Kossentini and Dr. Khanh Nguyen for the useful reference materials they has provided me and our research engineer Dr. Alen Docef for making our lives easier with his great technical support. Last but certainly not the least, I thank all my colleagues and friends for their invaluable friendship. This research was supported by a grant from the National Sciences and Engineering Research Council (NSERC) of Canada and by a grant Rogers Communications.

Chapter 1

Introduction

With the growing deployment and commercialization of the multimedia applications, comes the demand for higher performance to be offered to such applications at the lowest possible cost. Additionally, multimedia processing platforms must become more flexible and easily re-programmable to keep up with changing standards and application domains. Currently, video signal processing is the dominating task in terms of computational demands as well as amount of information bandwidth, unless compression technology is used. The wide demand for video coding applications have led to the development of video coding standards for video compression such as ISO/IEC MPEG-1 and MPEG-2. The required processing rate for video compression ranges from 100 mega operations per second (MOPS) to more than one tera operations per second (TOPS). For instances, the real-time MPEG-2 video decoding for National Television Systems Committee (NTSC) resolution requires more than 400 MOPS, while MPEG-2 video encoding can require up to 30 giga operations per second (GOPS) [1]. Such requirements are beyond the capabilities of contemporary microprocessors and have been satisfied mainly through the use of Application Specific Integrated Circuits (ASICs) technology only. However, this trend is changing with the advent of new fundamental algorithmic enhancements to video encoding, and with the development of media-enhanced processors from various vendors such as the Visual Instruction Set (VIS) extension of Sun Microsystem's UltraSPARC [2], the MMX extension of Intel's Pentium processors, the MAX-2 extension of Hewlett-Packard's PA-RISC [3], and the Multimedia Instruction Set extensions of Silicon Graphics' microprocessor [4]. These enhancements have helped real-time video applications in software become reality.

Current solutions for real-time MPEG-2 video encoding and decoding are based principally on VLSI implementations such as the custom hardware (ASIC) systems. However, in order to make the implementations flexible and cost effective, migrating functionality from application specific hardware into software running on a programmable general-purpose processor or DSP start to gain great deal of interest. Recently developed microprocessors and programmable DSP chips offer powerful processing capabilities to do real-time video compression/decompression. These processors were designed with the target of realizing a software-implemented MPEG-2 encoder/decoder in real-time.

In this thesis, we have developed and tested a highly optimized, low complexity, high-quality MPEG-2 video encoder software that runs in real-time or near real-time on Texas Instruments' fixed-point TMS320C6201 VLIW (very long instruction word) processor. The high performance TMS320C6201 VLIW DSP processor was chosen among all the microprocessors and programmable DSP because it employs the VelociTI VLIW architecture [5]. Our encoder software is based on the encoder design by Dr.

Kossentini and others of the SPMG group at UBC. However, because of the distinctive features of the VLIW architecture, the original MPEG-2 video encoder software had to be remapped and modified substantially according to the memory and processor architecture of the targeted VLIW processor. The modifications and optimization have been applied mainly to the time consuming functions of the software, which include the DCT, IDCT, block matching (SAD), quantization, and variable length coding. Given the inefficiency of the optimizing C-compiler of the C62x, the optimization of most coder functions is mainly done at the assembly language level to maximize the attainable ILP (instruction level parallelism) of the C62x VLIW architecture. As a result of the assembly level optimization and the memory re-mapping the MPEG-2 video encoder runs approximately 32 times faster on the C62x than the unoptimizied video encoder implementation. Moreover, the resulting MPEG-2 video encoder implementation can encode SIF video format at 16 frames per second with both I and P pictures, and I pictures only in CCIR-601 video format at 15 frames per second on the Texas Instrument's C62x Evaluation Model board. We believe that this is a big leap forward towards achieving real-time video encoding in software-only on a VLIW DSP processor.

This thesis is organized as the follows. Chapter 2 provides an overview of VLIW technology with the Instruction Level Parallelism (ILP) architecture and the related issues of the compiler technology of VLIW. Following the discussing of VLIW technology, the MPEG-2 video coding standard and the software solution for real-time MPEG-2 video encoder proposed by the SPMG group of UBC is discussed in chapter 3. Chapter 4 introduces the Texas Instrument's C62x VLIW architecture and provides some brief information on its instruction set and memory structure. This chapter also describes the mapping issues of the MPEG-2 video encoder on the EVM and the memory configuration and host interface of the algorithms. In Chapter 5, the methodology of optimization of the software MPEG-2 video encoder is discussed along with some result comparison. Finally, conclusions for the real-time software MPEG-2 video encoder and suggestions for the future work towards the full feature real-time MPEG-2 video encoder are given in Chapter 6.

Chapter 2

An Overview of VLIW Technology and Different Architecture Approaches for Video Processing

2.1 Introduction

As stated earlier, the main objective of this thesis is to develop an efficient implementation of the MPEG-2 video encoder on the VLIW architecture. Therefore, a mutual understanding of the VLIW architecture and realizing the main differences among programmable multimedia processor architectures are needed. In this chapter, we first present the basic concepts of ILP. This is followed by a discussion of ILP architectures. A presentation of the compiler-architecture technology for the VLIW processors is given. The chapter then concludes with a discussion of the different architectural approaches to programmable processors, which are desirable for the implementation of video processing.

Traditional DSP-oriented chips were mainly built for inner-loop-oriented engines that acted as slave processors to a CPU, and let the CPU run the rest of the application and the operating system [6]. However, over the past two decades research on VLIW architectures has led to the development of stand alone DSP-oriented microprocessor chips based on this concept. Additional advancements in the field of compiling and architecture technology has made it possible to write the actual compute-intensive loops as part of the whole application. As many of the emerging multimedia applications continuously change, programmable architectures that provide higher degree of flexibility and greater processing power became desirable. Recently, VLIW is one of the fastest growing architecture designs that fulfill high performance needs. The VLIW architecture has its concept aims at exploiting the instruction-level parallelism inherent in many multimedia algorithms to improve the performance. Multiple operations are specified within a single long instruction word for concurrent execution in every clock cycle. Hence, multiple parallel functional units are implemented in such a way that they can concurrently execute instructions. However, full utilization of the parallelism of the functional units is difficult to achieve and sustain, often resulting in the occurrence of idle resources at run time. Furthermore, compilers for DSPs have generally been unable to exploit the features of ILP architecture efficiently. As a result, a significant part of DSP applications has to be optimized manually at the assembly language level. Therefore, VLIW architectures have two major issues that must be addressed: one is the ILP issues and its architectures; and the other is the compiler-architecture interaction for VLIW.

On the other hand, the differences between architectural approaches for programmable processors must also be addressed, as the different architectural techniques applied would be able to increase the multimedia performance. By exploiting the differences of the architectural approaches, we would further understand the adequate/inadequate features of the current processor for handling multimedia applications. The techniques can be divided mainly into parallelization strategies and adaptation strategies. For the parallelization strategy, it's done primarily on the data, instruction, and task levels, which can result in a considerable increase of computational power by achieving high level of concurrency of operations. The adaptation strategy increases the efficiency of an architecture by economical use of hardware resources.

2.2 Instruction-Level Parallelism (ILP) issues

VLIW is a processor with a design philosophy similar to that of the reduced instruction set computer (RISC). A VLIW processor is basically a logical extension of the RISC processor, except it has more advanced features by adding ILP to a processor. As mentioned before, ILP is the key architectural technique employed by today's high-performance DSPs. ILP is a set of design techniques that speed up the programs by executing several RISC-style operations in parallel, such as memory loads and stores, integer additions, or multiplications. These operations execute single-cycle operations on hardware functional units, rather than performing parallel tasks. Only at the finest-grain level are these operations assigned to every possible function units such as adders and memory units at the same time as shown in Figure 1. Parallel execution occurs in every functional unit during a single-cycle of operation [7]. This parallelism is largely transparent to the user. However, advanced users may be well aware of its operation and may restructure the code or carry out other actions to enhance ILP. Distinguishing between the latent or available ILP inherent in a segment of code and the realizable or achievable ILP provided by the hardware would be a key to maximizing the efficiency of the ILP architecture.

ALU 1	ALU 2	MUL	Data1	Data2	ALU 3
add r1, r2	sub r3, r4	mul r5, r6	load r7, r8	store r9, r10	and r11, r12

***all six functional units are assigned with operations**

Figure 1: Example of finest-grained level of operations assigned to all functional units.

2.2.1 ILP Hardware

Several different ways of hardware support are offered in ILP. For instances:

- more than one functional unit in a processor can execute concurrently,
- functional units with latency longer than one cycle can be pipelined,
- multiple functional units can access different register files to add register bandwidth for the purpose of parallel execution.

This hardware support enables the parallelization of the same RISC operations that are executed sequentially in the view of the programmer [8]. An example of the execution hardware enhanced to exploit potential ILP is depicted in Figure 2, assuming that the long instruction word execution takes place during a single-cycle. Having separate register banks for the integer and floating-point data can help reducing potential hardware resource conflicts. Also including extra integer units enables access to more than one integer operation per cycle.

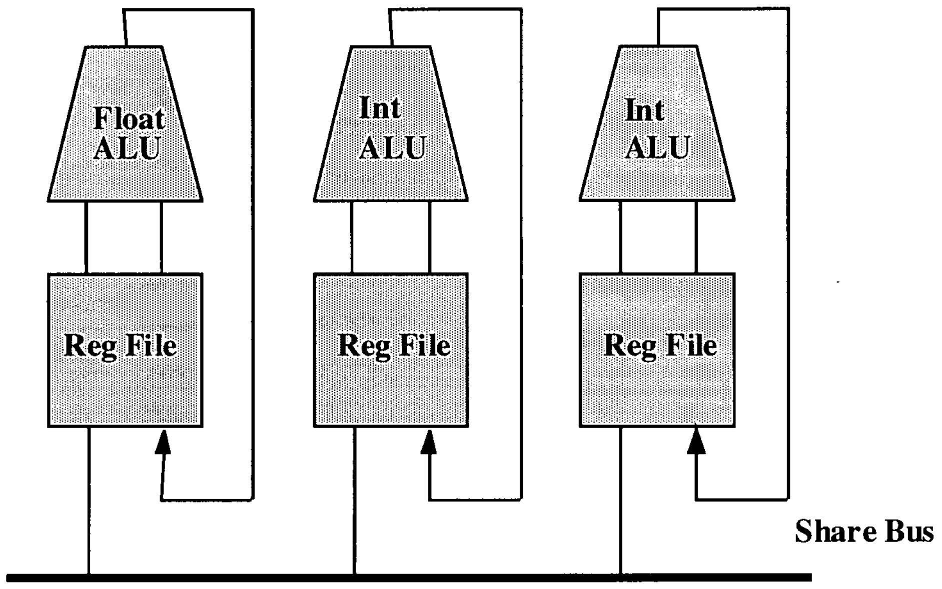


Figure 2: Hardware enhancement for ILP.

Besides the execution hardware of an ILP processor, microprocessors have control hardware as well, which controls the flow of operations instead of carrying out the operations that contribute to the desired computation. There are mainly two types of architectures embodying the ILP methodology in modern microprocessor control units, and both can increase code performance by statically or dynamically scheduling parallel streams of single-cycle operations. The two types are: superscalar processors which employs dynamic instruction scheduling, and VLIW processors which employ static instruction scheduling. It should be noted that the pipelining structure of the two processor types is different.

2.2.1.1 Superscalar Processors

In general, most general-purpose microprocessors embody a form of ILP called superscalar execution. However, the main difference between scalar and superscalar general-purpose processor is that the control hardware is much more complex for the latter processor type. Superscalar processors are designed to exploit more ILP in user programs. Only independent instructions can be executed in parallel without causing wait states. The amount of ILP varies widely depending on the type of code being executed. A superscalar processor fetches several RISC-level instructions from a scalar instruction stream and processes these instructions in parallel, all these decisions are made at the runtime. The control hardware of the processor checks the dependencies among the operations, reorders the instructions to take advantage of free function units or other resources, performs register renaming, then executes operations in parallel if possible; thus speeding up the computation [9]. This more complex hardware is often referred to as scheduling hardware. A superscalar processor is handed ordinary code, compiled for a sequential model of computation, and the scheduling hardware produces the ILP. For the superscalar type of architecture, a specific compiler is not required.

- **Pipelining in Superscalar Processors:** The fundamental structure of a superscalar pipeline is illustrated in Figure 3. The diagram shows the use of three instruction pipelines operating in parallel for a triple-issue superscalar processor. The superscalar processor shown below, of degree three, can issue up to three instructions per cycle, where the base scalar processor implemented either in RISC or CISC can only have a degree of one. To fully utilize a superscalar processor of degree three, three instructions must be executable in parallel. However, this situation may not happen in all clock cycles. In that case, some of the pipelines may be stalling in a wait state. For most of the superscalar processors, the simple operation latency should require only one cycle. To achieve a higher degree of instruction-level parallelism in programs, the superscalar processor depends on an optimizing compiler to exploit ILP.

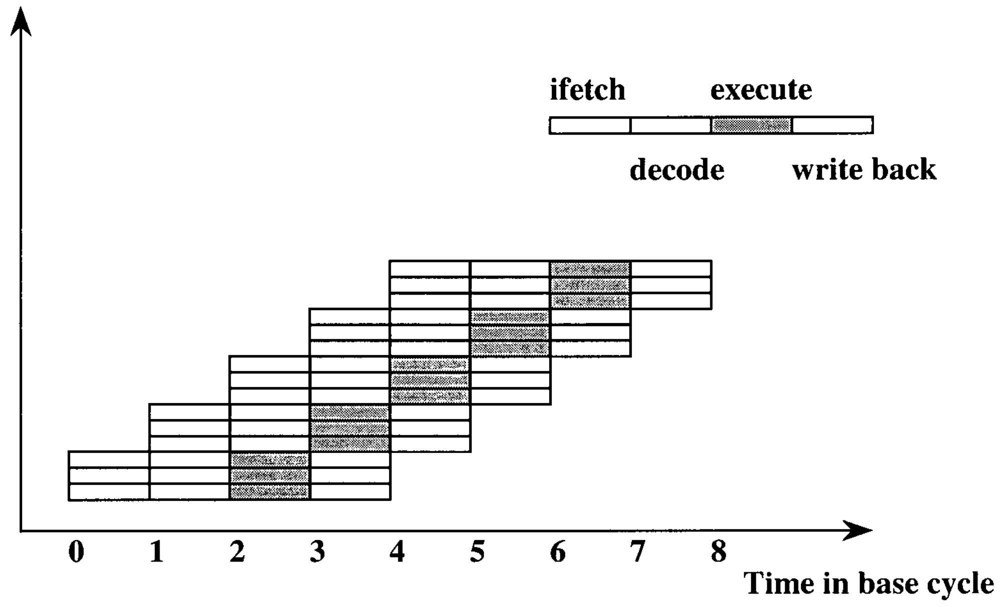


Figure 3: Fundamental structure of a superscalar pipeline with degree of three.

2.2.1.2 VLIW Processors

The VLIW processors are in general very similar to superscalar processors in the sense that both processors use the same or similar execution hardware to achieve the ILP, except that the VLIW processors have extremely simple control units. A typical VLIW processor has instruction words hundreds of bits in length such as Texas Instruments' fixed-point TMS320C6201 VLIW processor is 256-bit long [10]. In a VLIW, it is the task of the compiler to determine the ILP and to schedule operations to run on multiple functional units concurrently. All the functional units share the use of a common large register file and the operations to be simultaneously executed by the functional units are synchronized in a VLIW instruction. The compiler communicates the information about where, how, and when things are done via the program itself, by specifying directly in each cycle exactly what each function unit is to do, where it gets its data from, etc. Since this design philosophy mandates the simplest possible control hardware, it is common to put no-ops in each instruction corresponding to those functional units, making their task in that cycle explicit. However, inserting no-ops to all the unused functional unit leads to the problem of code size expansion. Coding size is especially critical issue for DSP applications. Furthermore, the static nature of scheduling has also led to a more complex compiler toolkit.

The immediate advantage of VLIW methodology is that both power consumption and silicon area are significantly reduced compared to an equivalent superscalar architecture. One drawback of traditional VLIW architectures is code expansion resulting from inserting the no-op operation into the unused functional units during a VLIW execution packet instruction cycle. Depending on the number of functional units, N, VLIW code expansion could theoretically increase the code size by as much as N times.

- **Pipelining in VLIW Processors:** The execution of instructions by an ideal VLIW processor is shown in Figure 4, where each instruction specifies multiple operations.

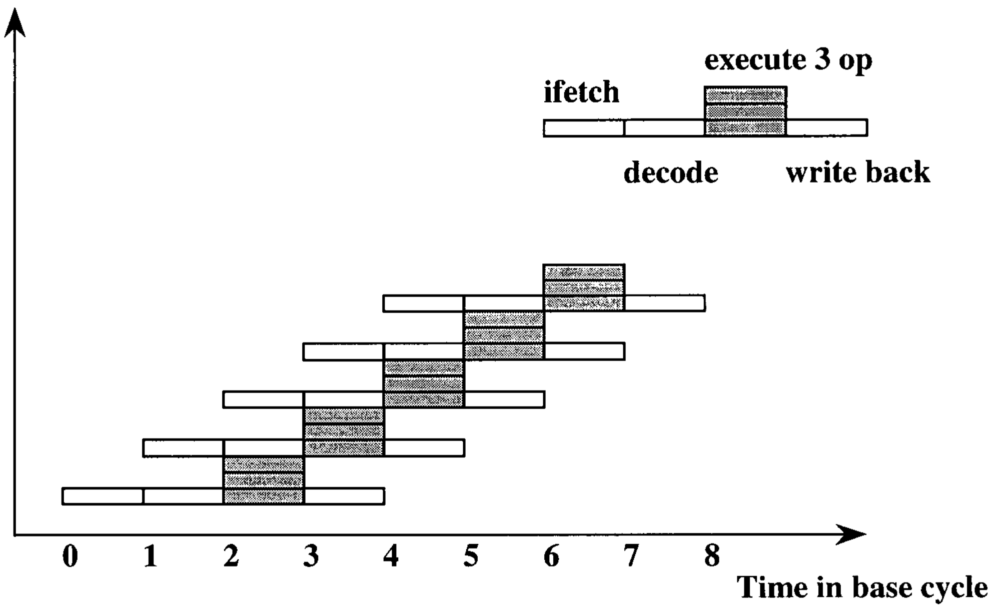


Figure 4: Execution of VLIW processor instructions in a pipelined fashion.

VLIW processors behave much like superscalar processors except for the following three main differences:

1. VLIW instructions are much easier to decode than the superscalar instructions.
2. In terms of code density, the superscalar processor is better when the available ILP is less than that exploitable by the VLIW processor. This is due to the fixed VLIW format that includes bits for non-executable operations such as the no-ops described previously, while the superscalar processor issue only executable instructions.
3. A superscalar processor can be object-code-compatible with a large family of nonparallel processors. In contrast, a VLIW processor exploiting different amounts of parallelism would require different instruction sets.

2.2.2 The Use of VLIW in DSP Applications

The extreme popularity of the use of VLIW architecture in the DSP world comes for several reasons. One of the most important reasons is the ability to take advantage of the abundant ILP that's available in typical DSP codes. However, the control hardware can become enormous when it must control large numbers of functional units. However, the major disadvantage of VLIW architecture as general-purpose microprocessor is the issue of object code compatibility. Programs for different VLIW are likely to be different, and thus the code must be recompiled for each processor architecture. The object code compatibility tends not to be an impediment in the DSP applications, since recompilation and unique hardware have been the norms. The relatively static flow of control in DSP and multimedia applications has made VLIW a good choice.

2.3 Compiler-Architecture Technologies for VLIW

Compiling for VLIW DSP processors was done poorly for the past decade, and the only way to obtain an efficient coding was though the hand coding assembly. Clearly, a high-level language approach was much more desirable. Unfortunately, until 1980s, there were no approaches that matched, or even came close to, the performance that hand code could provide. The major barrier faced, even by experienced compiler writers, was that available practical compiler technology only enabled the scheduling of operations that come from single straight-line segments of code, called "basic blocks" by compiler writers. Whenever the compiler encountered a jump of any form, or even a branch target, the compiler would stop any further scheduling, declare that section done, and starts all over with the next section. Unfortunately, there was little performance to be gained that way. If operations could somehow be moved globally between blocks rather then locally, far higher degrees of ILP would be available. Therefore, performance gains achieved with VLIW architectures will critically depend on the degree of exploitable instruction-level parallelism in the target algorithm as described previously. To maximize the pool of operations that can be scheduled into a single instruction word, sophisticated compiler techniques such as loop unrolling, software pipelining, trace scheduling, and guarded execution may be applied [11].

2.3.1 Trace Scheduling

As the name of the scheduling algorithm states, trace scheduling is centered on traces [12]. The compiler mainly focuses on loop-free sequences of basic blocks embedded in the control flow graph rather than a single basic block. Traces are selected and scheduled in order of their frequency of execution. The selected trace is then scheduled as if it were a single basic block as giving no special consideration to branches. By scheduling the trace all at once, the compiler implicitly moves operations between blocks. After a given trace has been scheduled, the next most frequently executed trace is selected from among the remaining unscheduled operations, including those added in the process of scheduling earlier traces. This continues until the entire program has been scheduled.

The basic concept initiated by trace scheduling is to do the code motions implicitly as part of scheduling region, which is much larger than a basic block. Frequently in DSP and multimedia applications, the compiler is presented with a single small loop. Hence, trace scheduling does not proceed beyond a back edge, there would be little opportunity for the compiler to do the global code motions that allow trace scheduling to be truly efficient. Trace scheduling compilers typically do a sophisticated job of turning a small piece of code into a long chain of blocks without back edges and without the artificial data dependencies that a naïve job of unrolling would introduce. Unrolling the loops can be very effective as will be discussed further in the following section. However, it also bring the problem of generating much extra code and sometimes loses performance because there is always a start up and close down delay at the beginning and the end of each series of unrolled iterations.

2.3.2 Loop Unrolling

Loop unrolling can be a very effective compiler technique, as it replicates the original loop body multiple times, and adjusts the loop termination code and eliminates redundant branch instructions. The overhead of branching is significantly reduced with loop unrolling, especially when the latency of branching is large which is usually the case in most of the DSP architectures. This technique usually incorporates reduction of data dependencies to fully utilize the efficiency of loop unrolling. The resulting larger basic block increases the probability that the instruction scheduler can reorder instructions to exploit ILP [13]. Figure 5 shows an example of the compiler performing the loop unrolling technique. The initial iteration of the loop is 16 and the loop body is performed once every iteration. After performing loop unrolling, the compiler transform the loop iteration to four, and the loop body is executed four times every iteration. Given that unrolling the loop not only reduces the redundancy of branch instructions with long latency, but also increases the probability of efficient pipeline scheduling.

```
int size = 16;
byte out[size], in1[size], in2[size];
for (i = 0; i < 16; i++)
        out[i] = in1[i] + in2[i];
```

Before loop unrolling

```
Cycle 0:        load t1_0 = [in1 + i0]
                load t1_1 = [in1 + i1]
                load t1_2 = [in1 + i2]
                load t1_3 = [in1 + i3]
cycle 1:        load t1_0 = [in2 + i0]
                load t1_1 = [in2 + i1]
                load t1_2 = [in2 + i2]
                load t1_3 = [in2 + i3]
cycle 2:        add t3_0 = t1_0, t2_0
                add t3_1 = t1_1, t2_1
                add t3_2 = t1_2, t2_2
                add t3_3 = t1_3, t2_3
cycle 3:        store [out + i0] = t3_0
                store [out + i1] = t3_1
                store [out + i2] = t3_2
                store [out + i3] = t3_3
```

Code section for the loop body after unrolling the loop in the compiler

Figure 5: An example of loop unrolling compiler technique.

However, loop unrolling generates extra code and sometimes loses performance because the scheduler's effectiveness is limited by artificial dependencies created by loop unrolling's naïve reuse of registers and other data dependencies between instructions. Code size expansion tends to be a problem in most DSP processors, because the program memory size is very limited. Therefore, the level of loop unrolling must be compensated with the limited code size expansion.

2.3.3 Software Pipelining

Software pipelining parallelize loops by starting new iterations before previous iterations complete. Successive iterations start at every initiation interval cycles. A single iteration's schedule can be divided into count stages, each consisting of initiation interval cycles. Figure 6 shows the execution of five iterations of a four-stage software pipelined loop. The three phases during execution of the loop are ramp up, steady state, and ramp down. The first stage count minus one cycle constitutes the ramp-up phase when not all stages of the software pipeline are executed. The steady-state portion begins with the last stage of the first iteration. During the steady-state phase, one iteration completes for every one that starts. The steady-state phase ends when the first stage of the last iteration has completed. For the final stage count minus one cycle is known as the ramp-down phase, one iteration completes every initiation interval cycle. The code generated for the three phases of the software pipelined loop are prologue, kernel, and epilogue [14].

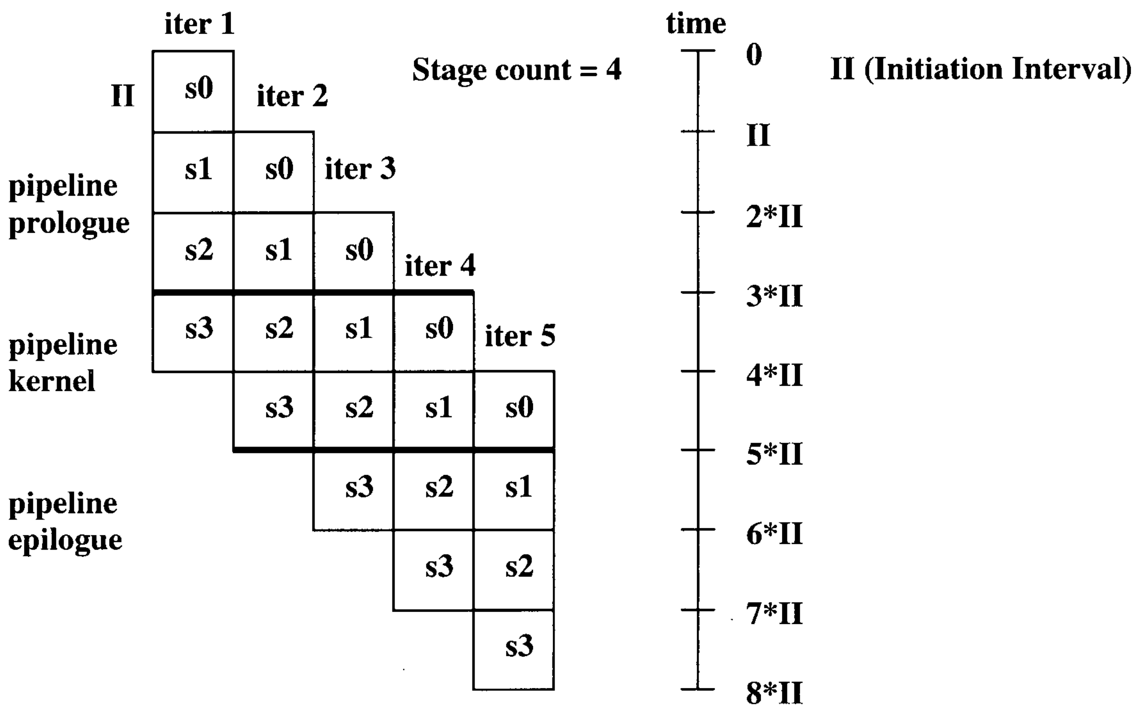


Figure 6: Software pipelined loop execution.

Assume that without software pipelining every iteration takes N cycles to complete. With software pipelining, after the prologue, every new iteration takes one initiation interval cycle to complete. The speed up is a factor of N divided by initiation interval, when ignoring the overhead of the prologue. Scheduling the prologue and epilogue code in parallel with the code outside the loop would minimize this overhead. The overhead is insignificant for loops with large trip counts. Software pipelining works very efficiently for very simple loops and continuing research has broadened its applicability considerably.

2.4 Architectural Approaches for Programmable Processors

In this section, various approaches are discussed that are widely employed to enhance the processing capability of programmable architectures for multimedia. Parallelization strategies are examined separately on data level (SIMD, split-ALU), instruction level (VLIW), and task level (MIMD, associative controlling). Adaptation strategies are divided into instruction set design modifications (specialized instructions) and the incorporation of dedicated hardware modules (coprocessors). Additionally, some remarks are given on the memory system design for programmable multimedia processors.

2.4.1 SIMD

Data parallelism in image and video processing algorithms can efficiently be exploited by SIMD (single instruction stream, multiple data streams) architectures. In SIMD processors, a number of parallel data paths is centrally controlled by a global control unit. All data paths execute the same stream of instructions, but operate on different data items. To enable conditional operations, individual data paths can be excluded from the execution of single instructions by binary masking [9]. Figure 7 shows the basic structure of the SIMD architecture.

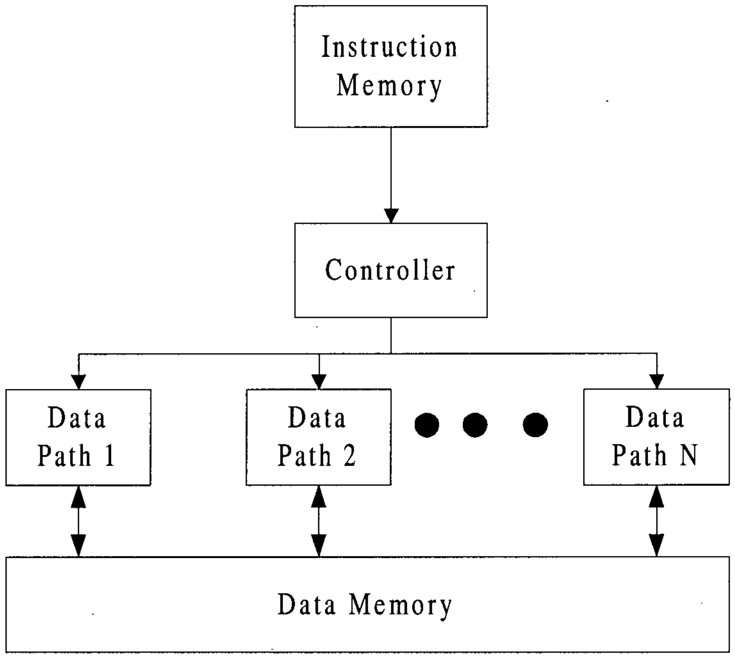


Figure 7: SIMD multiprocessor architecture.

As SIMD processors feature just a single control unit, the largest portion of available silicon area can actually be spent for the implementation of multiple data paths, leading to high degree of parallelism. However, the reduced control overhead is paid for by a lack of flexibility in case of more diverse computation requirements. While SIMD processors are highly efficient for algorithms with highly regular computation patterns, i.e., low-level tasks, utilization rapidly decreases for more irregular algorithms involving a higher portion of data-dependent processing. In consequence, pure SIMD architectures are not well suited for the implementation of complex processing schemes of heterogeneous nature as frequently encountered in multimedia applications.

2.4.2 Split-ALU

Based on a principle similar to SIMD, the split-ALU concept also targets data-level parallelism in multimedia applications. The concept, also referred to as subword parallelism, involves parallel processing of several lower-precision data item on a single ALU of higher word length: a 64-bit ALU, for example, can execute a single operation on eight 8-bit data items simultaneously [15]. The implementation of a split-ALU requires only minor hardware extensions – basically, the carry signals arising in arithmetic operations have to be prevented from being propagated across the boundaries of separate data items. A possible split-ALU implementation is shown in Figure 8.

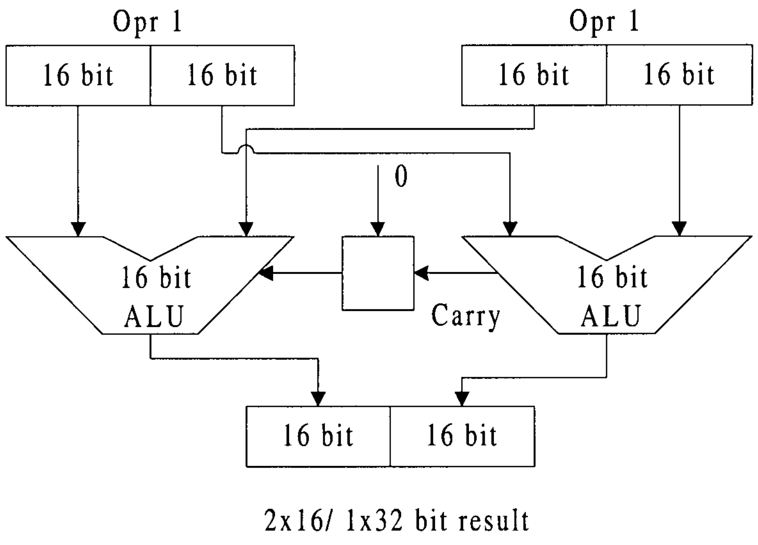


Figure 8: Split-ALU implementation.

Similar to SIMD architecture, the main benefit from a split-ALU is obtained for highly regular low-level algorithms involving identical operations executed on large data volumes. In addition, the degree of exploitable data parallelism depends on the precision required for an operation: with increasing word-length demands, fewer data items can be processed in parallel. Since image and video processing tasks mainly involve operations on low-precision (8-bit) pixel data, subword parallelism can effectively be exploited to enhance the performance for these computation types. By providing different split-ALU instructions for various data formats, the achievable parallelism can gradually scale with the precision requirements.

The small incremental hardware cost for a split-ALU makes this concept well suited for the extension of existing general-purpose processors with respect to multimedia processing. Typical operations to be executed in a split-ALU may include parallel addition/subtraction, multiplication, or compare. Furthermore, packing and unpacking operations have to be supported to enable a transition between conventional data words and packed subwords. As a drawback, split-ALU instructions are generally not supported by current compilers due to the lack of adequate high-level language constructs to express the desired operations. Moreover, complex code transformations usually become necessary before the packing of low-precision items into single long data words is enabled. Therefore, split-ALU instructions typically have to be inserted into the high-level language code manually by the programmer in form of intrinsic functions.

2.4.3 VLIW

The VLIW processor architecture concept aims at exploiting the instruction-level parallelism. There are two types of VLIW processors. One is the VLIW processor with static mapping of the operations slots which relies on static operation scheduling at compile time. The other is the superscalar processor which dynamically detects instruction-level parallelism at runtime.

Performance gains achieved with VLIW architecture critically depend on the degree of exploitable instruction-level parallelism in the target algorithm. However, the performance limitations of VLIW processors may arise from the growing hardware expense for multiported register files and crossbar switches when the number of functional units increases. Moreover, compiler techniques like loop unrolling and trace scheduling lead to a heavy increase in code size. The already high bandwidth requirements for instruction supply are likely to worsen due to the growing gap between processor and I/O speed. This problem is addressed by compressing the instruction words in memory and expanding them for execution. Details were given in the previous sections.

2.4.4 MIMD

An approach exploiting parallelism on both data level and tasks level is given by the MIMD (multiple instruction streams, multiple data streams) architecture concept. In contrast to SIMD architectures, MIMD processors feature a private control unit for each single data path. Consequently, each data path is provided with its individual instruction stream, enabling concurrent execution of different programs or tasks. Likewise, by supplying several data paths with identical instruction streams, data parallelism can be targeted as well [16]. The MIMD multiprocessor concept is depicted in Figure 9.

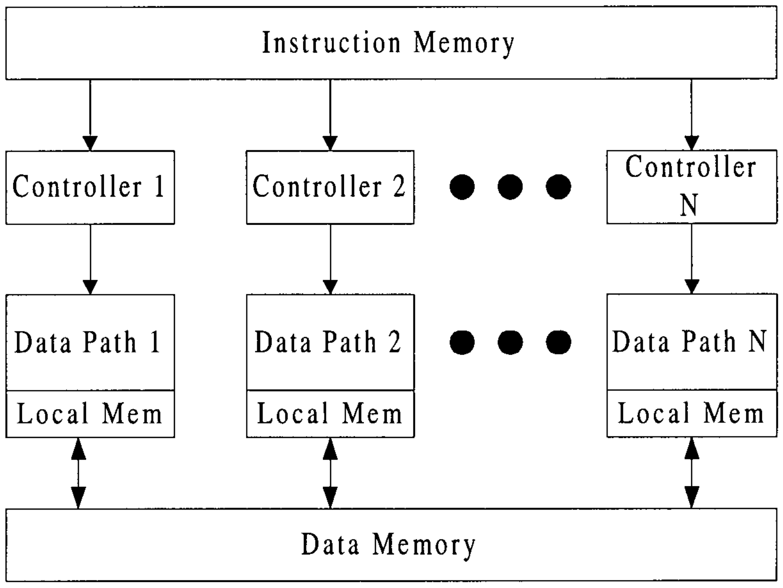


Figure 9: MIMD multiprocessor architecture.

The major advantage of MIMD architectures is their high flexibility as each data path can be controlled individually. Therefore, MIMD architectures are typically better suited than SIMD architectures for the execution of compound multimedia processing schemes comprising low-, medium-, and high-level tasks. However, the duplication of control units and high-bandwidth requirements for a continuous supply of several instruction streams dramatically increase the hardware expense, thus limiting the number of data paths that can be economically implemented on a single chip. Furthermore, MIMD processors include poor programmability and lack of synchronization support. Typically, scalar programs have to be developed separately for individual data paths, and synchronization among processing elements has to be achieved manually by the programmer. The highly complex and time-consuming application development may be one reason for the non-popularity of the MIMD processors in multimedia processing.

2.4.5 Associative Controlling

Associative controlling denotes a new concept for multiprocessor architectures aiming to prevent the typical drawbacks of both SIMD and MIMD processors in multimedia processing. The outstanding feature from the hardware point of view is a lower number of control units than parallel data paths. Therefore, in terms of hardware expense, an associatively controlled multiprocessor can be classified between SIMD and MIMD architectures [17]. Figure 10 gives a structural overview of an associatively controlled processor.

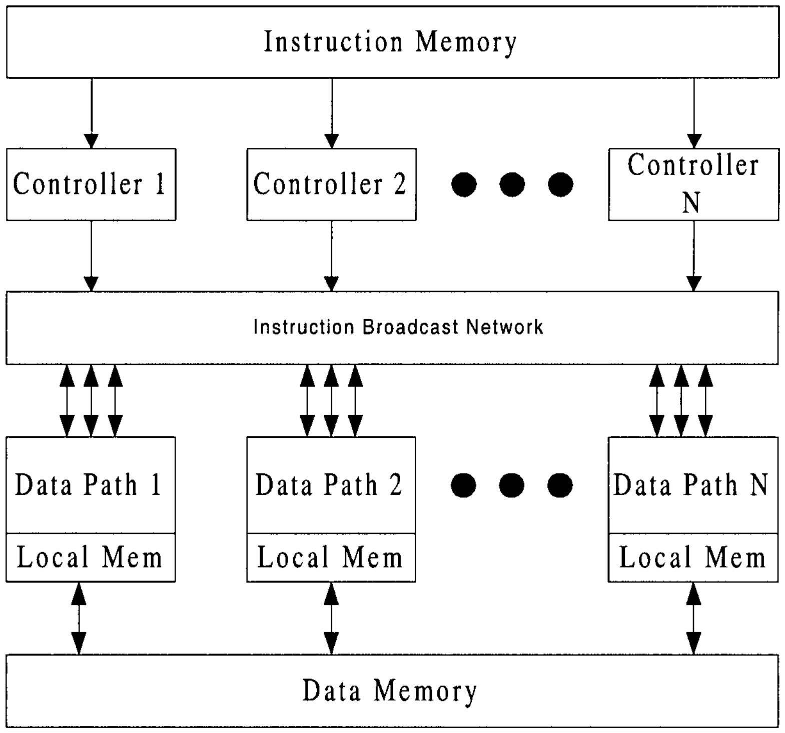


Figure 10: Associatively controlled multiprocessor.

The control units concurrently issue their individual instruction streams via an instruction broadcast network to all parallel data paths. Among the multiple instruction streams offered, each data path autonomously selects an appropriate instruction stream for execution. In order to enable unambiguous identification, special signatures are assigned to the instruction streams and supplied to the data paths. Individual data paths can dynamically switch between different instruction streams offered by the control units. This enables, for example, an efficient execution of data-dependent control flow by simultaneously supplying the instructions of alternative branches following a decision to all data paths and letting each data path select the proper branch according to its local status. Moreover, an associatively controlled processor is able to adapt to varying parallelization degrees during computation by supporting a dynamic clustering of data paths. Issues to be further investigated include the possible area and power overhead for the distribution and selection of concurrent instruction streams and the necessary compiler support to assist software development for associatively controlled processors.

2.4.6 Specialized Instructions

Programmable processors can be adapted to specific algorithms by introducing specialized instructions for frequently occurring operations of higher complexity. The use of specialized instructions reduces the instruction count and accelerates program execution [18]. For example, the use of a specialized MAC (multiply-accumulation) with saturation replacing a longer sequence of standard instructions including branches is very common in multimedia processing. Other examples of specialized instructions for multimedia include extended shift, minimum/maximum, average, and add-sign operations.

The incorporation of specialized instructions necessitates incremental hardware cost for additional function units such as a multiply_add. However, the design complexity of additional units can usually be kept modest due to high specialization and optimization toward the targeted operations. The decision on which specialized instructions to implement involves a tradeoff between additionally required hardware effort and probability of their use. The benefit from specialized instructions is not universal, since only a subset of algorithms will experience an acceleration.

2.4.7 Coprocessor/Heterogeneous Multiprocessor

Both parallelization and adaptation principles are combined within coprocessor/heterogeneous multiprocessor architectures. Coprocessor architectures for multimedia typically comprise a flexible general-purpose processing module, for instance, a standard RISC code performs high-level tasks of lower computational requirements as well as control and I/O functions, whereas the adapted module executes the computation-intensive but regular low-level tasks [19]. Figure 11 shows an example of a coprocessor architecture organization.

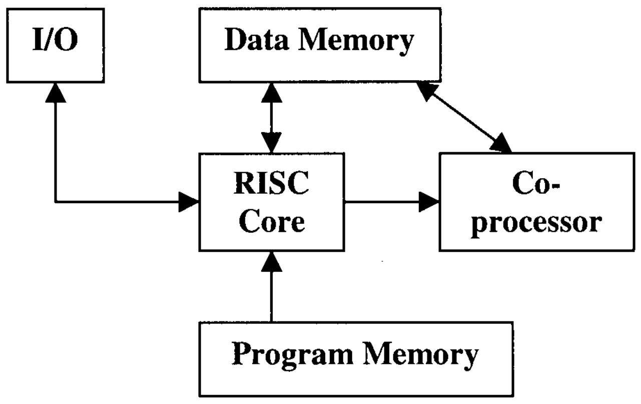


Figure 11: An example of coprocessor architecture.

Alternatively, specialized modules may also be employed for tasks of lower computational requirements, but with other special algorithm characteristics that make them difficult to implement on standard processors. The mixture of several programmable and dedicated modules results in a heterogeneous multiprocessor architecture, which can be seen as a generalization of the coprocessor concept. While heterogeneous multiprocessors allow us to reach higher performance levels by emphasizing parallel execution of several tasks, they are still more tied to a specific processing scheme than homogeneous multiprocessors due to the adaptation of individual modules.

2.4.8 Memory System

As multimedia processing schemes operate on high data volumes, memory system design has a significant impact on the overall system performance. The streaming nature of multimedia data deviates radically from conventional data access patterns in general-purpose computing. Typical cache strategies employed in general-purpose processors rely on frequent accesses to the same data items, and thus offer little benefit for multimedia processing. However, particularly in the low-level parts of multimedia algorithms, memory access patterns are very predictable. Therefore, special stream caches have been proposed that employ prefetching techniques in order to access shortly needed data in advance [20].

In addition to the streaming multimedia data types, data structures of nonvolatile nature may also be involved in multimedia processing such as coefficient lookup tables.

When placed in cache, those data structures would be exposed to frequent replacement by the streaming data types. Therefore, the additional integration of software-controlled on-chip SRAM is advantageous where data are safe from being replaced and can always be accessed within the shortest time [21].

The performance of multimedia processing is also influenced by the instruction memory behavior. As typical for most signal processing applications, multimedia processing involves a limited set of tasks periodically executed on incoming data streams: the same set of instructions is repetitively fetched and executed. Therefore, conventional cache strategies may prove useful for speeding up instruction access, provided the cache is large enough and mutual code replacement can effectively be prevented. Code positioning schemes have been proposed that help to increase instruction cache performance in multimedia applications [22]. Instruction memory performance can further be enhanced by the integration of on-chip SRAM for the most heavily executed code parts, thus always guaranteeing fastest access without stall cycles due to cache misses.

Chapter 3

A Real-Time MPEG-2 Video Encoder

3.1 Introduction

After realizing the basic architecture of VLIW processors and understanding the different approaches of programmable multimedia processors, the next crucial step is to obtain an efficient implementation of a real-time MPEG-2 video encoder. In this chapter, we first present the basic concepts of video coding. This is followed by a discussion of the MPEG-2 standards and the real-time implementation of the MPEG-2 video encoder proposed by our Signal Processing and Multimedia Group of UBC. This chapter concludes with an analysis of the proposed software solution for the real-time MPEG-2 video encoder.

3.2 Basics of Video Coding

In general, video sequences contain a significant amount of statistical and subjective redundancy within and between frames, i.e. video sequences usually contain statistical redundancies in both the temporal and spatial domains. Thus, the goal of video coding is to remove statistical and psychovisual redundancies in a video sequence while keeping intact as much perceptually important information as possible. Temporal redundancies normally exist due to structured motion and camera movement, while the spatial redundancies are normally due to the smoothness and edge continuity. Consequently, these two types of redundancies are efficiently reduced through employing the two well known compression schemes: inter-frame coding and intra-frame coding. Intra-frame compression reduces spatial redundancies within a video frame by employing transform coding as in still image coding, therefore, the frames can be encoded independently. This type of compression is ideal when the difference between two consecutive video frames is very large. Intra-coded pictures are also referred to as I-pictures in MPEG-2. Inter-frame compression takes advantage of temporal redundancies by coding the difference between a predicted frame and the current video frame instead of encoding the current frame itself. A forward predicted inter-frame is also referred to as a P-picture while a bi-directionally predicted inter-frame is referred to as a B-picture. The combination of intra and inter-frame coding is called hybrid coding [23]. The generic hybrid video coder is shown in Figure 12.

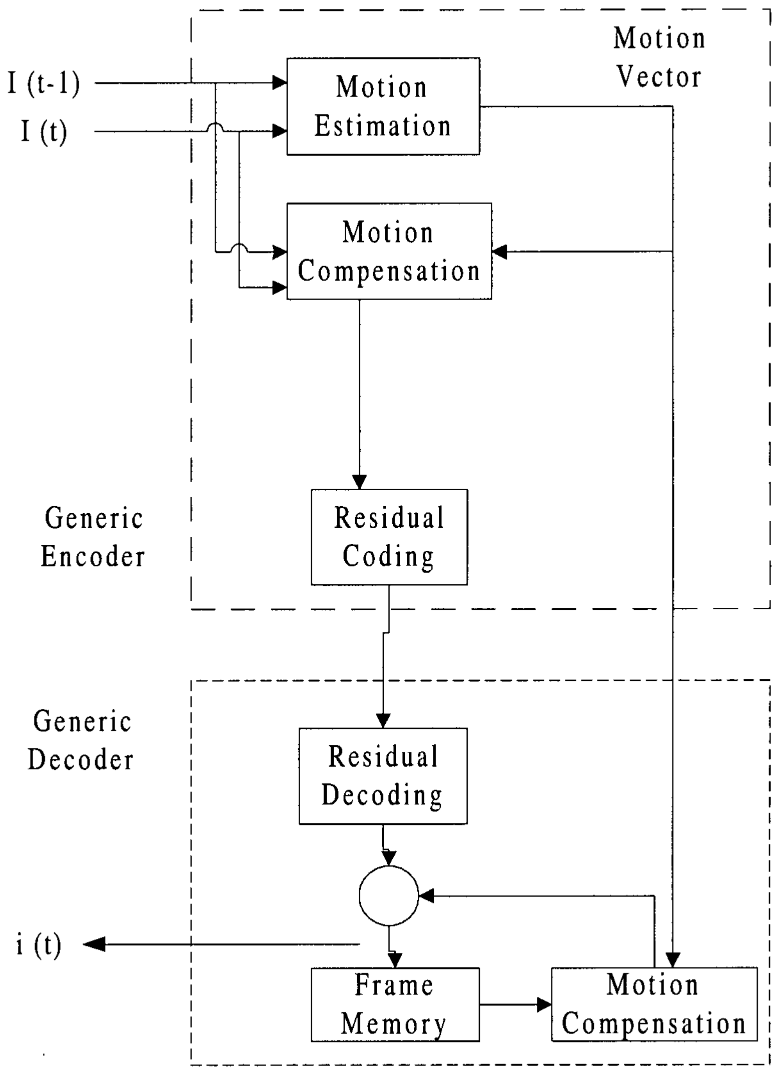


Figure 12: Generic hybrid video coder and decoder.

3.2.1 Motion Estimation and Motion Compensation

In general, in order to achieve a good prediction of the picture currently being encoded, the motion of the objects in the video sequence must be considered. Since if the objects in present picture are well predicted from the previous picture, then the amount of information needed to be encoded is relatively small given that the difference between pictures is small. Therefore, the introduction of motion compensation (MC) is needed. Motion compensation is used to exploit the temporal redundancy inherent in moving pictures. It is used for the compression of P-pictures and B-pictures. The idea behind this is to reduce the information needed for the storage of the difference between the predicted block and the current block along with the motion vector associated with the current block, than would be required for storage of the current pixel data. Motion vector is the relative distance between the best-matched block in the previous picture and the current block. There are two distinct types of motion compensation. The first is called forward motion compensation, which is the result of coding a frame based on a previous frame in the actual temporal sequence. Figure 13 illustrates forward motion compensation. The other form is symmetric, using a subsequent frame in the sequence to act as a basis frame for prediction.

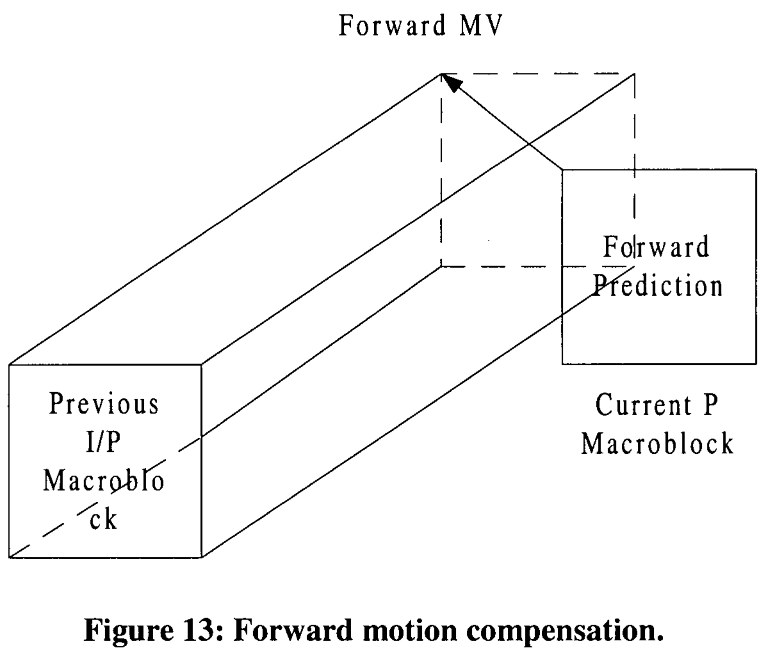


Figure 13: Forward motion compensation.

Block-base motion compensation is considered as one of the best approaches for eliminating temporal redundancies. Since in this method, the picture is divided into blocks with size M x M pixels each. For every individual block, the previous picture is searched for the best match block of the current picture. This searching process is called Motion Estimation (ME) as shown in figure 11 also. As we can see, the computation that searches for the best match block of the entire picture can be very expensive, therefore, ME is usually performed using a limited window size called search window for the search process. Figure 14 shows the full search method with search window size that is $\pm p$ pels in the x-direction and $\pm q$ pels in the y-direction with block size (M, N).

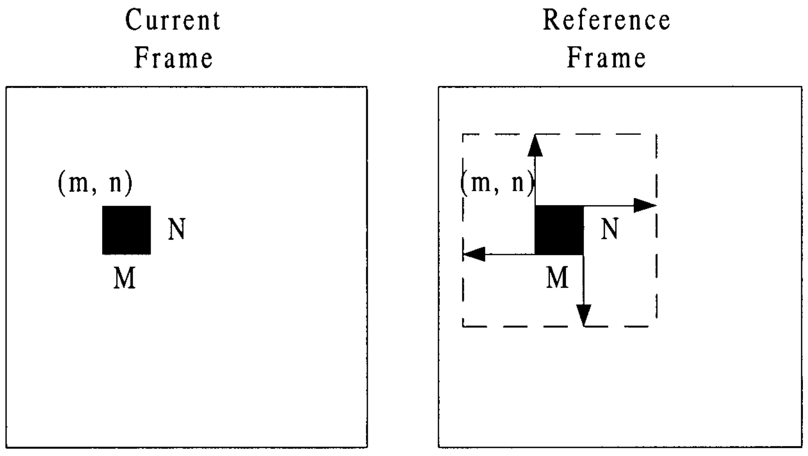


Figure 14: Full search motion estimation process.

The best-block matching algorithm uses a very simple cost function to evaluate the displacement vectors that describe motion in image sequences. In most cases, a simple pel-based absolute difference is used to determine the similarity between blocks. Two of the most popular cost functions used to measure the similarity between blocks are the mean square error (MSE) and the mean absolute error (MAE). For a block at location (x_m, y_n) with block size M x N, the MSE and MAE for motion displace **d** = (d_x, d_y) are,

$$MSE(d_x, d_y) = \frac{1}{MN} \sum_{x=xm}^{xm+M} \sum_{y=yn}^{yn+N} (I(x,y) - I_{ref}(x+d_x, y+d_y))^2, \qquad M = N, \qquad \textbf{(1)}$$

$$MAE(d_x, d_y) = \frac{1}{MN} \sum_{x=xm}^{xm+M} \sum_{y=yn}^{yn+N} \left| I(x,y) - I_{ref}(x+d_x, y+d_y) \right|, \qquad M = N, \qquad \textbf{(2)}$$

where I and I_{ref} are the pixels of the present frame and the reference frame, respectively. The M and N are the dimensions of the block and $I_{ref}(x + d_x, y + d_y)$ is the value of the block element that is located in row $x + d_x$ and column $y + d_y$ in the reference frame. The difference in subjective performance between using the MSE and MAE is quite small for most video sequences. Hence, the number of operations necessary for calculating MAE is much smaller than that of MSE, therefore, in general, the MAE error measure is more commonly used. However, even using the MAE, the computational complexity is still quite high for most video sequences, simply because hundreds of reference blocks must be considered for each candidate block. For example, for CCIR-601 video (30 frames/sec, 720x480 pixels/frame) with a (±16, ±16) search window size, a 30 GOPS (giga-operations per second)) capable processor is required for performing full search block matching algorithm in real-time.

As we have already suggested, motion estimation and motion compensation are the cornerstone of most video coding systems presently in vogue and are the basic mechanisms by which temporal redundancies are captured in the current H.261, H.263, MPEG-1, and MPEG-2 standards. Therefore, the importance of exploiting an efficient algorithm for the two tasks is essential. There are many existing efficient algorithms such as logarithmic search and hierarchical search have been suggested in order to reduce the number of operations required to find the best matching block. Later in this chapter, we present a fast ME search algorithm developed in our laboratory at UBC.

3.2.2 Transform Coding, Quantization and Variable Length Coding

After performing the motion estimation, the difference block between the best matching block and the current block is computed. Then a transform operation is applied to the different blocks to reduce the spatial redundancies. Transform coding does a reasonably good job of exploiting statistical redundancy in images by de-correlating the pel data and by compacting information into the lower order coefficients. The most commonly used coding algorithm is the Discrete Cosine Transform (DCT), an algorithm that greatly de-correlates signals and has a smaller complexity compared to that of other transforms with similar performance. The definition of the 2-D DCT transform for an 8x8 block is given by

$$y_{kl} = \frac{c_k c_l}{4} \sum_{i=0}^{7} \sum_{j=0}^{7} x_{ij} \cos\left(\frac{(2i+1)k\pi}{16}\right) \cos\left(\frac{(2j+1)l\pi}{16}\right) \qquad \textbf{(3)}$$

where k, l = 0, ..., 7 and

$$c_k = \begin{cases} \frac{1}{\sqrt{2}} & : \; k = 0 \\ 1 & : \; k \neq 0 \end{cases}$$

Following the transform coding, resulting coefficients are quantized and the quantized DCT coefficients, motion vectors, and side information, such as the picture coding type and the quantizer step are entropy coded using the Variable Length Codes (VLCs). For the quantization, the coding method is lossy coding while VLC coding is lossless. After the quantization, the coefficient block is reconstructed by dequantization and application of Inverse Discrete Cosine Transform (IDCT) and feed back to the coding loop to be used for prediction of temporally future blocks. The original 8x8 block of pixels can be recovered using an 8x8 inverse DCT (IDCT) as follows:

$$x_{kl} = \sum_{i=0}^{7}\sum_{j=0}^{7} y_{ij} c_k \cos\left(\frac{\pi(2i+1)k}{16}\right) c_l \cos\left(\frac{\pi(2j+1)l}{16}\right) \tag{4}$$

where k, l = 0, …, 7 and

$$ck,l = \begin{cases} \sqrt{\frac{1}{8}} & : \quad k = 0 \\ \sqrt{\frac{1}{4}} & : \quad k \neq 0 \end{cases}$$

Although exact reconstruction can be theoretically achieved, it is often not possible using finite-precision arithmetic. While forward DCT errors can be tolerated, inverse DCT errors must meet the MPEG-2 standard if compliance is to be achieved.

3.3 The MPEG-2 Video Coding Standard

The objective of MPEG-2 video coding is to provide high quality and multi-channel compressed video signals transmission over limited-capacity broadcasting infrastructure such as Cable/HFC and ATM networks. Specifically, MPEG-2 was given the mandate of providing a video quality no less than NTSC/PAL and up to CCIR 601 quality with a target bit rates in the range 2 to 10 Mbit/s [24]. The MPEG-2 coding standard only specifies the syntax for encoded bit-stream, and many of the complex encoding decisions are to provide maximum flexibility for implementing compliant video codecs. This flexibility has posed quite a challenge for video coding engineers to design and determine the trade-offs between coding performance (high compression with acceptable quality) and implementation complexity of various MPEG-2 video coding schemes.

Like H.261 [25], H.263 [26], and MPEG-1 [27], the MPEG-2 video standard is based on motion compensated prediction and DCT residual coding. However, MPEG-2 offers a more efficient means to code interlaced video signals and supports scalable video coding. The readers should be aware that MPEG-2 is a very complex standard. The description we provide here is only an overview of its video part. The MPEG specification is intended to be generic in the sense that it serves a wide range of applications. For example, coded MPEG-2 bit rates of up to 400 Gigabits/s and picture sizes up to 16,000 by 16,000 pixels can be defined. In order to cope with the great variety of video applications within one coding standard, MPEG-2 adopts a "toolkit-like" approach; that is, MPEG-2 is a collection of tools which satisfies the requirements of specific major applications. The range of coding support provided by MPEG-2 is divided into profiles and levels. For each profile/level, MPEG-2 provides the syntax for the coded bit stream and the decoding requirements.

A profile is a defined subset of the entire bit stream syntax specified by MPEG-2. The MPEG-2 profiles can be divided into two categories: non-scalable and scalable. The two non-scalable profiles are the Simple and Main profiles. The three scalable profiles are the SNR scalable, Spatially scalable, and High profiles. Within a profile, a level is defined as a set of constraints imposed on the parameters of the bit stream. For each profile, the four levels are the Low, Main, High-1440, and High levels. The constraints of the coding parameters for each level of a profile and the various profiles are shown in table 1 and table 2, respectively. Among its profile and level combinations, the most widely used combination in the MPEG-2 industry is the main profile/main level.

Level	Parameters
HIGH	1920 x 1152 60 frames/s 80 Mbit/s
HIGH-1440	1440 x 1152 60 frames/s 60 Mbit/s
MAIN	720 x 576 30 frames/s 15 Mbit/s
LOW	352 x 288 30 frames/s 4 Mbit/s

Table 1: Constraint parameters at each level of a profile.

Profile	Algorithms
HIGH	4:2:2 – YUV I, P, B
SPATIAL Scalable	4:0:0 – YUV I, P, B
SNR Scalable	4:2:0 – YUV I, P, B
MAIN	4:2:0 – YUV I, P, B
SIMPLE	4:2:0 – YUV I, P

Table 2: Algorithms and functionalities supported with each profile.

3.3.1 Bit Stream and Macroblock Layer

In MPEG-2, the structure of the bit stream syntax depends on the profile adapted by the application. As we will concentrate on only the main profile/main level part of the standard, we will only describe the bit stream syntax for the main profile. The MPEG-2 video bit stream consists of six hierarchical layers starting at the block layer (composed of 8x8 pels) followed by the macroblock, slice, picture, group of pictures (GOP), and the sequence layers as shown in Figure 15.

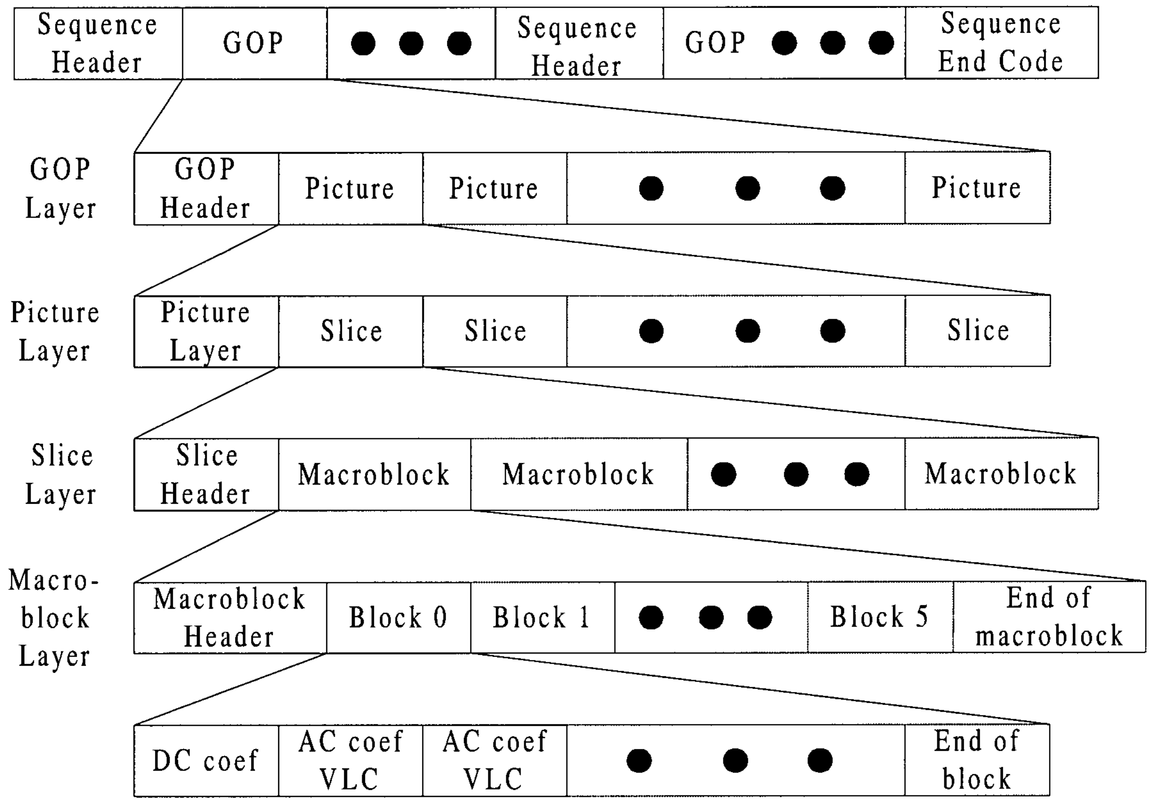


Figure 15: Video sequences layer.

The GOP level is where the video sequence determines the ratio of I, P and B frames, and where the most relevant work on temporal picture structure is carried out. In a typical GOP size of 12 interlaced video frames, the first frame (I-picture) is intra coded, and the following 11 frames are inter coded using alternatively the forward motion compensation (P-pictures) and bi-directional motion compensation (B-pictures) as shown in Figure 16.

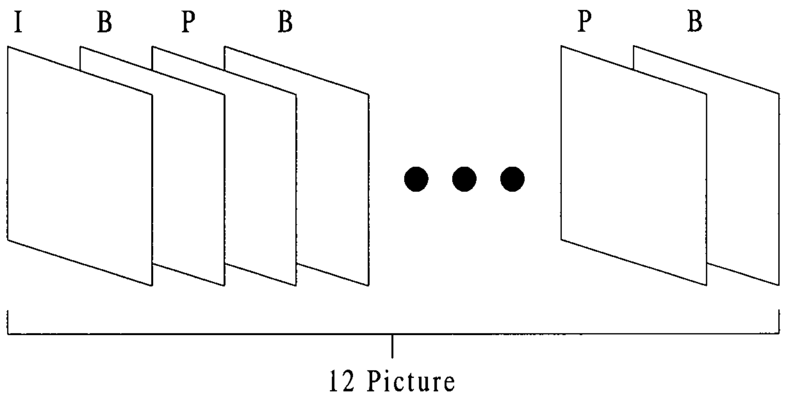


Figure 16: GOP of 12 frames.

MPEG-2 video supports different standardized picture formats such as CCIR-601 (720x480 resolution) and SIF (320x240 resolution), etc. Each input picture from the video sequence is divided into macroblocks, and every macroblock (MB) consists of a 16x16 block (or four 8x8 blocks) of luminance (Y) component and two 8x8 blocks of chrominance (C*b* and C*r*) components. Figure 17 shows the MPEG-2 picture structure at CCIR-601 resolution.

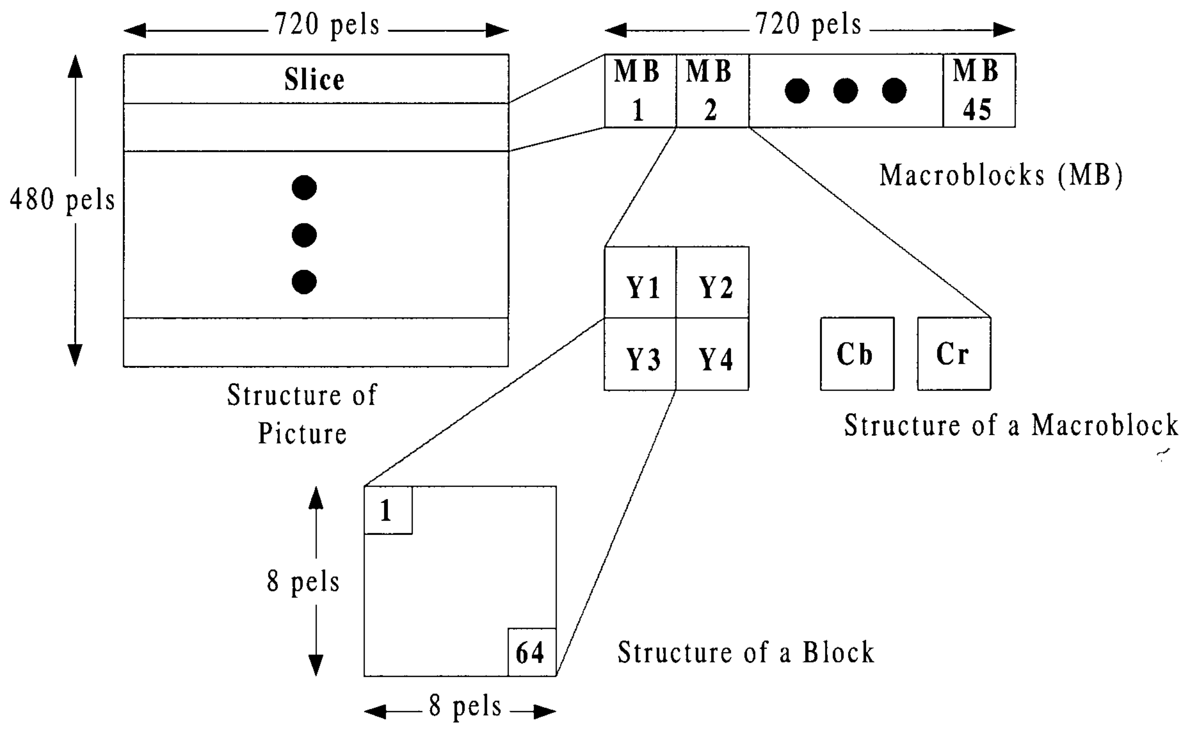


Figure 17: MPEG-2 picture structure at CCIR 601 resolution.

3.3.2 Baseline MPEG-2 Video Encoder

As shown previously, each picture in the input video sequence is divided into macroblocks, and the motion compensated prediction operation is performed at the macroblock level, and most of the operation is perform at that level. In MPEG-2 encoding, a high percentage of the pictures are *inter* coded to obtain a higher compression ratio, except for the first picture of each GOP, which is *intra* coded. Therefore, the *inter* coding operations dominated most of the encoding computations time. Figure 18 shows the encoding diagram for the MPEG-2 baseline encoder.

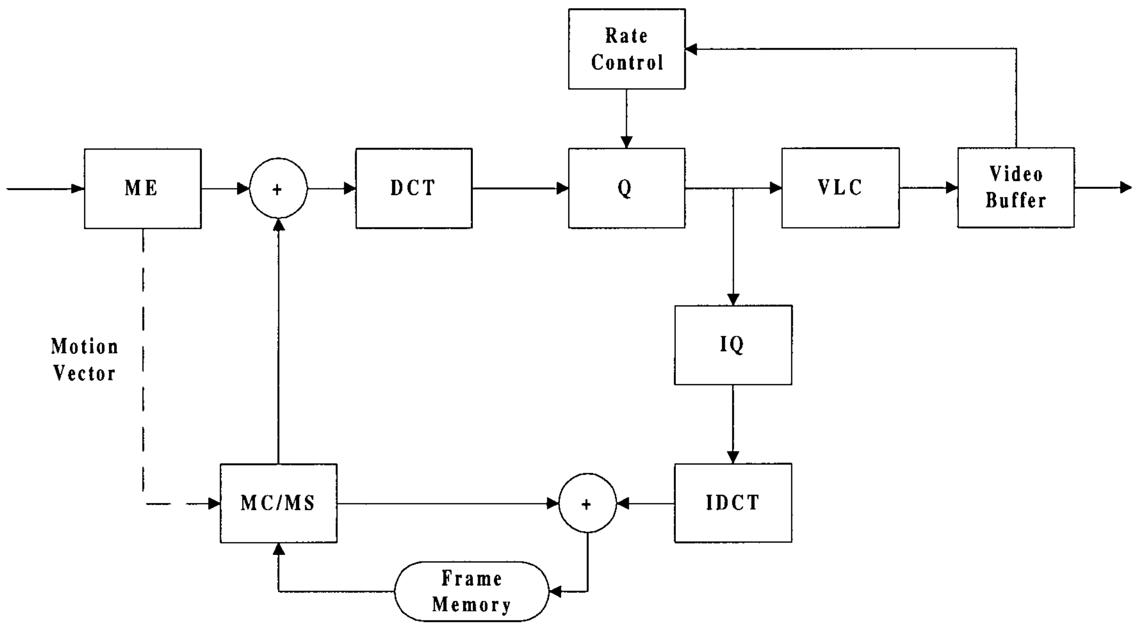


Figure 18: Block diagram of the MPEG-2 encoder.

For the basic video coding algorithm, the MPEG-2 algorithm has features similar to the ITU-T H.261 and the MPEG-1 algorithm. There are six key video coding components on the MPEG-2 video encoder, which includes motion estimation and motion compensation (ME/MC), mode selection (MS), transform coding (DCT/IDCT), quantization (Q/IQ), variable length coding (VLC), and rate control (RC). For intra coding mode, motion estimation is not required to be performed on the picture, i.e., the 8x8 luminance and chrominance blocks are transform coded with two dimensional DCT and then the transform coefficients are quantized and VLC coded. For the *inter* coding operation, motion estimation searches for the best motion vector with reference to the pervious frame and performs motion-compensation prediction to reduce the temporal redundancies. Then transform coding (using discrete cosine transform (DCT) algorithm) encodes the motion-compensated prediction of the difference frame to reduce the spatial redundancies. Following transform coding, the resulting coefficients are quantized and then the quantized DCT coefficients, the macroblock (MB) coding mode, the quantization step, the frame/field motion vectors, and the residuals are finally variable-length encoded.

For the motion estimation, full integer pixel motion estimation is preformed first, and then the motion search continues with a half-pixel search around the current motion vector position. Half-pixel values are found using the eight neighbors of the best full-pixel motion vector as shown in Figure 19.

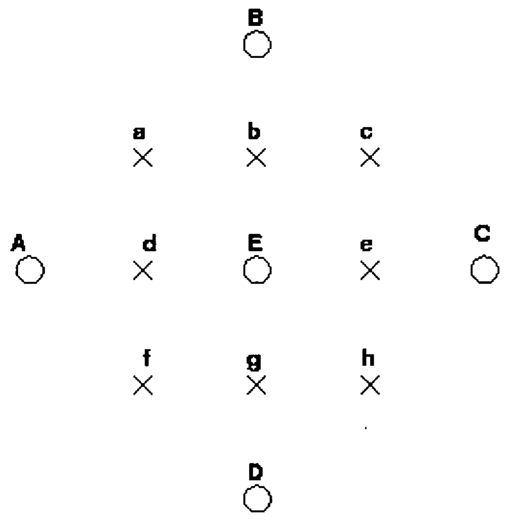


Figure 19: The half pixel motion estimation.

Similar to a Differential Pulse Code Modulator (DPCM), MPEG-2 decodes and reconstructs the image in the encoder. Then the reconstructed picture is added to the current predicted picture in order to obtain the decoded picture, which is then stored in frame memory.

3.3.3 MPGE-2 Video Decoder

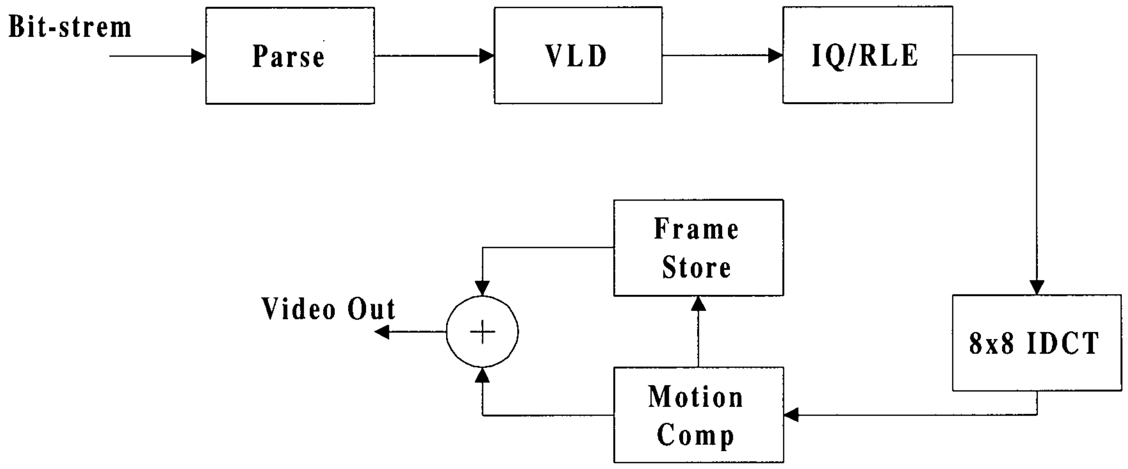


Figure 20: Block diagram for the MPEG-2 decoder.

Figure 20 shows a block diagram of the MPEG-2 video decoder. The main functions involved in the decoder are bit stream parsing, variable length decoding (VLD), inverse quantization (IQ) and run length expansion (RLE), inverse discrete cosine transform (IDCT), and motion compensation. First, the bit stream is parsed and variable-length decoded to obtain the coefficients, the motion vectors and other side information. After the VLD outputs are run-level pairs in a zigzag scan order. The next step is to expand the zero runs, quantize the level values and write the result in a row major scan order. After the coefficients are decoded by inverse quantization, IDCT is applied to the coefficient blocks. If the block is *intra* coded, the reconstructed block is equal to the result of the inverse transformation. For *inter* coded blocks, the reconstruction is formed by motion compensation, i.e. summing the predicted block and the inverse transformed block.

3.3.4 Forward and Inverse Quantization

Depending on the prediction method selected at the encoder, coefficients that are going to be quantized can have a wide variation of statistics. In the case of intra prediction, the first elements of the coefficient block, called the DC coefficient, takes much larger value than the rest of the coefficients, called the AC coefficients. If the block is inter (non-intra) coded, then all the coefficients take smaller AC coefficients. In order to efficiently quantize intra coded blocks, one quantizer is used for the intra DC coefficient and one of 31 predefined quantizers is used for the AC coefficients.

Although the decision levels of the quantizers are not defined within the standard, it is suggested that the quantizer for the DC coefficients be a uniform quantizer with a step size equal to 8. Each of the other 31 quantizers use different spaced reconstruction levels with a non-uniform step size. For the blocks that are inter coded, the AC coefficients have a uniform step size. The sign of the quantized transform coefficients is signaled at the end of the VLC code word. After inverse quantization, the reconstruction levels of all coefficients other than the DC coefficient are clipped to the range -2048 to 2047.

3.3.5 VLC Coding of Transform Coefficients

After quantization, the lowest DCT coefficient (DC coefficient) is treated differently from the remaining AC coefficients when performing the VLC coding of transform coefficients. Since the human visual system is more sensitive to blocking artifacts, which are mainly due to the DC coefficients, the DC coefficient is treated separately from the other 63 coefficients. The DC coefficient corresponds to the average intensity of the component block and is encoded using a differential DC prediction method. The coding employs a first order predictor which is given by

$$DIFF = DC_i - DC_{i-1}, \qquad (5)$$

where DC_i is the DC coefficient of the current block and DC_{i-1} is the DC coefficient of the previous block in row scan order. Most of the remaining AC values that represent higher frequencies become zero. In order to exploit this before VLC coding, the 63 nonzero quantizer values of the remaining DCT coefficients and their locations are the scanned in a zig-zag scan order as illustrated in Figure 21. Then, they are run-length entropy coded using variable length code (VLC) tables.

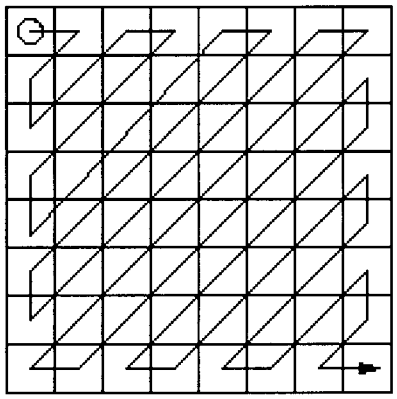


Figure 21: Zig-zag scan order of a 8x8 block.

The scanning of the quantized DCT domain 2-dimensional signal followed by variable-length code-word assignment for the coefficients serves as a mapping of the 2-dimensional image signal into a 1-dimensional bit stream. The rearrangement places the

DC coefficient first in the array, and the remaining AC coefficients are ordered from low to high frequency. The nonzero AC coefficient quantizer values called *levels* are detected along the scan line as well as the distance called *run* between two consecutive nonzero coefficients. Each consecutive *run* and *level* pair is encoded by transmitting only one VLC code word. Variable length codes are not defined for all the combinations of *level* and *run*. If a VLC is not found for a specific combination, the combination is coded by a 6 bit *escape* code followed by 6 bit *run* and 12 bit *level* codes. Other information such as prediction types and quantizer indication are also entropy encoded by means of VLC's.

3.4 Proposed Software Solution for Real-Time MPEG-2 Video Encoder at UBC

The starting point of a software-only real-time MPEG-2 video encoder on the C62x VLIW processor, is the development of efficient code that can be efficiently compiled on the C62x. Extensive work has been carried out on software MPEG-2 video encoder to improve and to implement an efficient real-time MPEG-2 video coding algorithm. Over the past two years, the Signal Processing and Multimedia Group at the University of British Columbia, has carried out extensive work towards developing software-only MPEG-2 compliant video encoders. We have mainly targeted general-purpose processor platforms such Intel's Pentium II and III, and VLIW DSPs such as Texas Instrument's C62x. The computational complexity reduction in our MPEG-2 software implementation is achieved by employing several advanced techniques which are briefly described in the following subsections. More detailed description of the algorithms can be found in [28],

[29], and [30]. The macroblock layer has been the major focus of our optimization since this is where coding gains are obtained and where most of the computational load resides.

3.4.1 Motion Estimation (ME)

As we have discussed previously, motion estimation is considered as one of the most time consuming tasks to perform in block-based video encoding, as it involves searching for the best match between the current block and candidate blocks in a confined search windows of the previous encoded frame. The location of the best match block is the estimated motion vector and the motion vector is computed using the most widely known matching function -- sum of absolute differences (SAD). The sum of absolute differences SAD is defined by

$$SAD = \sum_{k=1}^{16}\sum_{l=1}^{16}\left|B_{i,j}(k,l) - B_{i-u,j-v}(k,l)\right| \qquad \textbf{(5)}$$

where $B_{i,j}(k, l)$ represents the (k, l)th pixel of a 16x16 macroblock from the current picture at the spatial location (i, j), and $B_{i-u,\ j-v}(k, l)$ represents the (k, l)th pixel of a candidate macroblock from a reference picture at the spatial location (i, j) displaced by the vector (u, v). To find the macroblock producing the minimum mismatch error, we need to calculate the SAD at several locations within a search window. The simplest, but the most compute-intensive search method, known as the full search or exhaustive search method, evaluates the SAD at every possible pixel location in the search area. Therefore, the main objective of ME is finding an optimal motion vector in a limited search window using a minimal number of SAD computations while not scarifying the video quality.

Several algorithms proposed by the SPMG group at UBC that restrict the search to a few points are employed, and will be discussed in the following subsections.

3.4.1.1 Optimum Fast Block Matching Strategy

As the search range becomes fairly large for large picture sizes or sequences with fast motion, the exhaustive search computational complexity grows very rapidly. Therefore, the proposed fast ME algorithms break up the search process into a few sequential steps, and each step is based on searching the candidate points located on diamond-shape contours. The set of points to be searched at the next step is determined by the current-step result. There are two versions of the matching strategies: fixed-center diamond search, and floating-center diamond search. Figures 22 a) and b) illustrate both fixed-center and floating-center diamond search matching strategies, respectively.

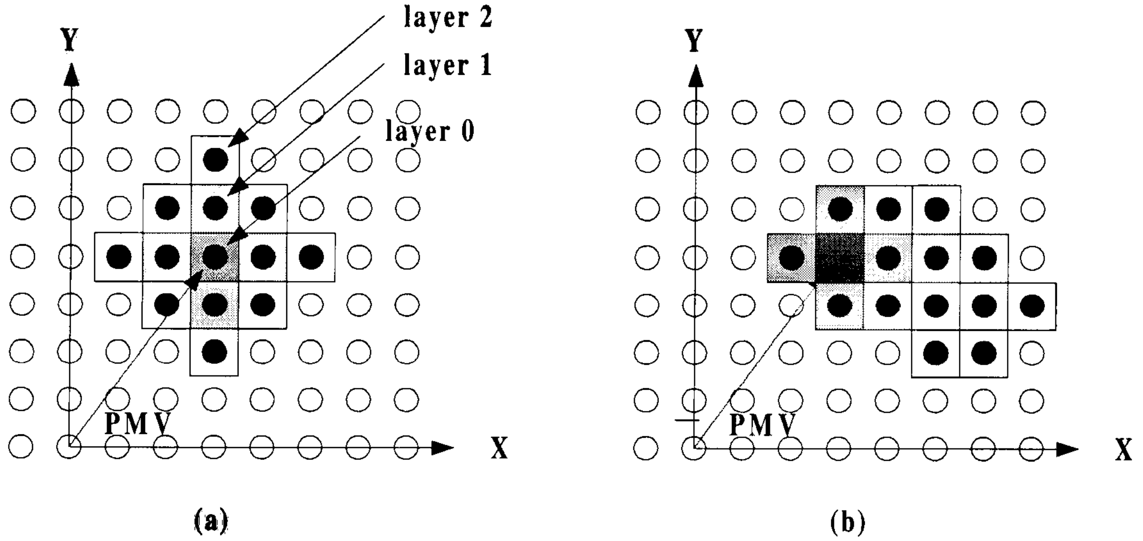


Figure 22: (a) Fixed-center, (b) floating-center diamond shape motion vector search area.

The fixed-center diamond search algorithm searches for the best matching macroblock in a diamond-shape area (layer). For each layer, the center is fixed and the size keeps increasing until it covers the entire rectangular search window when the number of motion vector candidates increase as the diamond area expands. However, instead of having fixed center, the floating-center algorithm allows the center to move to the best matching point of the previous step. For every step, the size of the diamond shape is kept constant and contains only the four immediate neighbors of the previous search center. The search stops when the current search area is outside the allowed search region or when the SAD corresponding to the best candidate in the search area of the current step is larger than the SAD of the best candidate at the previous step. In order to further reduce the computation an earlier stop criteria, called distortion-computation optimized ME (DCME) is used (see [29]). Experiments have shown that the number of SAD operations is reduced by 50% and the actual number of computations is reduced by a factor ranging from 100:1 to 250:1 compared to the full-search ME [29].

3.4.1.2 Hierarchical Block Matching Strategy

The disadvantage of the two algorithms presented above is that they are often trapped in local minima due to the implicit assumption of a monotonically changing matching function. In order to solve the problem while reducing the computation, a hierarchical search algorithm reduces the computation by utilizing lower resolution of the current and reference frames. This is done by downsampling both the current and reference frames by a factor of two in both directions. While the search step is preformed at the lower resolution of the images and the macroblocks, one half of their original search area is performed with either full-search or fast ME search algorithms. Experiments have shown that the hierarchical algorithm performs very close to its single-resolution counterpart for both the fixed and the floating center strategies while the SAD computation required is reduced by 30% or more depending on the video sequences used [29].

3.4.1.3 Additional Stopping Criterion

The number of computations associated with motion estimation can be further reduced by allowing the algorithm to terminate even earlier using a prediction method that detects zero macroblocks. Throughout the search process, if the current best match macroblock generates a prediction error block that is most likely to be set to zero during quantization, then the motion search is terminated [29]. The reason behind this is that any future candidate motion vector will produce a zero quantized block as well.

The idea behind this is whenever a candidate is determined to be the minimum so far, its minimum distortion average over the pixels in the block is compared against an adaptive threshold. If the value of the distortion is less than the threshold then the motion search is terminated. Experimental results show that a reduction in computations by a factor of up to three can be achieved with a little loss in video quality.

3.4.1.4 The Optimum Block Matching Function

The encoder uses a standard procedure called partial distortion technique to reduce the computations associated with SAD operation. Since during motion estimation we are only interested in candidates whose SAD is smaller than the current minimum SAD. Therefore, as soon as the running SAD of a candidate grows larger than the previous minimum SAD, the candidate can be discarded before completing the computation. Given this, the unnecessary operations are avoided at the cost of additional comparisons. Comparing the running SAD with the current minimum SAD after each pixel difference may offset the gain achieved by reducing the number of subtractions, absolute value operations, and additions. However, comparison done on each pixel is rather computational expensive, thus, comparison is made only after processing a complete row of 16 pixels in the macroblock.

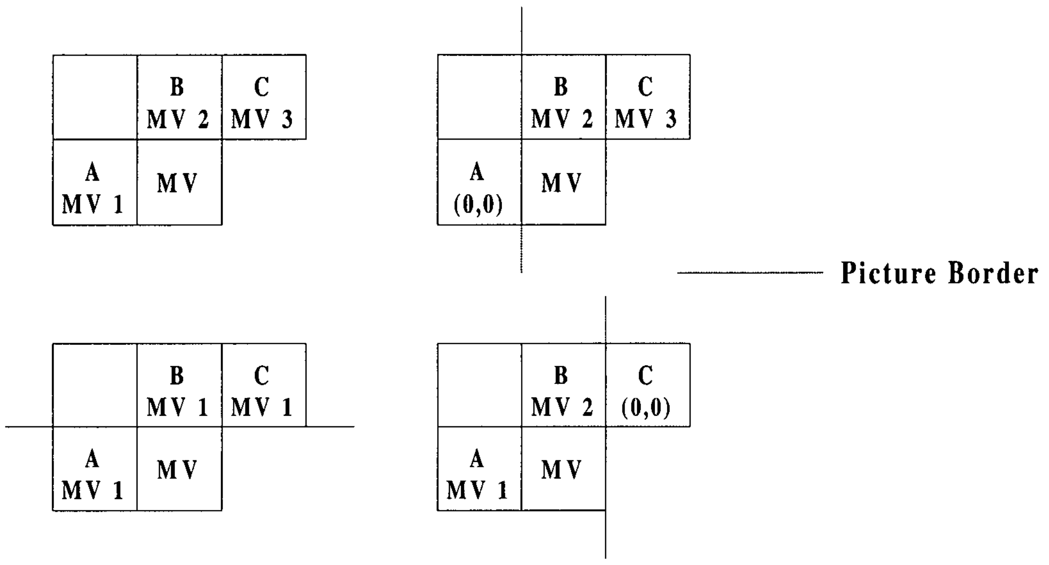
In order to apply the partial SAD computation in the context of the distortion-computation optimized search, the Lagrangian used for the motion vector search termination criterion is computed after each row *n*:

$$J_\beta(n) = SAD(n) + \beta C(n), \tag{6}$$

where *SAD(n)* is the running SAD after the *n*th row is processed, and *C(n)* is the running number of operations executed since motion estimation started for the current macroblock. For each candidate, C(0) is initialized as the number of operations performed before considering the candidate. The Lagrangian J_β^n is then compared to the current minimum Lagrangian J_β^*. If $J_\beta^n \geq J_\beta^*$, the candidate motion vector is no longer considered and the SAD computation is stopped. Otherwise, the process continues until the current Lagrangian is larger than the minimum one or the complete SAD has been computed. If the latter case is true, the current Lagrangian replaces the minimum Lagrangian J_β^*. The results of using the partial distortion have shown that the reduction in computations ranges between 50% to 60% for the full search ME and only 15% to 30% for the other ME algorithms [29].

3.4.1.5 Robust Motion Vector Prediction

A reliability measure of the predicted motion vector (PMV) has been incorporated to access the PMV before using it as a center for motion search. The performance of any fast motion algorithm is directly affected by the accuracy of the predicted motion vector. If the predicted motion vector is far from the best-match motion vector, the fast ME search algorithm will then be quickly trapped into a local minimum, or the algorithm will perform too many computations to find the best-match motion vector. Therefore, we use a predictor which is the median of three previously coded motion vectors corresponding to the macroblock to the left (A), above (B) and above-right (C) as illustrated in Figure 23.



MV: Current Motion Vector
MV1, MV2, MV3: Three motion vector predictor that are used to predict MV

Figure 23: Three predictor of motion vector prediction from surrounding macroblock.

The motion vectors corresponding to these MBs are considered as candidates for the PMV. The block matching function is then computed for all candidates MVs as applied to the current MB. Finally, the PMV is set to be equal to the candidate MV which yields the best match. If the PMV was unreliable, then the fast diamond search will be abandon and a full search will commence in the lower resolution image. By using different MV predictor and the reliability measures, the PSNR of fast motion sequences is improved significantly while the computational demands are also reduced slightly [31].

3.4.2 Rate-Distortion Optimized Mode Selection

The proposed encoder obtains improved rate-distortion (RD) performance by employing a rate-distortion based macroblock mode selection. The coding mode is chosen to minimize the RD Lagrangian as the RD-based mode selection leads to substantial performance improvements. However, from the view of optimal RD performance, having an exhaustive Lagrangian minimization is computationally very expensive. Therefore, an efficient rate-distortion optimized mode search algorithm is used to attain a close-to-optimal level of compression performance while maintaining the number of computations low by exploiting the statistical redundancies in the video sequence [29].

3.4.3 Combination of Discrete Cosine Transform (DCT) and Quantization

After the macroblock coding mode is determined, the chosen motion compensation and block transform are applied, the transform coefficients are quantized, and the MB header, motion vector, and quantized DCT coefficients are variable-length encoded. For the hybrid macroblock coding modes, both the DCT and quantization need to be performed. By merging the DCT and quantization into single operation, known as quantized-DCT or QDCT, the number of computations can be substantially reduced. QDCT is performed only during inter-frame encoding leading to an order of magnitude reduction in computations. An early Prediction technique has been incorporated into the QDCT algorithm to further reduce the computations required for the normal QDCT [32]. The integer implementation of QDCT is especially efficient to run on the C62x platform since it uses fixed-point arithmetic.

3.5 Analysis of the proposed MPEG-2 video encoder

Timing analysis is critical to identifying the most computational intensive components of the MPEG-2 encoder which need substantial optimization. Table 3 shows the relative weights of the most compute-intensive MPEG-2 encoding functions. Clearly, motion-estimation is still the most computationally demanding algorithm.

Function	Computational Load
Quantization	4.2%
DCT	5.1%
IDCT	5.3%
ME (SAD Computation)	50.1%
Motion Compensation	4.4%
MSE	1.2%
VLC	10.3%

Table 3: Relative computational costs of the most compute intensive tasks of the MPEG-2 video encoder.

We have identified the set of components that are computationally intensive but which potentially can be optimized efficiently on the C62x platform by exploiting the instruction-level parallelism inherent in the algorithm. The main components optimized for the C6x encoder implementation include the SAD calculation, DCT/IDCT, Quantization, QDCT, mean square error (MSE) calculation, and variable length encoding (VLC).

Chapter 4

Mapping the MPEG-2 Software Encoder on the Texas Instrument's TMS3206201 DSP Platform

4.1 Introduction

After an extensive research on the available media processors and VLIW DSPs, we have decided to use Texas Instrument's 1600 MIPS TMS320C6201 VLIW DSP for our initial test bed. Texas Instrument's TMS320C6201 VLIW DSP is representative of a new class of DSPs that exploit explicit instruction-level parallelism (ILP) and advanced compiler techniques to provide thousands of MIPS to potential applications. Among the other architectures, the C62x VelociTI VLIW architecture offers the highest probability of achieving the software real-time MPEG-2 video encoding.

In this chapter, we will first give an overview of the TMS320C6x CPU architecture, and some brief information on its data path and instructions set. Then, the memory structure of the C62x evaluation model (EVM) board that we used for our implementation is also described. Afterwards, the design flow and implementation of the MPEG-2 video encoder software on the C62x EVM board is discussed. The memory mapping of certain video encoder functions are also discussed along with the implementation issues. This chapter concludes with the implementation bottlenecks of the MPEG-2 video encoder software on the C62x EVM board such as the memory issues, optimized compiler, and instruction latency.

4.2 Texas Instrument's TMS320C6201 CPU Architecture

The TMS320C6x VelociTI CPU architecture was the first off-the-shell DSP to use an enhancement of the traditional VLIW to achieve high performance through increased instruction-level parallelism. A traditional VLIW architecture consists of multiple execution units running in parallel to perform multiple instructions during a single clock cycle. Parallelism is the key to achieving high performance gains. VelociTI is a highly deterministic architecture, with few restrictions on how or when instructions are fetched, executed, or stored. This architectural flexibility is key to the breakthrough efficiency levels of the C62x compiler.

4.2.1 VelociTI Principles

The VelociTI CPU architecture is based on eight principles, underlying the C62x CPU, which focus on increasing DSP performance while maintaining ease of programming and reducing application development time by allowing creation of a high-performance compiler.

4.2.1.1 Parallelism

The VelociTI architecture allows parallel fetch, decode, and execution of multiple instructions that compose the VLIW instruction word. During each execution, each instruction is performed on a single function unit. For the C62x, there are eight 32-bit instructions supplying control for eight independent functional units.

4.2.1.2 Pipelining of Critical Speedpaths

The VelociTI architecture sets the simplest of CPU instructions to determine the cycle time for the processor. The critical path for C62x is the time for a register-to-register ALU operation such as an ADD instruction, and more complex instructions, such as multiply, which are implemented with a one-cycle latency. In order to access the high-performance synchronous on-chip memory, instruction fetch and data access are performed in multiple pipeline stages. The pipelining allows the C62x CPU to operate at 200 MHz or 1600 MIPS [33].

4.2.1.3 Reduced Instruction Set Computer (RISC)

The VelociTI instruction set consists of simple, atomic, and completely independent instructions. The DSP algorithm performance results from program compilation techniques such as software pipelining and loop unrolling. The RISC architecture provides ease of CPU design while maintaining flexibility for high-performance algorithms.

4.2.1.4 Load-Store Architecture

The VelociTI is a load-store architecture in which the memory operations have been decoupled from the arithmetic operations. It also lowers the number of data fetches from a particular algorithm and thus lowers the CPU power consumption.

4.2.1.5 Orthogonality

The most frequent instructions can be executed on the largest number of function units. In the C62x, the CPU is divided into two identical data paths. Thus, every instruction can execute on at least two functional units. The most frequent instructions such as ADD and SUB can execute on six functional units. Like most of the other instruction set, the register file is highly orthogonal, where any register can be an operand to any instruction or any type of functional unit.

4.2.1.6 Determinism

The VelociTI pipeline is unprotected and is thus fully exposed to the compiler. Run-time interdependencies such as pipeline interlocks between phases used in other DSPs are difficult to predict at compile-time. The VelociTI CPU model at compiler time fully reflects the execution and completion order of instructions at run time.

4.2.1.7 Conditional Instructions

Every VelociTI instruction is conditional to efficiently avoid branch latencies.

4.2.1.8 Instruction Packing

VelociTI architecture uses a novel instruction packing technique to achieve code size comparable with scalar RSIC processors.

4.2.2 Data Paths

The C6x VLIW core consists of two symmetrical data paths with four different functional units each and each of the data paths has access to 16 multiported 32-bit registers. The register files have 16 ports which included ten read ports and six write ports. Direct communication between both data paths is provided through the cross path (1X, 2X) to read operands from the other register file. Also, addresses from one data path may be used to load and store values to other data path [34]. Figure 24 shows the block diagram of the C62x CPU data paths.

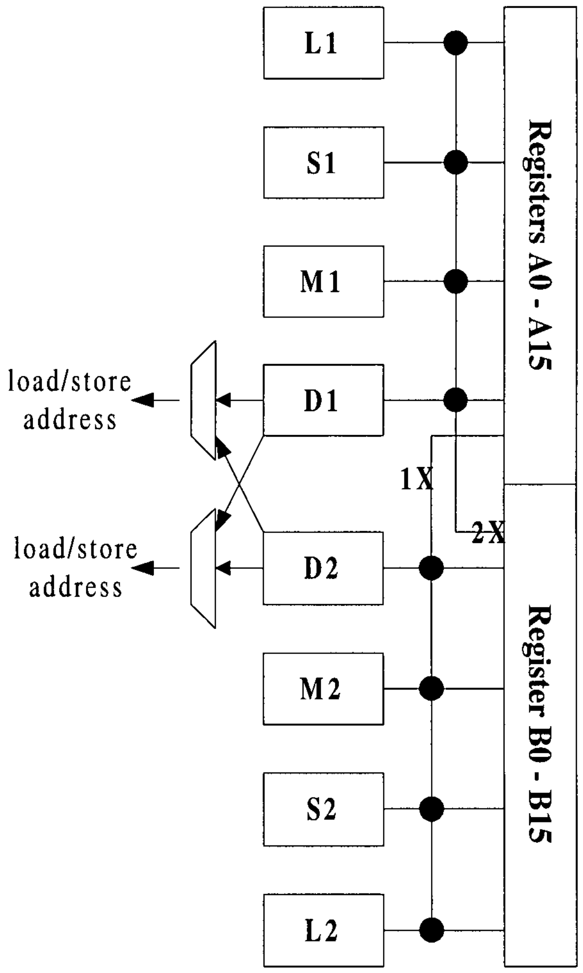


Figure 24: C62x CPU data paths.

4.2.2.1 Functional Units

There are four functional units in each data path where each unit can begin execution of a new instruction every cycle. Subunits on the same functional unit share the same interconnect to the register file to save register porting. Table 4 summarizes subunits mapping to the functional units.

L-unit	S-unit	D-unit	M-unit
Integer Adder	Integer Adder	Integer Adder	Integer Multiplier
Logical	Logical	Load-Store	
Bit Counting	Bit Manipulation		
	Shifting		
	Constant		
	Branch/Control		

Table 4: C62x functional units and subunits.

4.2.2.2 Data Types

In general, the functional units perform 32-bit integer operations. The four functional units of each data path are divided into a logic/arithmetic/compare unit, a shift/arithmetic/branch unit, a 16-bit multiplier unit, and a data address calculation unit as shown in Table 4. For both of the S-Units and L-Units, they can also perform 40-bit integer operations to handle overflow. The integer multipliers in the M-Units generate 32-bit outputs from two 16-bit inputs. In addition, source operands of different types are used, singed (sign extension) or unsigned (zero-filling).

4.2.3 C62x Features and Instruction Set

Texas Instrument's C62x has a total of 38 instructions, and for all the arithmetic calculations, saturation arithmetic is supported.

4.2.3.1 Saturation Arithmetic

One of the important features of C62x instructions is saturation arithmetic. In regular fixed-point arithmetic when an operation overflows or underflows, the most significant bit is lost. For example, the addition of two unsigned 32-bit numbers residing in a 32-bit register may result in an unsigned 33-bit result. Naturally, this number is too large to be represented in a 32-bit register and while the result's low order 32 bits appear in the 32-bit register, the 33rd bit does not fit and therefore lost. There is usually a status flag that shows overflow. Unless a special attention is paid, this kind of behavior may cause serious problems, especially in graphics applications. However, in saturation arithmetic, when such an overflow occurs, instead of generating a 33-bit number and loosing the most significant bit, the corresponding instruction clamps the 33-bit result to the largest possible unsigned number that can be represented by 32 bits, i.e. FFFF FFFF in hexadecimal format. C62x supports two types of saturation arithmetic:

- **Unsigned saturation**: In case of overflow, the register holds the largest number, i.e.2^n-1, where n is the bit number.
- **Signed saturation**: If overflow occurs the register takes 2^{n-1}-1, where n is the bit number

In the C62x architecture, saturation arithmetic is not a mode which may be activated by setting a control bit. But instead, some instructions inherently perform saturation during their operation. For example, some C62x add instruction employ conventional wrap-around arithmetic while others employ saturation arithmetic.

4.2.3.2 C62x Instruction Set

C62x instructions mostly perform 32-bit operations which are packed into a 32-bit register, however, limited support is provided for 16-bit instruction data. The mapping of instructions to functional units is done at code-generation time. In addition, most common integer operations can be mapped onto four to eight functional units, and each functional unit has special-purpose features which are summarized in the following sections.

4.2.3.2.1 Integer Comparison

For integer comparison in C62x, rather than having multiple status bits for each functional unit, the C62x employs explicit comparison instructions that generate a 1 or 0 result in a general-purpose register. Placing the result in a general-purpose register allows the value to be used directly in computation.

4.2.3.2.2 Dual 16-bit Pair Arithmetic

For 16-bit addition and subtraction, there are two instructions ADD2 and SUB2 that add/subtract two 32-bit registers while inhibiting the carry between the 15th and 16th bit positions. These instructions allow increased add/subtract throughput of 16-bit values. Figure 25 illustrate the operation of the ADD2 dual 16-bit pair arithmetic.

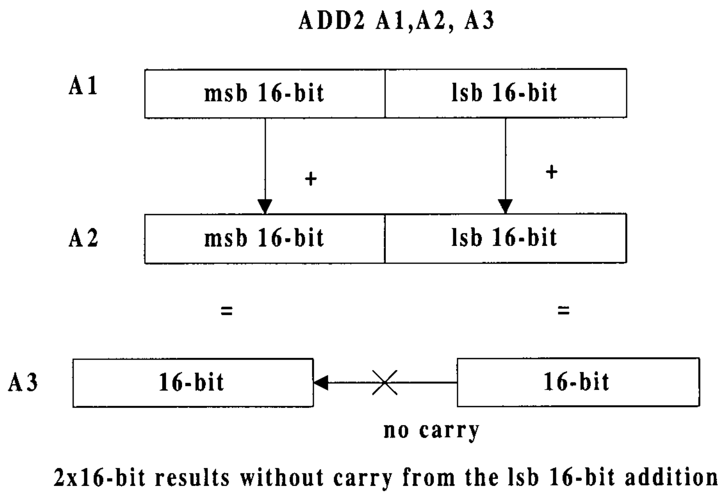


Figure 25: ADD2 dual 16-bit pair arithmetic.

4.2.3.2.3 Memory Data Types

The C62x can load or store bytes, 16-bit half-words, and 32-bit words. Byte and half-word loads are sign-extended (signed) or zero-filled (unsigned). In addition, certain registers have the option of having circular arithmetic performed during address calculation (for circular buffers).

4.2.3.3 Instruction Packing

The C6x CPU has a 256-bit path for internal program access to fetch eight 32-bit instructions every cycle. In typical VLIW architectures, each instruction would correspond to a particular functional unit. If that functional unit were idle on any particular cycle a NOP would be placed in that functional unit's instruction slot. On the other hand, the VelociTI decouples fetch packets from execute packets through a novel instruction encoding system. The least-significant bit of every C62x instruction is a called the parallel or p-bit. The p-bit of a particular instruction is set if the instruction starts execution in parallel with the next instruction. In addition, the reduced code size of the VelociTI instruction also results in fewer program fetch accesses.

4.2.3.4 Conditional Instructions

Every C62x instruction can be conditioned on either the zero (false) or the nonzero value (true) of one of the five general-purpose registers (A1, A2, B0, B1, B2). All instructions will enter the first phase of execution regardless of the evaluation of their condition. However, if the condition is not met by the end of the first phase, the instruction will not have its results written back to the register file. In addition, a conditional load or store instruction whose condition is not met is canceled before entering the data memory portion of the execution pipeline. In this way, it prevents any undesired accesses to memory, including memory-mapped peripherals where simple access has undesired side-effects. Conditional instructions can be used to avoid branch latency. In control code, conditional instructions allow increased parallelism as multiple paths can be executed simultaneously. For instance, both *if* paths as well as the *else* path are executed in a single fetch packet.

4.2.4 Memory Architecture

The memory architecture of the C6201 consists of four components: the EMIF (External Memory Interface); the DMA controller; 64-Kbytes of on-chip program memory configurable as mapped memory or as direct mapped cache; and 64-Kbytes of interleaved data memory. The instruction cache may alternatively be configured as an on-chip program memory containing a valid address space. The C62x Evaluation Model (EVM) board by Texas Instrument has been used for our preliminary software implementation platform. The board has a 200 MHz C62x fixed-point DSP processor with access to two 4- Mbytes of SDRAM (synchronous DRAM running at ½ the CPU clock rate), and 256-Kbyte of SBSRAM (synchronous burst SRAM with 800 Mbytes/second throughput).

The internal programming memory contains 16K 32-bit instructions or 2K 256-bit fetch packets and the block size of the cache is a fetch packet or eight instructions. For the 64-Kbyte C6201 internal data memory, it is organized into eight 8-Kbyte 16-bit-wide banks of memory, where these banks are grouped into two blocks consisting of four banks each. The address of each bank in a block is interleaved, and the banks are arbitrated on a 16-bit cycle-by-cycle basis. The communication between the Host PC and the DSP can either use DMA transfer or Host Port Interface (HPI) which is a 16-bit wide bi-directional port that interfaces with little logic to a variety of industry-standard embedded RISC processors, and microprocessors. The HPI interface can operate at up to 50 MHz for 100 Mbytes/second of data throughput. More details of the EVM board can be found in [35].

4.3 Memory Configuration and Host Interface of the MPEG-2 Video Encoder

There are two version of the software MPEG-2 video encoder implemented on the C62x EVM board. The first version of the software encoder is implemented at CCIR-601 resolution with I-frames only, and the second version of the encoder is implemented at

SIF resolution with I- and P-frames. Both versions of the software encoder share the same structure on the communication between the Host PC and the C62x EVM board. The Host PC side, either transmits the uncompressed picture frames to the C62x EVM board or it receives the compressed video bit-streams from the C62x to the Host PC. The compressed video bit-stream is stored onto the local hard disk to be played later on. The video encoding of the MPEG-2 is done on the C62x EVM board with the input of YUV frame data either captured from the camera or loaded from the local hard disk. The output of compressed video bit stream is also produced for display. Figure 26 illustrates the flow of our implementation of the software MPEG-2 video encoder over the C62x platform.

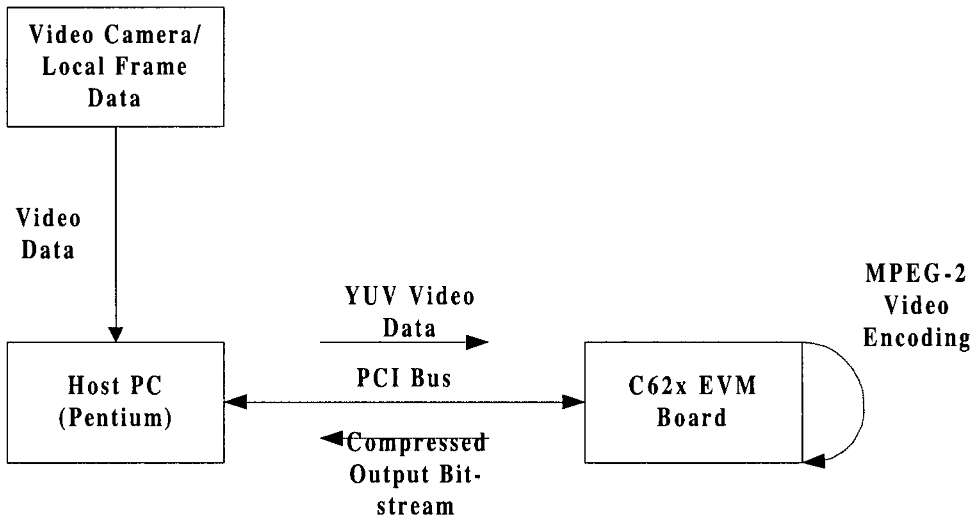


Figure 26: Software MPEG-2 video encoder over the C62x platform.

The main video encoding part resides on the C62x EVM board for both the CCIR-601 and SIF versions are different because the I- and P-frames require more complicated algorithms, and the memory requirements are more restricted due to the nature of frame store for motion estimation etc. The following subsections will describe the structure of both version of the video encoder on both Host PC and C62x EVM board.

4.3.1 Video Encoder on the Host PC

The software MPEG-2 video encoder resides on the Host PC mainly for communication purpose. Therefore, both versions of the MPEG-2 encoder (CCIR-601 or SIF resolution) are handled the same way on the Host PC. Figure 27 on the following page shows the flow diagram of the MPEG-2 video encoder program on the Host PC side.

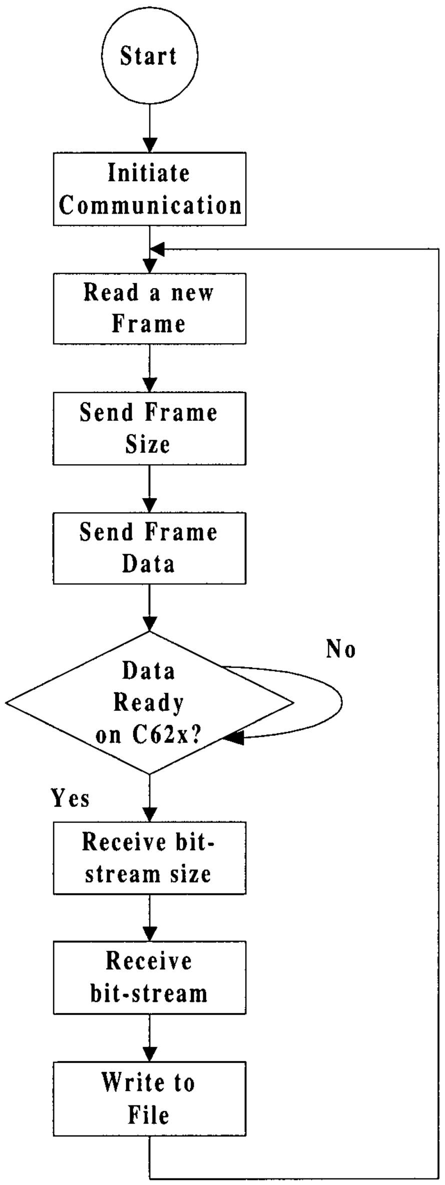


Figure 27: Flow diagram of MPEG-2 video encoder on the Host PC.

The program starts with initializing all the necessary parameters to setup the communication between the Host PC and the C62x EVM board. The configurations are done according to the following sequence:

- Open handle to the EVM board

- Cause a hardware reset on the target board
- Setup the board configuration such as the DSP_CLOCK_NORMAL and LITTLE_ENDIAN_MODE
- Set the boot mode and cause a DSP reset
- Establish a connection to the HPI of a target board
- Initialize the EMIF (external memory interface) registers
- Set the Aux DMA priority higher than DMA
- Read a COFF file and write the data to DSP memory
- Release the DSP from the halted state

After the initialization of the C62x EVM board and loading the program into the memory, the video encoder on the C62x EVM board is ready for execution. However, before running the program, the input video streams are fed from a video camera or from video data files stored on the local hard disk of the PC. Once the video data is ready, it will write the frame size to the C62x EVM and follow with the frame data along with a flag to notify the complete download of the video frame data onto the C62x memory. All the transfer are done through the HPI. Then the Host PC will be put into a sleep state while waiting for the encoder on the C62x EVM board to process the video encoding on the video data. The Host PC will be keep pulling for a flag from the C62x EVM to notify the completion of encoding while in the sleep state. Once the flag is set, the HPI will transfer back the compressed video bit-stream. However, since the HPI always transfers in 4 bytes size each time, the read length need to be a multiple of 4 to ensure the ending bit-streams data are properly transferred without truncation.

We can see that putting the Host PC in sleep state while waiting for the C62x EVM to do the encoding or halting the C62x to wait for the HPI transfers video data back and forth seem to be wasting a lot of resources. Therefore, we have implemented a dual buffer system to keep both the Host PC and the C62x EVM busy all the time in order to maximize the utilization of both systems. The dual buffer system consists of the following structure:

- Two buffers reside on the single 4-Mbytes SDRAM of the C6x EVM board, and
- each buffer consists the first 4 bytes as the ready flag, and
- 1020 bytes for the input video data, and
- the first 4 bytes after the input video data as the video output bit-stream size, and
- 1020 bytes for the video output bit-stream data.

The dual buffer system works while downloading the video data into the first buffer, the C62x encodes the video data on the second buffer which has already downloaded previously. Once the C62x finish encoding the second buffer's video data, it sends a flag to tell the Host PC that the compressed video bit-stream is ready for upload, and the C62x will switch to the first buffer with the recently downloaded video data and start encoding. Then after the Host PC received all the output video bit-streams from the second buffer, the process continues and vice versa. This process is continued until the last frame has been encoded and transfers back to the Host PC. By using the dual buffer system for our system, the encoder on the C62x EVM is fully utilized except the initial setup time for the very first frame. Figure 28 shows the memory structure for the dual buffer system implemented for the C62x EVM board.

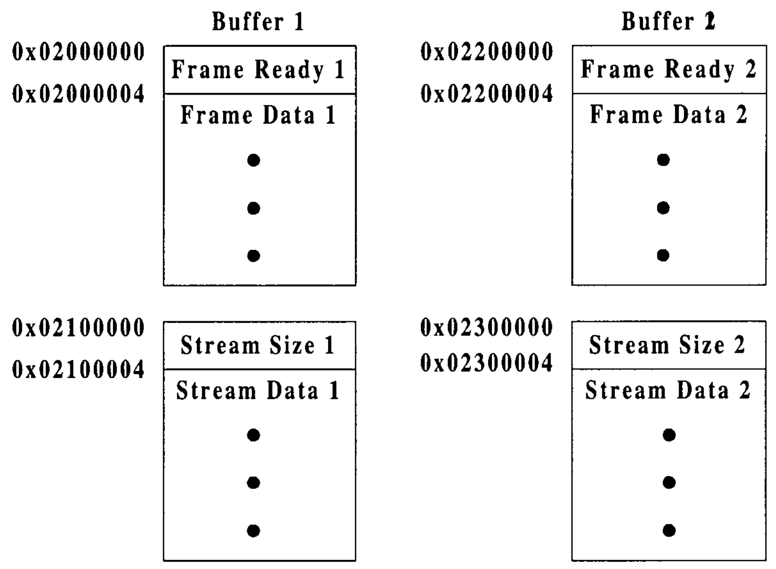


Figure 28: Dual buffer memory structure for the video encoder on the C62x EVM.

After the video encoder finishes encoding all the frames, the output bit-streams are stored onto the local hard disk. Then we have to close the connection with the C62x EVM board. This process includes closing the HPI session, causing a hardware reset on the target board, setting the boot mode that cause a DSP reset, and finally closing a previously opened driver connection to the EVM board.

4.3.2 MPEG-2 video encoder on C62x EVM board

For the software MPEG-2 video encoder implemented on the C62x EVM board, there are two versions of it: one is the CCIR-601 resolution with I-frames only, and the other is SIF resolution with both I- and P-frames. The following subsections will describe the implementation of both encoders on the C62x EVM board.

4.3.2.1 CCIR-601 resolution on C62x

Figure 29 depicts the implementation of the MPEG-2 video encoder on the C62x EVM board with the CCIR-601 resolution on I-frames only. The program starts with initializing the pointer to the input video data (YUV) on the SDRAM. Since the internal data memory of the C62x is limited to 64-Kbytes and it is where the data is being processed at the fastest speed, we have to partition the video frame into slices (YUV data for each slice are @720x16 for Y, 360x8 for U and V) in order to fit it into the internal data memory. Each iteration of the program processes three slices of Y, U, and V with a total of 51.84-Kbytes which is just enough to fit into the internal data memory, and the rest is reserved for other uses. The three slices of YUV data are moved from the SDRAM to the internal data memory. For every iteration, the program performs the video encoding individually since the intra-coding of each slice is independent from other slices. All the encoded bit-streams are written back to a continuous block of SDRAM in order to be transfer back to the Host PC when a complete frame is being encoded. The data transfers between the Host PC and the C62x external memory (buffer) are transparent to the video encoder on the C62x. To start the execution of the program to encode each frame is triggered by receiving a flag from the Host PC for complete data transfer, and its ended by writing a flag to the specific memory location in order to notify the Host PC to be ready for transfer the compressed video bit-streams back to the PC.

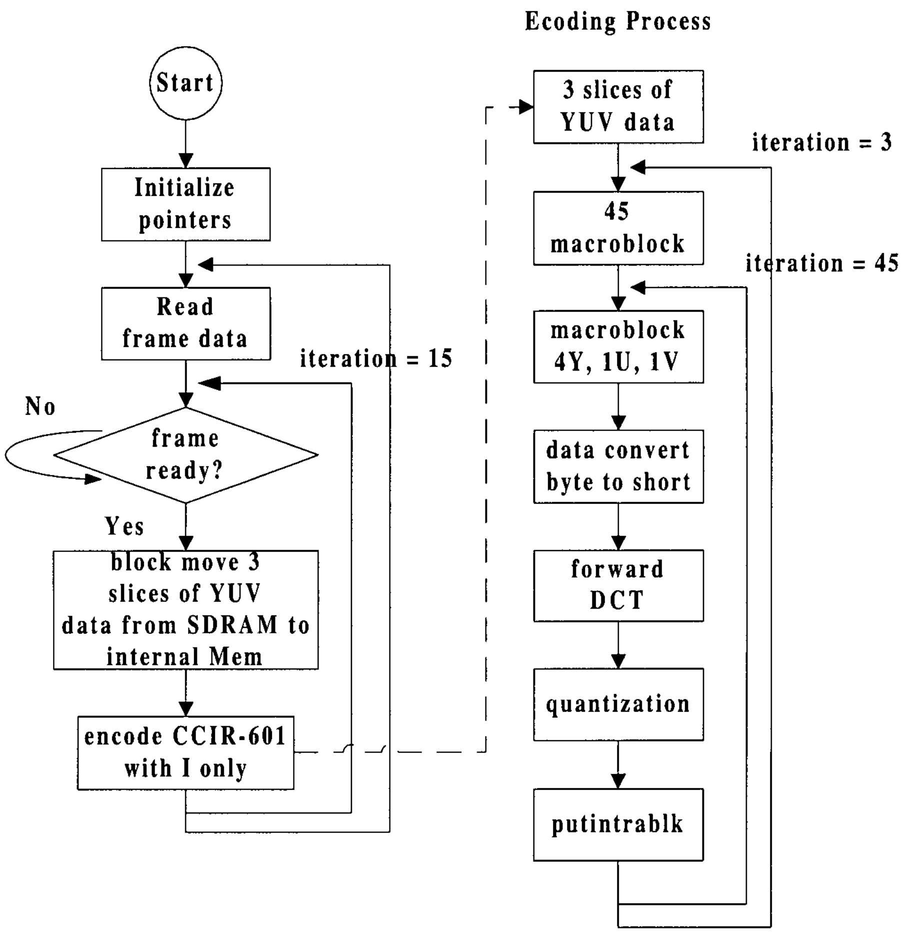


Figure 29: Flow diagram for the CCIR-601 resolution video encoder on C62x.

As shown in Figure 29, after the data is passed over to the encoder, the encoder processes the data on a macroblock base for each slice. For every macroblock, six identical functions are processed on each 8x8 blocks. These functions include converting a byte data into a short data type for 16-bit calculation, forward DCT, quantization, the run-length entropy coded using variable length code (VLC) tables, and putbit operation to write the data back to the external memory.

The memory map of the CCIR-601 with I-frames only is as follows: the encoder program is mapped into the internal programming memory; data, stack, and heap are mapped onto the internal data memory; the constant and initialization data are mapped onto the SBSRAM (synchronous burst SRAM). The input frame (YUV) data and the compressed output bit-streams are placed onto the SDRAM.

The performance of the CCIR-601 resolution video encoder on the C62x without any assembly optimization runs approximately **0.48 frame/second**, which is far from the desired real-time video encoding speed at 30 frames/second. Therefore, optimization of individual functions is required to achieve an acceptable result, which will be further discussed in the next chapter.

4.3.2.2 SIF resolution on C62x

The initialization and program execution of the MPEG-2 video encoder, on the C62x EVM board with the SIF resolution on both I- and P-frames, is very similar to the previous version of the encoder except one more pointer has been initialized for the reference frame which is mapped to the SBSRAM. Also, the encoding process for the IP encoder is much more complicated. The memory reserved at the SBSRAM is to store the reconstructed current frame in order to be use on the P-frame encoding later. Memory arrangements on the internal data memory are much different then the previous version of the encoder due to the intensive calculation of motion estimation and the memory required to store the reference data and low resolution data for the IP encoder. The encoder encodes only one slice of the YUV data (320x16 for Y, 160x8 for U and V) at a time with a total of 7.68-Kbyte used of the internal data memory. This slice of YUV data is copied from the SDRAM to the internal data memory. Three slices of the reference frame are reserved on the internal data memory in order to do the motion estimation calculation and they are copied from the SBSRAM, with a total of 23.04-Kbytes used. Memory transfers between SRAM and SBSRAM to internal data memory are transparent to the IP encoder. Also, 7.68-Kbyes of internal data memory is reserved for the low resolution YUV data that are generated for low resolution motion vector search. Figure 30 shows the flow diagram of the MPEG-2 IP video encoder with SIF resolution.

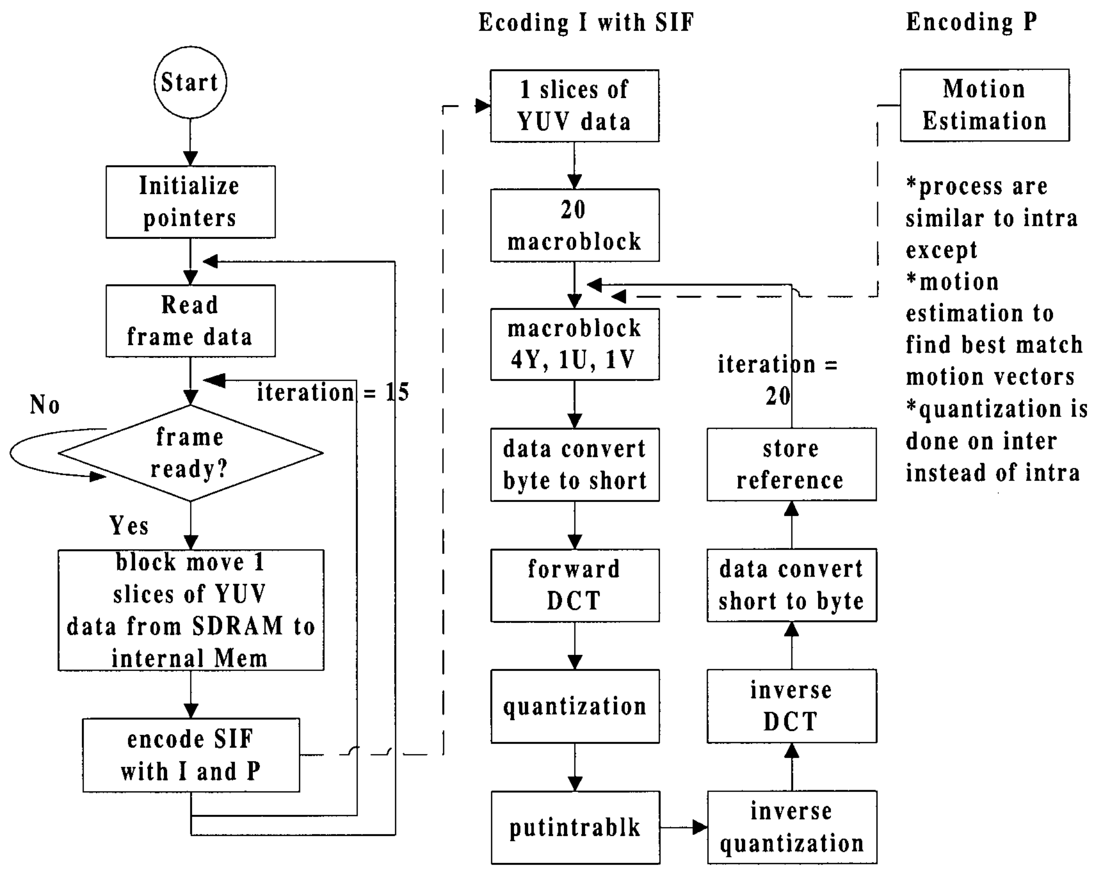


Figure 30: Flow diagram of the MPEG-2 IP video encoder on C62x.

As shown in the diagram above, the intra frame encoding is similar to the previous version of the encoder except that the current version of the IP encoder includes the inverse DCT and inverse quantization to reconstruct the transformed image for later use as the reference frame. However, the inter frame encoding, which does not exist in the previous version of encoder, is the most computational intensive part of the IP encoder. The inter frame encoding accounts for more than 85% of the entire IP encoder, therefore, we will provide a more detailed discussion here. As the C62x has limited internal data memory, we can only process one slice of YUV data of the current frame at a time because we need to hold another three slices of YUV data from reference frame in order to do the motion estimation calculation. The reason we choose three slices of reference frame is that we need to have a search window size of ±16 pels horizontal and ±16 pels vertical. However, the reference slices in the internal data memory vary with the three different conditions as shown in Figure 31.

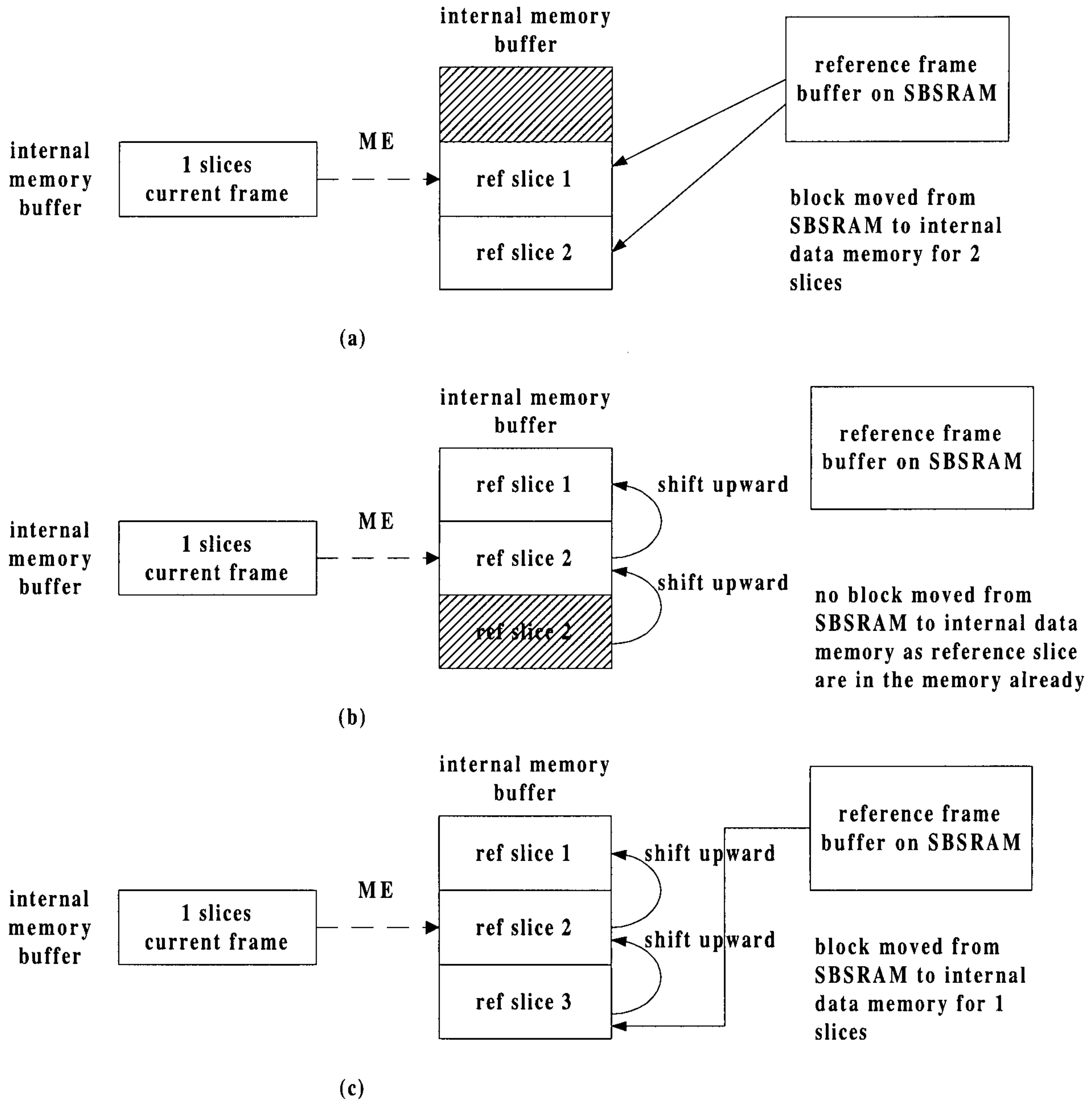


Figure 31: Three different condition of the reference slices in the internal data memory: a) first reference slice, b) last reference slice, c) all other slices.

In Figure 31a, the search window for the first slice of a frame constitutes only the positive 16 pels vertically, since the negative pels search area of the reference frame are out of the image boundary. Therefore, we only need to compare two slices in the reference frame: one is the same location as the compared slice, and the other one would be the next slice of the reference frame. Thus, we move in only two slices from the reference frame memory. For Figure 31b, instead of having a positive search window size, the last slice of the frame only needs the negative 16 pels vertically. The positive pels search area is out of image boundary also. But instead of copying from the reference frame memory, the two reference slices already reside in the internal data memory, therefore, we only need to shift up the two slices to do the motion vector search. Figure 31c shows the arrangement of all other video slices for the reference window. Since the search window is ±16 pels vertical, we need two other reference slices in the reference window, which include one slice above and one slice below the current reference slice. We only need to copy in one new slice from the reference frame memory, since the other two reference slices are already inside the reference window. Therefore, we only shift up the two slices that are already in the reference window and then copy the new slice to fill up the bottom reference window. Similar processes are done on the low resolution motion vector search too.

After all the necessary YUV data being placed in the internal data memory, the inter frame encoding will start first by doing the motion search to find the best motion vector of the current macroblock. Algorithms used to find the best motion vector using the SAD are discussed in the previous chapter such as predicted motion vector, hierarchical block matching strategy, diamond search, etc. After finding the best motion vector, the macroblock would process the forward DCT and inter quantization to obtain a transformed block. The inverse quantization and inverse IDCT is also performed to obtain the reconstructed image to be use as a reference frame for the next inter frame encoding. The transformed macroblock is then run-length variable length encoded and write out the compressed video bit-streams back to the buffer (SDRAM).

The memory map of the IP encoder with SIF resolution is as following: the encoder program is mapped into the internal programming memory; data, stack, heap, constant, and initialized data are mapped onto the internal data memory. The reference frame data are stored onto the SBSRAM. The input frame (YUV) data and the compressed output bit-streams are placed onto the SDRAM.

The performance of the IP video encoder on the C62x without any assembly optimization runs approximately **0.52 frame/second**, which is far from reaching the goal of real-time video encoding at 30 frames/second for I- and P-frames. Therefore, optimization of individual function is required into to achieve an acceptable result, which will be further discussed in the next chapter.

4.4 Implementation Bottlenecks

As shown in the previous section, the preliminary result of both version of the encoder is not running as efficient as we would expect, or even worst than expected. Therefore, in this section we would discuss what are the potential problems that could cause the inefficiency for execution of the program on the C62x platform.

4.4.1 Memory Issues

The internal on-chip bus, external memory interface (EMIF), and peripherals often represent the critical bottlenecks in a real-time system. Access to vital external programs and data must pass through these critical components on their way to the VLIW core for processing. In fact, external memory access delays form the greatest performance bottleneck to our MPEG-2 video encoding on the C62x platform. There are two memory issues that need to be addressed; one is the internal program memory, and the other is the internal data memory. As one major problem was with the internal data memory is that the MPEG-2 video encoder program is much larger than the 64-kbyte on-chip memory of the C62x, which means that the main program must be run on an external memory. This can cause significant slowdown of the program. Therefore, one of the alternatives is that we have treated the internal program memory as a memory cache and attempted to optimize the cache hit-ratio to enhance the performance; the other alternatives would be limiting the functionality of the program in order to fit into the internal program memory. Since even with a close to optimal cache hit-ratio, running the entire program under the internal program memory still outperform the other option.

As the internal data memory can transfer data 20 to 30 times faster than the external memory, it is desirable to fetch all the data for computation from the internal data memory. However, given the limited internal data memory size of the C62x has, fitting all the data into the internal data memory would seem to be impossible and impractical. As shown in our implementation of the encoder in the previous section, the encoder swapping data in and out from the internal memory to the external SBSRAM or SDRAM memory. The swapping process could potentially reduces the performance of the program be executed. One alternatives would be use an optimized block move algorithm to do the swapping; the other alternatives would be to use the DMA transfer without the intervention of the CPU while the data are being swapped.

4.4.2 Compiler Optimization Issues

Even though TI provides an optimized compiler with different levels of code optimization, the typical code performance is improved only by 30% - 40% on average. However, for time critical real-time systems, the level of performance improvement is still not sufficient. Much more aggressive optimization can be obtained by hand-optimized assembly code. For many of the video coding components, assembly-level optimization performs three to twenty times faster than the code generated by the optimized compiler. However, the development time for using assembly-level optimization is several order of magnitudes than just switching on the build-in compiler option. Further discussion of the hand-optimization methodology will be given in the following chapter.

4.4.3 Instruction Latency

Instruction latency for most of the C62x arithmetic operations are one cycle as refer to single cycle instruction (ISC), and others such as multiplication instruction requires two cycles latency. However, load/store and branch instructions have a latency of five and six cycle, respectively [36]. And for many of the video coding components, load and store represents most of the optimization bottlenecks when optimizing the video encoder. Further discussion of the load and store instruction issues will be shown in the following chapter.

4.4.4 Memory Bank Conflict

The memory architecture of C62x consists of four memory banks, and each of which is half-word size (or 16-bytes) wide. Simultaneous memory access to the same bank can cause a memory bank conflict which stalls one extra cycle for every load or store instructions. A possible alternative to avoid memory bank conflict would be arranging the entire data element to 32-bit memory align.

Chapter 5

Optimized VLIW Algorithm for MPEG-2 Video Encoding

5.1 Introduction

In this chapter, we present platform-specific (VLIW) optimizations for the computationally intensive components of MPEG-2 video encoding mentioned in the previous chapters. These video coding components include the SAD computation, Quantization, Quantized Discrete Cosine Transform (QDCT), variable length encoding (VLC), and MSE computation functions which can achieve a much better performance with assembly-level hand optimization. The purpose of these optimizations is to take advantage of the VLIW architecture of the C62x and to overcome some inherent memory-access and I/O bottlenecks so as to enable real-time video encoding on the C62x processor.

Video coding systems, like many other multimedia systems, has very computationally intensive core functions that can be implemented efficiently using ILP. Thus, a significant performance increase can be expected if the VLIW architecture is properly exploited. Therefore, a major goal is to have an efficient MPEG-2 video encoder running on the C62x platform.

5.2 Optimizing SAD Computations for Motion Estimation

During inter-frame coding in MPEG-2, motion vectors are found by performing a number of SAD operations on 16x16 blocks. For every SAD calculation, there are two unsigned load byte instructions, one subtraction, one absolute operation, and one addition which consumes a total of five out of the eight available functional units (the two multiplier functional units are not use in SAD calculations). The performance of the inner loop of the SAD computations is constrained mainly by the number of load/store functional units in the C62x processor. Because only two load operations can be performed per cycle, the SAD computation can compute the absolute difference of these two values in one step, producing one result per cycle at most. Two approaches can be used to optimize the SAD computation.

5.2.1 First Approach of SAD Optimization

Considering the number of available functional units that C62x has, the inner loop could perform one result per cycle at most if every functional unit only processes one video data at a time. The first approach unrolls the 16x16 two-dimension loop into a one-dimensional loop with 16 iterations, each consisting of 16 SAD computations. This loop unrolling reduces the number of loop-end branch instructions from 64 to 16. Because each branch instruction has a latency of 6 cycles, reducing the number of branch instruction has significant impact on speedup. This structure of the SAD program is as follows. Before entering the inner loop, eight execution packets are preloaded to increase the pipeline efficiency and eliminate load-instruction latency (which is 5 cycles). The inner loop contains 16 execution packets representing a row of 16 SAD operations, with each execution packet consisting of at least two load instruction, one subtraction, one absolute instruction, and one addition which are executed concurrently in different functional units. The 11th execution packet also includes a "delayed" branch instruction that actually takes effect after the 16th execution packet is computed. Figure 32 shows the scheduling of instructions on the first iteration of 16 SAD computation using assembly-level optimization.

availaable register file

(Maximum of 8 instruction per execution packet)

Increase Cycle Count

loop initial	S AB															
cycle 1	AD	L1	AB		S										L1	
2		L2	AD	L2	AB		S								X	
3		X		L1	AD	L1	AB		S						X	
4		X		X		L2	AD	L2	AB		S				X	
5		X		X		X		L1	AD	L1	AB		S		X	
6		X		X		X		X		L2	AD	L2	AB		S	
7		S		X		X		X		X		L1	AD	L1	AB	
8		AB		S		X		X		X		X		L2	AD	L2
9	L1	AD		AB		S		X		X		X		X		L1
10	L2		L2	AD		AB		S		X		X		X		X
11	X		L1		L1	AD		AB		S		X		X		X
12	X				L2		L2	AD		AB		S		X		X
13	X					L1		L1	AD		AB		S		X	
14	X							L2		L2	AD		AB			S
15	S									L1		L1	AD			AB
16	AB											L2		L2		AD
loop end	AD NOP 5															

L1: load from current frame buffer
L2: load from reference frame buffer
S: subtract the difference of L1 and L2
AB: absolute the result
AD: sum of the absolute difference
X: load instruction latency

Figure 32: Instruction scheduling of the first 16 SAD computation.

In Figure 32, the X represents the load instruction latency. As shown in Figure 32, each row presents one instruction packet with up to a maximum of six instructions being executed simultaneously since the other two multiplier functional units are not being used in the SAD computation. The reason for loading the two comparing video data in a two separate instruction is to avoid resource (simultaneous execution on the same functional unit) and memory conflict, and also the inner loop can be nicely packed with 16 instruction packets for 16 SAD computation. To complete the calculation of the absolute difference which requires eight cycles latency and one more cycle to do the sum of all the absolute differences. Extra instructions are inserted for the completion of the SAD operation such as incrementing the data memory pointer, reducing the loop counter, swapping register contents to avoid resource conflict.

In addition, the next step of optimizing the SAD computations involves resolving any possible memory-bank conflicts which can degrade performance. Indeed, the parallel loading of two contiguous unsigned data bytes from memory can result in memory bank conflicts 50% of the time. As every second load instruction in the same instruction packet would stall one extra cycle to load the data. Figure 33 shows how our program avoids this conflict by loading every other unsigned data byte during a parallel-load execution packet. By applying this technique, the memory accesses for loading data are guaranteed to access through different memory bank, thus, avoiding the memory bank conflict. Since computing the SAD of a block is independent of which order the pixels are compared as long as comparisons are done on the same pixel location between the current and reference frame, the correctness of computation is preserved.

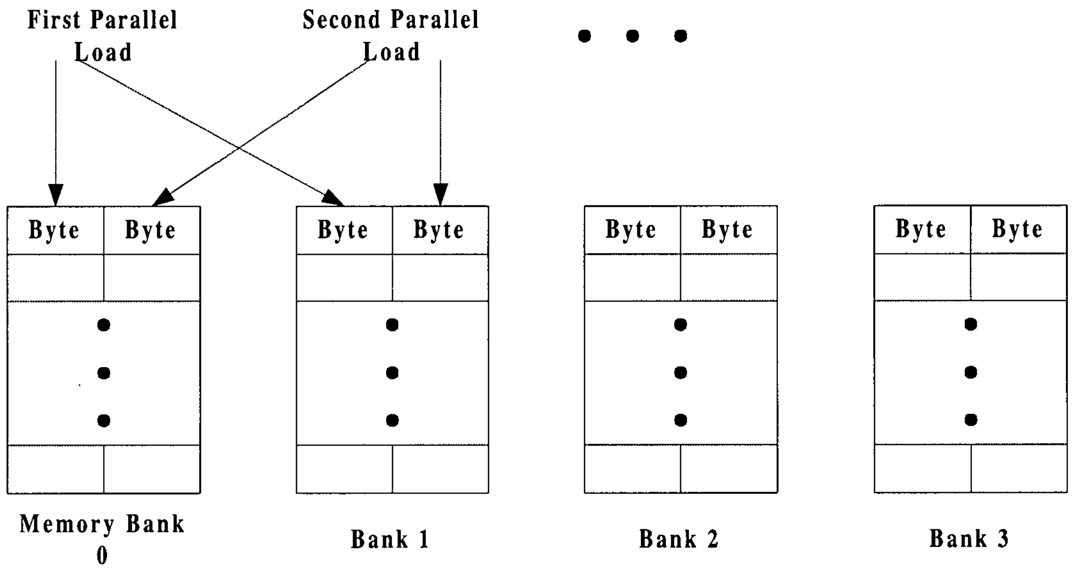


Figure 33: Avoiding memory bank conflicts by loading every second data from different memory banks.

The optimization of the SAD computation can include the partial SAD computation technique with little extra overhead. Since the sum of every row SAD is compared to the previous block's minimum SAD, if the current row's SAD is already larger than the previous block's SAD value, then the computation of current block of SAD will be terminated early. Overall, our SAD function implementation runs approximately 4 times faster than the version obtained from the optimizing C compiler of the C62x. Specifically, computing the full SAD function without early termination for a 16x16 block requires only **270 cycles**, as compared to **1220 cycles** if the C compiler is used. A more aggressive approach with the SAD computation is to downsample the 16x16 block to perform only one row of SAD calculation for every two rows of data. This will cut down the number of cycles required by the computation by half. Given such an aggressive approach, the downsampling of the SAD computation does not sacrifice the video quality while increasing the performance.

5.2.2 Second Approach of SAD optimization

Further optimization is possible by applying the subword parallelism feature of the C62x architecture. This can result in producing four SAD results per three execution cycles (i.e. about 1.333 results per cycle) compared to the one result per cycle attained by the first approach. Instead of loading the two unsigned data bytes in each cycle, the two data words, containing four unsigned bytes of data, are loaded. Loading four unsigned data bytes at a time eliminates the load instruction constraint as suggested in the first approach. The operation of the improved SAD computation technique is depicted in Figure 34. After the two data words have been loaded, the half-word subtraction function (SUB2) is used to compute the difference between A2 and B2 to obtain r2, and the difference between A4 and B4 to obtain r4. Then both data words are right-shifted by eight bits and a similar SUB2 function is applied to obtain r1 and r3. Now, both r1 and r2 use the EXTU function to extract the 8-bit result of the difference data; while r3 and r4 will be ANDed with 0x000Fh to clear the upper 24 unrelated bits. Four absolute computations are performed concurrently on the results (r1, r2, r3, r4), then the results are added up to obtain the SAD for four unsigned data bytes. This implementation requires two load instructions, two half-word subtract instructions, two unsigned right shift instructions, two unsigned extract instructions, two AND instructions, four absolute instructions, and four add instructions. Therefore, the total of number operations require 18 instructions which can be packed into thee execution packets, as opposed to four execution packet in the first approach. Hence, the resulting speedup. In addition, loading 32-bit word data also avoids memory bank conflicts.

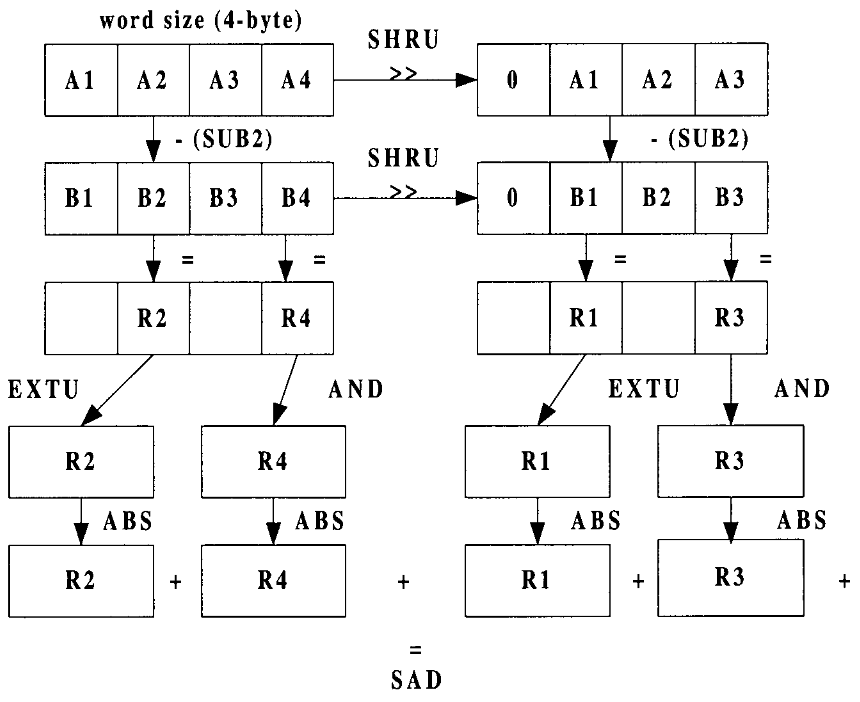


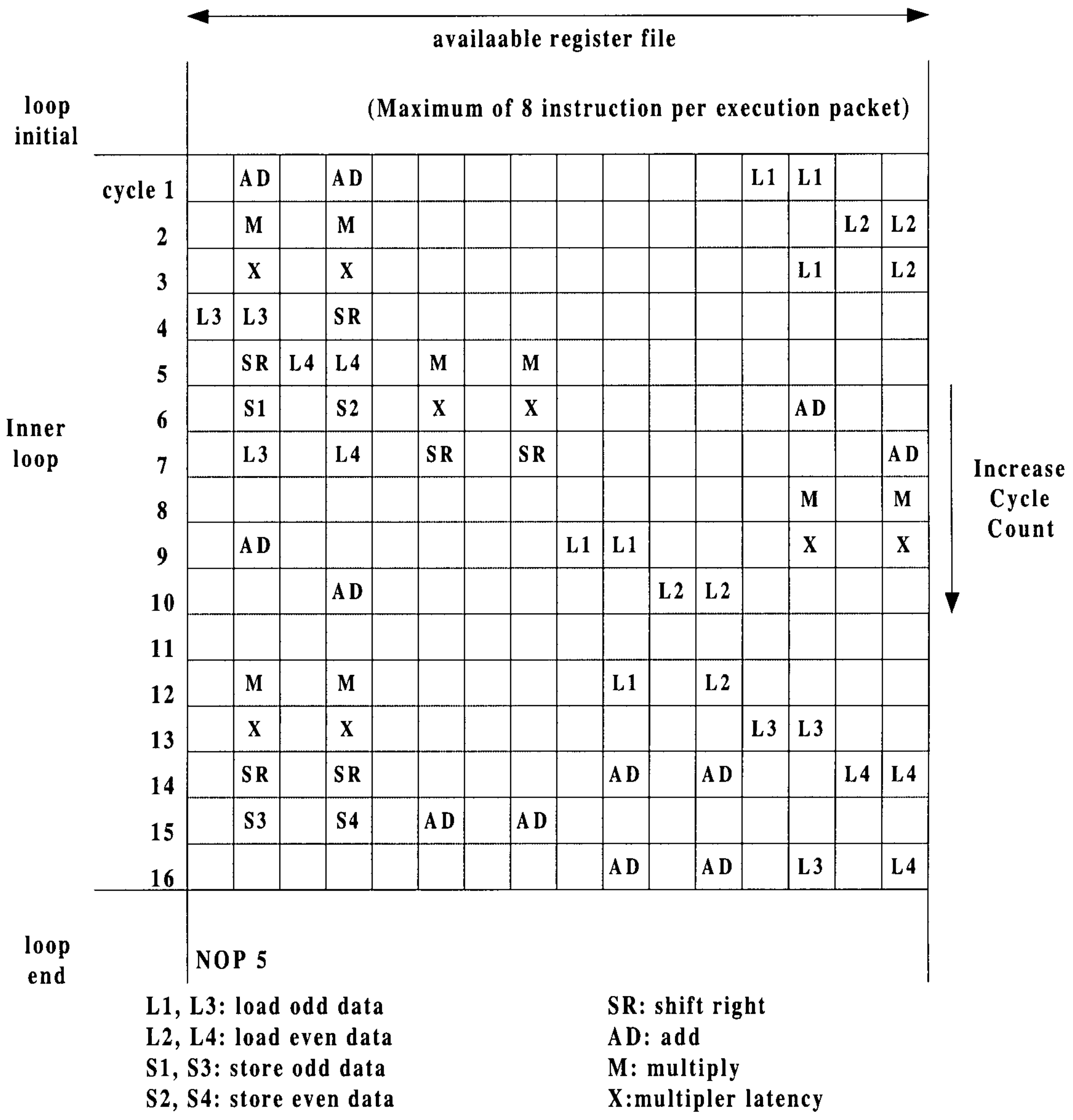
Figure 34: Flowgraph of the SAD computation in VLIW implementation.

5.3 Quantization

After the transform coding (DCT) is performed, quantization of the DCT coefficient will be processed. The quantizer we use is set to a fixed scale of 10 bits with all of the quantizer scale coefficient calculated at the beginning. We use this quantization function for the intra-coding stage only. The main quantization step performed is of the following form

$$Q[i] = \{(x[i] + c1[i]) \times c2[i]\} >> 16 \tag{7}$$

where x is a DCT coefficient, $c1$ is a quantizer scale, and $c2$ is a computed scale. The main computation here consists of quantizing 64 DCT coefficients for an 8x8 block. Although the computation required for the quantization of DCT coefficient is straight forward, the C6x compiler optimization could not fully exploit the available instruction-level parallelism. In our optimization, the main constraint is due to each execution packet allowing only two load/store instructions. The above quantization step requires three loads and one store. Under this constraint we are able to achieve two results per cycle in the inner loop. Loop initialization, loop unrolling, and delayed branching have been all used in the optimization. The main inner loop consists of 16 execution packets each consisting of two load/store operations. Loop unrolling gives the same effect of reducing the number of branch instructions and this has a significant impact on speedup. However, most of the execution packets are not fully utilized since only one addition, one multiplication, and one shift are performed within two cycles on average. The structure of the quantization step computation is as follows. Before entering the inner loop, six execution packets are preloaded to increase the pipeline efficiency and also eliminates load-instruction latency. The optimization structure of the program is slightly different than the SAD optimization, since the SAD computation has almost identical instructions inside every execution packet in the inner loop. However, different instructions are packed in most execution packets with a similar pattern in the quantization step computation. Figure 35 illustrates the scheduling of instructions in the quantization step computation program.


availaable register file
(Maximum of 8 instruction per execution packet)
loop initial
Inner loop
loop end
Increase Cycle Count
NOP 5
L1, L3: load odd data
L2, L4: load even data
S1, S3: store odd data
S2, S4: store even data
SR: shift right
AD: add
M: multiply
X:multipler latency

cycle 1		AD		AD									L1	L1		
2		M		M											L2	L2
3		X		X										L1		L2
4	L3	L3		SR												
5		SR	L4	L4		M		M								
6		S1		S2		X		X						AD		
7		L3		L4		SR		SR								AD
8														M		M
9		AD							L1	L1				X		X
10				AD							L2	L2				
11																
12		M		M						L1		L2				
13		X		X									L3	L3		
14		SR		SR						AD		AD			L4	L4
15		S3		S4		AD		AD								
16										AD		AD		L3		L4

Figure 35: Instruction scheduling for optimizing the quantization step computation.

As shown in the instruction scheduling of the inner loop, three execution packets are used to load two sets of input parameters for the quantization step computation. Result of the two computations are computed at the sixth to eighth cycles and the 10th cycle. Then store takes place in the 11th cycle. The two final results would not get stored actually until the fifth cycle after the initiation of the store instruction. Overall, our optimized quantization function runs on the C62x approximately 20 times faster than the optimized C-compiler version. Specifically, quantizing an 8x8 DCT block requires **128 cycles** by our method, as opposed to **2505 cycles** generated by the C-compiler, which is merely 20 times faster.

5.4 Quantized Discrete Cosine Transform (QDCT)

As mentioned before, the QDCT algorithm combines both DCT and quantization into a single computation and reduces the overall computational load substantially. Unlike SAD and quantization calculations, QDCT's high computational cost is dominated by its complexity. The two-dimensional QDCT computation must be unrolled to an 8x1 vector such that the column QDCT is preformed first then row QDCT is performed second. In QDCT, load and store instructions are no longer the dominant constrains for optimization. However, the order of load and store operations still has an important impact on the overall performance of the optimization.

Two methods have been implemented to determine which order of loads and stores provides a better optimization level of the QDCT. The first, and simplest, method is to do a parallel load of 8 data at the beginning of each iteration, then transformed results are store at end each iteration. The advantage of this method is that the code linearity is preserved and register usage becomes more flexible. The drawback of this method is that resources are poorly utilized when intermediate result, such as those from multiplications (2-cycle latency) and load operation (5-cycle latency), are not available.

The result of computing a column of QDCT requires 24-instruction cycle in the inner loop, and eight iterations are required for completing the column QDCT calculation for a total of 192 instruction cycles. In the second method, a sequential order is enforced on the load/store instructions such that one load and one store operation are performed per instruction execution. However, the paired load and store instructions do not access the same row or column of QDCT calculation in the same iteration. In other words, during the first iteration of computing the column QDCT, the load instruction is loading in the first set of data, and the store operation of the first iteration are disabled. Then for the second iteration of column QDCT calculation the store operations are storing the result of the first row of QDCT while the load operations are loading in the second data set. This method results in much better functional unit utilization than the first methods. The main limitation, however, is increased register usage caused by the use of preloaded quantized coefficients and the storage of intermediate results. The resource constrain on the number of available shift units become the one of the bottleneck as most results are shifted at the end to maintain the necessary precision of the integer QDCT computation. Using the second method, a QDCT column computation requires 14-instruction cycles in the inner loop, and nine iterations for completing the column QDCT calculation for a total of 126 instruction cycles. Therefore, the second method has a 34% performance gain over the first method. The two methods of load/store scheduling are depicted in Figures 36 and 37.

```
LDH       Data1     ;parallel load of
||LDH     Data2     ;two data

LDH       Data3
||LDH     Data4
  .
  .
  .
STH       Data1     ;parallel store of
||STH     Data2     ;two data

STH       Data3
||STH     Data4
```

Figure 36: Parallel execution of loads at the beginning and stores at the end.

```
LDH       Data1     ;sequential load/
||STH     Data1     ;store pairs

LDH       Data2
||STH     Data2
  .
  .
  .
LDH       Data7
||STH     Data7

LDH       Data8
||STH     Data8
```

Figure 37: Sequential execution of load/store pairs.

For the column QDCT computation, the average utilization of the C62x functional units on the 14 execution packets is 78%. However, for the row QDCT computation, the average utilization on 13 execution packets is 81%. Among all the optimization of the video components, the QDCT optimization achieves the highest utilization of the functional units in the C62x platform. Finally, the performance gain of using optimized assembly is almost 20 times faster as than the C6x C-compiler implementation of QDCT.

5.5 Variable Length Coding (VLC)

For the SAD computation, quantization, and QDCT, the performance of the optimization is independent of the context of the data, and all of these algorithms have tight inner-loops that are good candidates for optimization on an ILP architecture. The picture is quite different for the VLC algorithm whose structure is more inherently sequential with strong data dependencies. The performance of the VLC algorithm is highly dependent on the quantized DCT coefficients, the more zeros the quantized 8x8 blocks have, the more efficient the algorithm runs. Inserting a branch instruction after every quantized DCT coefficient within the 8x8 blocks poses the major hurdle to the VLC algorithm performance. In addition, the zig-zag scan of the DCT coefficients in an 8x8 block causes further inefficiencies in performance of the program code. We have, therefore, employed a method for hard coding the zig-zag scan pattern and generating a table of zero run-length codes for use with Huffman coding later on. The performance of our optimized code is three to four times faster than the code generated by the C-compiler. This has been one of the hardest algorithms to optimize for the C62x because of the high inter-dependencies in the program code execution. For example, in the intra-coding process, VLC accounts for almost 50% of the computational load after the other components have been optimized.

5.6 Mean Square Error (MSE)

Mean square error (MSE) computation is similar to the SAD computation, except that MSE calculate square of the difference between two data elements. For the MSE calculation, multiplication is used instead of the absolute function used in the SAD computation. The hand-optimized assembly code for MSE is very similar to that of the SAD except for the use of an additional multiplication (with a 2-cycle latency) per execution packet, and this results in a total of 271 instruction cycles for computing the MSE for a 16x16 luminance block. For the two 8x8 chrominance blocks, 95 instruction cycles are required for each block. The performance of the assembly level optimization on the MSE on a macroblock is 12 times faster than the optimized C version.

5.7 Putbit

Putbit is one of the encoder functions that flushes out the variable length encoded bit-stream to either the external memory of the C62x or to a file on the PC host. This function normally contributes to about 10% of the encoder time on intra-coding and less than 5% for inter-coding without any optimization. However, when all the other video components are being optimized, this putbit function becomes significant. Therefore, hand-coded assembly optimization has been done on the putbit function. The overall performance of the assembly level optimization on the putbit function is four to eight times faster than the optimized C version.

5.8 Overall Performance Improvement of the Application

There are currently two versions of the MPEG-2 video encoder being implemented on the C62x EVM board. The first is an I-frame only version of the MPEG-2 video encoder which does Intra-coding at SIF resolution. This version currently runs at above real-time rate of 40 frames per second. In this setup, the MPEG-2 video encoder directly captures video frames from a camera at SIF resolution, and encodes the video in real-time (or faster). The newest version of the I-frame MPEG-2 video encoder can encode **15.6 frames per second** at CCIR-601 resolution. The second version of the MPEG-2 video encoder is an IPP-version which does both intra- and inter-coding which significantly improves the compression ratio. The IPP-version of the encoding at **16.2 frames per second** at SIF resolution. As compared to the optimized C-compiler of the I-frame only and the IPP video encoder on C62 EVM board, the assembly-level optimized programs runs more than **30 times** faster.

Table 5 provides a comparison for the MPEG-2 video coding components running on a Pentium 166MHz machine, compared to encoder components running on the C62x platform, at 166MHz, before and after hand-coded assembly optimization.

Video Components (per frame @SIF)	Pentium @166MHz	C62x @166MHz w/o optimization	C62x @166MHz with optimization
SAD	142 ms	274.5 ms	32.6 ms
Quantization	88.8 ms	164.4 ms	8.4 ms
QDCT	160.3 ms	197.6 ms	9.3 ms
VLC	16.4 ms	40.8 ms	12.4 ms
MSE	4.9 ms	9.3 ms	1.1 ms
Putbit	15.3 ms	58.5 ms	8.4 ms
IDCT	197.6 ms	243.5 ms	10.4 ms

Table 5: Summaries the overall performance of video encoding components on different platforms with and without optimization.

Chapter 6

Conclusion and Future Research

In this thesis, we purpose a software only real-time MPEG-2 video encoder on a VLIW processor architecture. Mapping and implementing the real-time MPEG-2 video encoder significantly improved the performance of video coding running on the VLIW architecture processor. The implementation of the video encoder on Texas Instrument's C62x EVM board was tested. The contribution of our research consists of two parts. In the first part, we mapped and modified the existing MPEG-2 video encoder designed to run on PCs onto the C62x platform. We need to initiate the communication between the C62x EVM board and the host PC for transmitting video data to do the video encoding. Memory mapping is one of the key issues that we have to deal with as the memory size and memory transfer is critical to the performance of the video encoding on the C62x VLIW processor. Modification of the program structure and video components is required to run the encoder effectively on the C62x platform. These components included motion estimation, quantization, variable-length encoding, and putbit function. These functions are discussed in Chapter 4. In the second part, we propose assembly-level optimizations to exploit the instruction-level parallelism of the C62x VLIW architecture. This was done to improve the performance of the MPEG-2 video encoder, since the optimized C-compiler did not provide acceptable results for real-time coding. With a maximum throughput of processing eight instructions every CPU cycle, we achieved major speed ups for some of the major video computations, particularly the DCT, SAD, quantization, variable-length encoding, and data I/O. After employing the optimizations, we realized a much faster encoder that can encode more than 15 frames per second (fps) on both I-frame only with CCIR-601 resolution and IPP with SIF resolution, and an even faster than real-time (40 frames per second) on I-frame only video with SIF resolution. As the unoptimized version of our encoder can only encode less than 0.5 fps on the C62x EVM board. All the results are discussed in detail in Chapter 5.

Although this thesis focuses mainly on video encoding on the C62x VLIW architecture, some of the techniques and developments can be also applied to other VLIW architecture processors. Moreover, the performance of the software implemented MPEG-2 video encoder can be further improved by further optimization of other video components and increasing the data transfer throughput using DMA transfer without intervening CPU.

Another interesting research direction is to extend the hardware features of media processors, eg. by incorporating SIMD capability into the VLIW DSP architecture. The Philips Trimedia chip can possibly be one such candidate.

Bibliography

[1] Yuen-Wen Lee, Faouzi Kossentini, Rabab Ward and Mark Smith, *"Towards MPEG4: An Improved H.263-based video coder,"* in Signal Processing Image Communication, Oct. 1997.

[2] VIS extension of Sun Ultra Sparc web site at http://www.sun.com/microelectronic/vis/

[3] Hwelett-Packard PA-RISC web site at http://www.hp.com/

[4] Silicon Graphics web site at http://www.sgi.com/

[5] Texas Instrument web site at http://www.ti.com/sc/docs/products/dsp/c6000/

[6] E. A. Lee, "Programmable DSP architectures: Part I," IEEE Acoust., Speech, Signal Processing Mag., pp. 4-19, Oct. 1988.

[7] P. Faraboschi, G. Desoli, J. A. Fisher, "The Latest Word in Digital and Media Processing," IEEE Signal Processing Mag., pp. 59-85, Mar. 1998.

[8] B. R. Rau, J. A. Fisher, "Instruction-level parallelism," The Journal of Supercomputing 7, pp. 9-50, May 1993.

[9] K. Hwang, "Advanced Computer Architecture: Parallelism, Scalability, Programmability," McGraw Hill Publisher, 1993.

[10] N. Seshan, "High VelociTI Processing," IEEE Signal Processing Mag., pp. 86-100, Mar. 1998.

[11] M. A. Schuette, J. P. Shen, "Exploiting Instruction-Level Parallelism for Integrated Control-Flow Monitoring," IEEE, 1994.

[12] R. P. Colwell, R. P. Nix, J. J. O'Donnel, D. B. Papworth, P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," IEEE, 1988.

[13] J. W. Davidson, S. Jinturkar, "Improving Instruction-level Parallelism by Loop Unrolling and Dynamic Memory Disambiguation," IEEE Proc. of MRICRO-28, 1995.

[14] M. S. Lam, "Software pipeline: An effective scheduling technique for VLIW machines," ACM SIGPLAN'88, pp.318-328, 1988.

[15] R. B. Lee, "Subword parallelism with MAX-2," IEEE Micro. Vol. 16, pp. 51-59, Aug. 1996.

[16] M. J. Flynn, "Very high speed computing systems," Proc. IEEE, vol. 54, pp. 1901-1990, Dec. 1996.

[17] W. Gehrke, K. Gaedke, "Associative controlling of monolithic parallel processor architectures," IEEE Trans. Circuits Syst. Video Technol., vol. 5, pp. 159-169, Apr. 1993.

[18] K. Nadehara, I. Kuroda, M. Daito, T. Nakayama, "Low-power multimedia RISC," IEEE Mirco, vol.15, pp. 20-29, Dec. 1995.

[19] S. Y. Kung, Y. K. Chen, "On architectural styles for multimedia signal processors," Proc. IEEE Workshop Multimedia Signal Processing, Princeton, NJ, June 1997.

[20] D. Zucker, M. Flynn, and R. Lee, "Improving performance for software MPEG player," Proc. Compon, IEEE CS Press, pp. 327-332, 1996.

[21] K. Nadehara, H. J. Stolberg, M. Ikekawa, E. Murata, I. Kuroda, "Real-time software MPEG-1 video decoder design for low-cost, low-power applications," VLSI Signal Processing IX, IEEE, Oct. 1996.

[22] H. J. Stolberg, M. Ikekawa, I. Kuroda, "Code positioning to reduce instruction cache misses in signal processing applications on multimedia RISC processors," Proc. Int. Conf. Acoust., Speech, Signal Processing, pp. 699-702, Apr. 1997.

[23] V. Bhaskaran, K. Konstantinides, "Image and Video Compression Standards: Algorithms, and Architectures," Kluwer Academic Publishers, Boston, 1995.

[24] Thomas Sikora, *"MPEG Digital Video-Coding Standards,"* in IEEE Signal Processing Mag., Sept 1997.

[25] "Video codec for aduiovisual services at px64 kbits," ITU-T Recommendation H.261, Mar . 1993.

[26] "Video coding for low bitrate communication," ITU-T Draft Recommendation H.263, May 1996.

[27] ISO/IEC 11172-1/-2/-3, Information Technology – Coding fo Moving Pictures and Associated Audio for Digital Storage Media at up to About 1.5 Mbit/s: Systems/Video/Audio, 1994.

[28] Yuen-Wen Lee, Faouzi Kossentini, Rabab Ward, *"Efficient MPEG-2 Encoding of Interlaced Video,"* in Signal Processing Image Communication, Oct. 1997.

[29] Alen Docef, Faouzi Kossentini, Rabab Ward, *"Towards a Software Solution for Real-time MPEG-2 Encoding of Interlaced Video,"* in UBC Abstract draft.

[30] J.L.Mitchell, W.B.Pennebaker, C.E. Fogg, D.J.LeGall, *"MPEG Video Compression Standard,"* Chapman & Hall 1996.

[31] Ismaeil Ismaeil, Alen Docef, Faouzi Kossentini, and Rabab Ward, *"Motion Estimation Using Long Term Motion Vector Prediction,"* in UBC Abstract draft.

[32] Khanh Nguyen-Phi, Alen Docef, and Faouzi Kossentini, *"Quantized Discrete Cosine Transform: A combination of DCT and scalar quantization,"* in ICASSP, 1999. Submitted.

[33] *"TMS320C6000 Technical Brief,"* SPRU197D, Texas Instruments Press, Feb. 1999.

[34] "TMS32C6xx Architectures," Texas Instruments Press, 1997.

[35] "TMS32C6201 Evaluation Model Guide," Texas Instruments Press, 1997.

[36] "TMS32C62xx Instruction Set Guide," Texas Instruments Press, 1997.

List of Acronyms

2-D	Two Dimension
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfer Mode
CCIR	International Radio Consultive Committee
CISC	Complex Instruction Set Computer
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DCME	Distortion-Computation Optimized Motion Estimation
DMA	Direct Memory Access
DPCM	Differential Pulse Code Modulator
DSP	Digital Signal Processing
EMIF	External Memory Interface
EVM	Evaluation Model
GOP	Group of Pictures
gops	Giga Operations Per Second
HFC	Hybrid Fiber-Coax
HPI	Host Port Interface
IDCT	Inverse Discrete Cosine Transform
ILP	Instruction-Level Parallelism
I/O	Input/ Output
ISC	Single Cycle Instruction
ISO/IEC	International Organization of Standardization/ International Electrotechnical Commission
MAC	Multiply-Accumulation
MAE	Mean Absolute Error
MB	Macroblock

MC	Motion Compensation
ME	Motion Estimation
MIMD	Multiple Instruction Stream, Multiple Data Stream
MIPS	Million Instructions Per Second
mops	Mega Operations Per Second
MPEG	Motion Picture Expert Group
MSE	Mean Square Error
MV	Motion Vector
NTSC	National Television Systems Committee
PC	Personal Computer
PMV	Predicted Motion Vector
PSNR	Peak Signal to Noise Ratio
QDCT	Quantized Discrete Cosine Transform
RD	Rate-Distortion
RLE	Run Length Expansion
SAD	Sum of Absolute Difference
SBSRAM	Synchronous Burst Synchronous Random Access Memory
SDRAM	Synchronous Dynamic Random Access Memory
SIF	Source Intermediate Format
SIMD	Single Instruction Stream, Multiple Data Stream
SNR	Signal to Noise Ratio
SPMG	Signal Processing and Multimedia Group
SRAM	Synchronize Random Access Memory
tops	Tera Operations Per Second
VLC	Variable-Length Coding
VLD	Variable-Length Decoding
VLIW	Very Long Instruction Word
VLSI	Very Large Scale Integration
YUV	Luminance and Chrominance