OPTIMIZATION AND PORTING OF H.264 VIDEO DECODER FOR TRIMEDIA TM - 1300 PROCESSOR

SYNDICATE

NC MUHAMMAD OWAIS KHAN (LEADER)
NC UMAR KHAN
NC SARDAR ADNAN RAHIM

PROJECT SUPERVISOR:
Lec. IRTIZA ALI

Dissertation submitted for partial fulfillment of requirement of MCS/NUST for the award of the B.E. degree in Telecommunication Engineering.

Department of Electrical Engineering
Military College of Signals
Rawalpindi

APRIL 2004

DEDICATION

Dedicated to our parents who have been a source of constant encouragement for us and to our idols who have given us inspiration throughout our degree.

TABLE OF CONTENTS

CHAPTER 1
ABSTRACT

CHAPTER 2
INTRODUCTION

CHAPTER 3

An Overview Of H.264

CHAPTER 4

An Overview Of Trimedia Tm 1300

CHAPTER 5

Optimization

CHAPTER 6

Development Of Displays

CHAPTER 7
Conclusion

ACKNOWLEDGEMENTS

Completing projects is not a confining effort. It calls for certain interdependence, like patronage, guidance and encouragement. Acknowledgements are thus not formal but a genuine expression of gratitude. Praise is, thus, for Almighty Allah, without whose blessings and guidance we could not have done anything.

We would also like to express our gratitude to Mr. Ahmad Hassan. He remained a source of inspiration through out the duration of the project. Through out the project he provided us with brilliant ideas without which the completion of the poject would not have been possible. We are deeply indebted to Lec. Irtiza Ali for his support and encouragement through out the project even when the odds were against us. We would also like to thank Mr. Murtaza Khan for advising us take the optimization of the H.264 decoder as our Final year project and his invaluable support during the entire duration of the project.

ABSTRACT

H.264 is the latest video coding standard which is a result of the Joint venture of ITU – T and ISO. H.264 has been developed with the aim to provide greater compression as compared to its predecessors without compromising the video quality. H.264 incorporates error resilience to provide superior video quality over IP networks, as the primary utilization of the standard is in video conferencing applications.

Video conferencing solutions are mostly developed as embedded systems which employ the computational power of the DSP processor and its peripherals to achieve the objective. However, DSP processors have VLIW architectures and limited processing power, creating a hindrance in an effective implementation of the H.264 codec on the DSP.

The aim of our project was to Port and Optimize the H.264 video Decoder on the Trimedia TM – 1300 processor. The target optimization level was placed at 8 fps, which is the frame rate in most web based video applications. The optimization achieved in the end of the project was around 6 fps.

The major modules of the decoder were targeted in the optimization. The modules of Variable Length Decoding (VLD), Inverse Discrete Cosine Transform (IDCT) and Motion Compensation (MC) were optimized. The optimizations were carried out by developing new and efficient, compliant, algorithms for the modules, keeping into perspective the VLIW architecture of the DSP. The implementations involve the effective use of the processor's Custom Ops, hence fully utilizing the processing power of the DSP. The optimizations were carried out for the baseline H.264 decoder. A video rendering driver was also developed to display the decoder output on a TV.

The overall increase in the efficiency of the decoder after optimizations is enormous, i.e. from 0.33 fps to 6 fps giving overall increase in the speed of 1800%. The decoder now is capable to be used in web based video applications, hence achieving the aim of the project.

CHAPTER 2

INTRODUCTION

2.1 INTRODUCTION TO VIDEO CODING

2.1.1 WHY IS VIDEO COMPRESSION NEEDED?

Video signals are coded to achieve compression. Compression makes it easier for the video signal to be stored as the memory space it takes becomes smaller. It also becomes convenient for the signal to be transmitted over a transmission channel for Real-Time Video Conferencing and other Multimedia Applications.

To further explain what has been said above, we start with the analog video signal. The analog video signal is mostly the source of a digital video signal as the most common video signal today, is still in the analog form.

The analog signal contains two half pictures which are interlaced on display. The first field contains the even lines while the other one contains the odd lines. Because of this fact, the field rate is twice the frame rate. The PAL video signal contains 25 frames per second or 50 fields per second. In the computer environment, the interlacing is not mostly used today.

The CCIR recommendation 601 defines the standard for the TV industry and the conversion to a digital format is defined in the CCIR-656 recommendation. This digital signal contains the following resolution.

TV System	Active pixels	Active lines	Frame rate (Hz)
PAL	720	576	25
NTSC	720	480	19.97

Table 2.1 Different Video Signal Resolutions

The full resolution of a PAL video signal is $720 \times 576 = 414{,}720$ pixels for one picture. For true color, 24 bits per pixel are needed. Therefore $720 \times 576 \times 24/8 = 1{,}244{,}160$ bytes are needed for encoding one picture. The PAL video signal contains 25 frames per second. Hence in order to encode one second of a video, 31,104,000 bytes are needed. Therefore, we require a bit rate of around 249Mbits per second.

Similarly, in order to store a ninety minutes movie, around 156 GBytes are required. This huge bit-rate cannot be handled by the computer systems being used today. To develop affordable systems for the consumer market, date compression is required.

2.1.2 ALGORITHMS USED FOR VIDEO COMPRESSION

Currently available picture and video compression standards use similar algorithms to code the pictures. There are several algorithms combined to achieve efficient compression. These algorithms are as follows.

2.1.2.1 LOSSLESS AND LOSSY COMPRESSION

Compression techniques are classified into two categories, lossless and lossy approaches. Lossless techniques are capable of recovering the original data perfectly. These algorithms are used in the applications where the perfect recovery of data is essential.

Lossy techniques involve algorithms which recover data similar to the original one. These techniques provide higher compression ratios. Hence these techniques are more often applied in the image and video compression.

A video sequence contains two kinds of redundancies, spatial and temporal. Spatial redundancy occurs because neighboring pixels in each individual picture are related. The pixels in the consecutive frames are also correlated, which leads to substantial temporal redundancy. For any compression algorithm to be effective, it must exploit these redundancies.

The lossy compression algorithms make use of the anatomical characteristics of the human eye. A picture contains some information which is not necessary for its quality. The human system does not treat all the visual information with equal sensitivity. For example the eye is more sensitive for changes in luminance than in chrominance. For this reason it is sufficient to transmit only one chrominance pixel U and V for four luminance pixels Y. This indicates that four adjacent pixels have the same color information but different brightness levels on the display screen.

The human eye is also less sensitive to high frequencies. So it is better to send the lower frequencies more exactly than the higher frequencies.[1]

2.1.2.2 TRANSFORMATION FROM SPATIAL DOMAN TO TIME DOMAIN

The standardized video or image compression techniques use a transformation from the spatial domain to the time domain. This transformation enables the exploitation of the spatial correlation of the pixels by converting them into a set of independent coefficients. This transformation is done on a block basis. The pixels are divided into 8×8 pixel blocks. These blocks are transformed into frequency domain using Discrete Cosine Transform (DCT). DCT is preferred over other transform techniques mainly because the coefficients are in the real domain.

The forward 2-D DCT and the inverse 2-D IDCT are defined as:

DCT

$$F(u,v)=1/4*C(u)*C(v)*\sum_{j=0}^{7}\sum_{k=0}^{7} f(j,k)*[\cos(2*j+1)/4*u*\pi]*[\cos(2*k+1)/4*v*\pi]$$

IDCT

$$F(j,k)=1/4\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)*C(v)*F(u,v)*[\cos(2*j+1)/4*u*\pi]*[\cos(2*k+1)/4*v*\pi]$$

With

$C(w)=1/\sqrt{2}$ with w=0 and

$C(w)=1$ for any other value of w

$F(u,v)$ is the transformed pixel block in the frequency domain.

The result of the DCT is an 8×8 matrix with the frequency coefficients. In the value at the upper left corner F(0,0) is called the DC coefficient. The other coefficients are called the AC coefficients as they represent the frequency which the pixel block contains. The frequency increases from the right to the bottom.

Normally a picture contains many low frequency components. This means that the coefficients are concentrated in the upper left corner of the matrix.

2.1.2.3 QUANTIZATION

Quantization is a process used to reduce the precision of the DCT coefficients. The quantization is done by a division of the integer DCT coefficients by integer quantization values. The quantization values are chosen so as to minimize the distortion in the reconstructed pictures using the anatomical characteristics of the human eye.

The result of the quantization of higher frequencies is mostly zero; therefore all the values are not required to be transmitted. Hence compression is achieved.

2.1.2.4 RUNLENGTH AND HUFFMAN CODING

After the quantization, the matrix contains many zero coefficients particularly in the high frequency part. These can be coded with the help of Run Length coding efficiently. The rum length coding counts the number of zeros unless a non zero number appears. Then one pair of numbers is generated. The first value of the pair is the count of the zero coefficients and the second value is the value of the coefficient which is not zero. For example the run level code (5,20) represents the following values: 0 0 0 0 0 20.

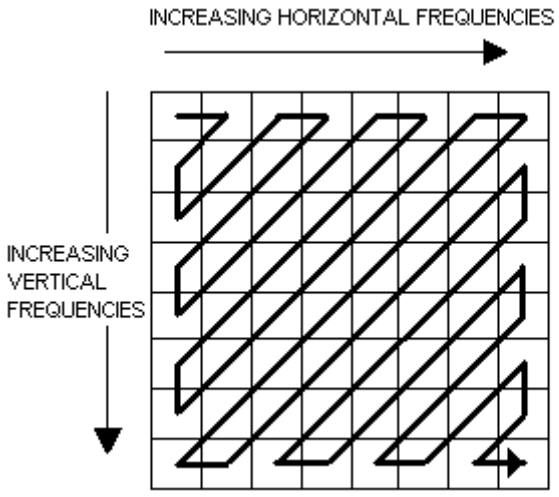


Figure 2.1 Zig-zag scan way

As shown in the above figure the “zig-zag scan” orders the DCT coefficients in ascending order of the spatial frequencies. The frequency increases from left to right and top to bottom. The run length coding is a form of lossless compression. After the run length coding, each run length couple is coded by entropy coding. Entropy coding is also called as “variable length coding” or Huffman coding.

For variable length coding, the appearance probability P_s of each run length couple is investigated. The run length with the largest probability of occurrence is coded with the shortest code and vice versa to achieve compression.

The optimum code length for a symbol L_s is given by;

$$L_s = log_2 (1/P_s)$$

And the entropy, which is simply the average number of bits per symbol, is given by;

$$entropy = \sum_{s} P_s log_2 (1/P_s)$$

The Huffman codes are constructed by pairing two symbols with the lowest probability combining them to a branch of a tree as shown in the figure. The branches of the tree are assigned codes 1 and 0. The tree is developed until every branch is covered. The Huffman code is created by concatenating the bits of the branches, starting from the root in the other direction.

The following example shows the results of the transformation to the frequency domain, the quantization and the run length coding.

********** 8×8 pixel block **********

130	125	133	136	139	149	135	137
119	132	150	150	135	128	124	122
135	136	127	120	122	117	133	137
88	106	133	138	140	134	126	104
142	151	142	134	116	120	125	140
120	113	118	148	165	149	147	130
129	139	141	127	124	120	129	150
132	126	122	121	134	147	157	149

Table 2..2(a) Original 8x8 Pixel Block

DCT RESULTS:

1056	-20	-16	-9	-3	1	-2	3
-10	18	-20	-6	-3	2	6	-3
22	-14	17	9	0	-1	7	0
20	1	-12	6	11	-1	3	-5
-3	-8	0	8	-10	2	6	-4
-18	-12	-9	0	-1	-4	3	5
9	-25	6	31	8	3	-7	3
8	30	60	-1	0	-9	2	-5

Table 1.2(b) Frequency Domain Coefficients

QUANTIZED RESULTS:

132	-1	-1	0	0	0	0	0
-1	1	-1	0	0	0	0	0
1	-1	1	0	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-1	0	0	0	0	0	0	0
0	-1	0	1	0	0	0	0
0	1	2	0	0	0	0	0

Table 2.2(c) Frequency Coefficients after Quantization

RESULTS OF RUN LENGTH CODING:

(0,132) (0,-1) (0,-1) (0,1) (0,1) (0,-1) (1,-1) (0,-1) (0,1) (2,1) (7,-1) (13,-1) (1,1) (10,1) (0,2)

****

These run length couples are coded using Huffman coding. This example shows this algorithm with the previously calculated symbol probabilities.

Run Level code	***Count***	***Probability(P_s)***	***Code***
0,-1	4	0.266	1
0,1	3	0.200	01
0,132	1	0.066	00111
0,2	1	0.066	00110
1,1	1	0.066	00101
1,-1	1	0.066	00100
2,1	1	0.066	00011
7,-1	1	0.066	00010
10,1	1	0.066	00001
13,-1	1	0.066	00000

Table 2.3 Huffman code for given probabilities

For the above data, the Huffman code can be developed by the following tree:

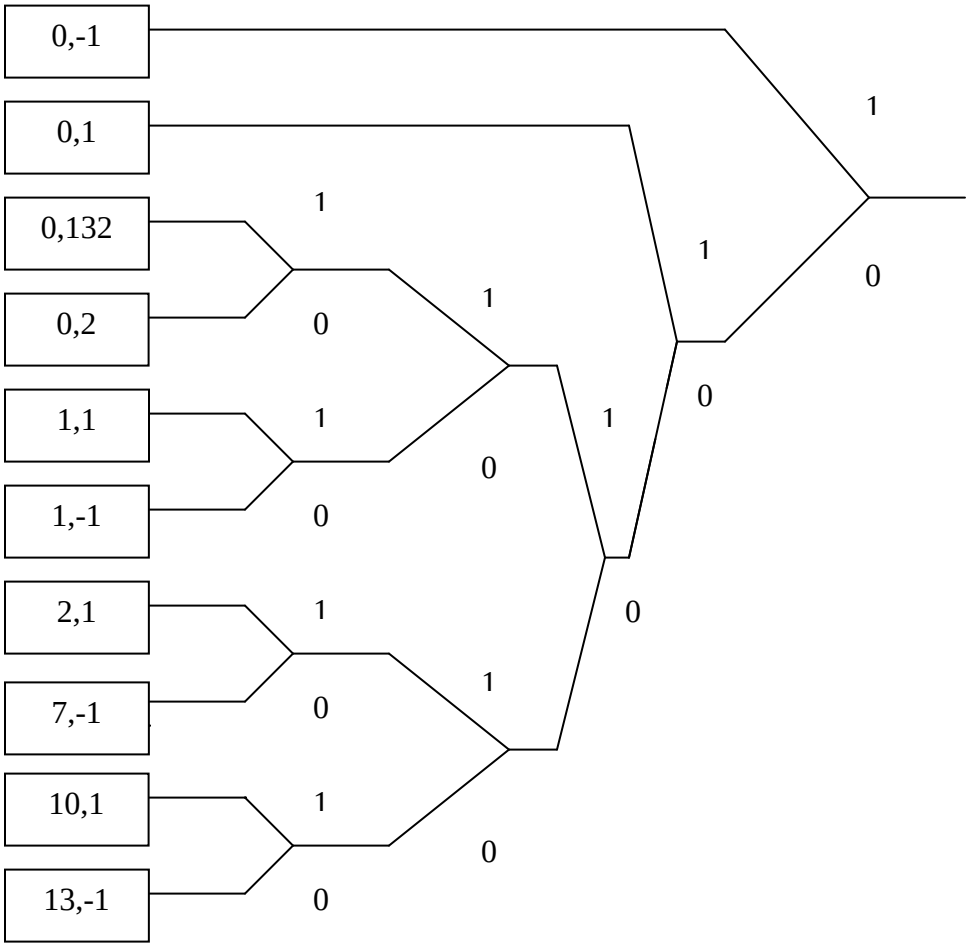


Figure 2.2 Huffman Coding

The run length couples are coded as follows:

00111 1 1 01 01 1 00011 00010 00000 00101 00001 00110 (50 bits)

This example shows a compression factor of 512/50 =10.24.

2.1.2.5 SPECIAL ALGORITHM FOR ENCODING A VIDEO SEQUENCE

All the algorithms that have been discussed are applied on each frame and are used in the JPEG. But we have other algorithms that shall compress video sequences and not only single pictures like JPEG.

There is only a very small difference between the consecutive frames in a video sequence. This temporal redundancy is exploited by a technique called motion compensation based on prediction. Since frames are closely related, it can be assumed that a current picture can be modeled as a translation of a previous picture. The picture is divided into macroblocks of 16×16 pixels. The motion compensation delivers a motion vector to a macroblock which closely matches the previous macroblock. The difference between the macroblocks is coded using the DCT, quantization, run length and variable length

The motion vectors of neighboring macroblocks are mostly similar, so the DPCM coding results in a shorter code.

In this section, only an overview of these video coding algorithms has been given. The complete detail of these algorithms will follow later on.

2.1.3 VIDEO CODING STANDARDS

2.1.3.1 MOTION JPEG

Motion JPEG is not an official standard. It simply means that each frame of a video sequence is coded using the JPEG standard and the JPEG encoder and decoder work in real time.

2.1.3.2 MPEG – 1

For the compression of real time video signals, ISO named a committee called Motion Picture Expert Group (MPEG). There first standard was released in 1993 called MPEG 1. The purpose of MPEG 1 is that a video signal and its associated audio can be compressed to a bit rate of 1.5 Mbits per second with an acceptable quality.

MPEG compresses the frames in three different ways.

- I Frames: These are independently compressed just like the JPEG standard.
- P Frames: These are compressed with reference to the previous frame.
- B Frames: These are compressed with reference to both previous and next frame.

2.1.3.3 MPEG –2

MPEG 2 was released in1994 and is a development to MPEG 1. It was developed for the digital television broadcasting. It is intended for higher data rates, larger picture sizes and interlaced frames. This standard is also able to use three different color formats which are 4:4:4, 4:2:2 and 4:2:0.

This standard has backward compatibility with MPEG-1. It means that the MPEG 2 decoder can decode MPEG 1 data streams.

MPEG 2 also has the provision of secure data transfer. A fast encryption algorithm is described in the standard. Any unauthorized modification can thus be discovered.

The MPEG 3 standard was targeted towards HDTV. But these features were already included in the MPEG 2 standard. Hence, the MPEG 3 standard was altogether dropped and the next standard was MPEG 4.[2]

2.2 INTRODUCTION TO OPTIMIZATION

2.2.1 OVERVIEW

The following Flowchart gives a high level diagram of the complete optimization process.

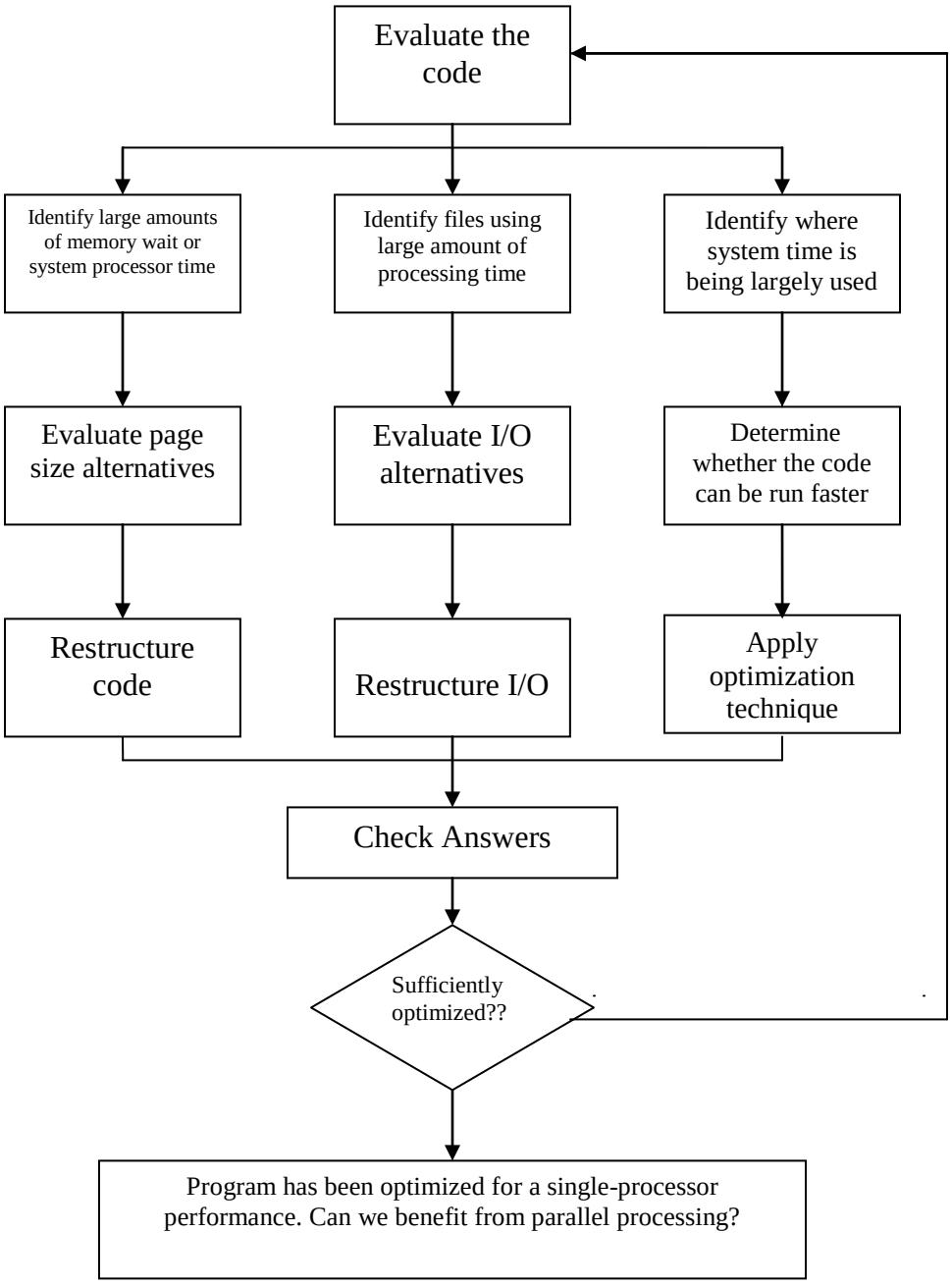


Figure 2.4 Overview of Optimization Process

In practice, optimization is an incremental and subjective process that follows this general flow:

Measure: Different methods and tools are used to measure the current performance of the code as it executes. Then the most significant problem areas are identified where most of the resources are being utilized.

Evaluate: Then the problem areas are evaluated in detail to determine the best method of addressing it.

Apply: Then one of the optimization techniques is applied to improve the performance.

Test: Then answers after each code modification are checked to avoid regression. The performance of the modified code is determined to determine as to whether any improvement has in fact taken place.

Repeat: Then the above steps are repeated until the required results are obtained.[1]

Optimization is an iterative process that requires a lot of recompilation and retesting and has no fixed end point. The first step in optimizing a program is to evaluate its overall performance. This allows you to decide where to focus optimization efforts.

When you compile and execute a program, it typically will be dominated by one of the following activities:

- Memory management
- Input/output (I/O) processing
- Processor computation

A program is considered to be memory bound, I/O bound, or processor bound when its performance is limited by problems with the dominant activity. Use the procedures described in this chapter to help identify these problems.

When program performance is limited by memory allocation issues, the program is considered *memory bound*.

If a program spends most of its time performing input/output(I/O), it is considered I/O bound and I/O optimization can offer significant savings in elapsed time.

If the code spends most of its time performing processor calculations, it is considered *processor bound*. If your code is neither memory bound nor I/O bound, then it must be processor bound and then the focus should be on improving single-processor performance.

[1] **Optimizing Applications on the Cray X1™ System - S-2315-51**

Now we discuss one by one different techniques of optimization.

2.2.2 HAND TUNING OPTIMIZATIONS

There are a number of reasons why a compiler cant perform optimization itself. Sometimes the compiler won't perform optimizations even though it can. The compiler assigns a higher priority to producing consistent and correct code, than optimizing performance.

Sometimes a potential compiler optimization could result in errors: (A + B) + C does not always equal A + (B + C). Certain optimizations can re-associate floating point operations and potentially accumulate into significant errors. The difference is usually small as shown in the following example. Finite precision floating-point numbers do not always associate.

```
      (A + B) + C        A + (B + C)

3.483986447771696 3.483986447771695
4.467320344550364 4.467320344550365
                  ^
                  |
        --- 15th decimal place
```

Sometimes the difference is huge: Finite floating-point exponents can lead to "Catastrophic Cancellation" shown in the following example.

```
Result of the original loop is 0.000000
Result of the interchanged loop is 143166118.018429
```

In cases where the compiler can't or won't perform optimizations, or when it produces incorrect results, hand-tuning becomes necessary.

The remainder of this tutorial concerns "hand tuning" optimization techniques in the areas of:

- Arrays and memory management
- Loops
- Arithmetic operations
- I/O

2.2.2.1 MEMORY CONSIDERATIONS

2.2.2.1.1 MEMORY HEIRARCHY

Many optimizations target efficient use of memory resources. Understanding memory related factors is highly useful in understanding optimization techniques. Two aspects of memory that are critical to understanding optimization are:

The Memory Hierarchy, which involves the different physical "layers" of memory and their different characteristics, and the Spatial and Temporal Locality involving the program characteristics which will improve your code performance.

Most computer architectures share a similar memory structure. Memory is organized within a hierarchy with the fastest, smallest, most expensive memory components at the top, and the slowest, largest, least expensive components at the bottom.

The Memory Hierarchy is necessary because the main memory is about 50 times too slow to keep up with the CPU. Now we discuss each component in the memory hierarchy in detail.

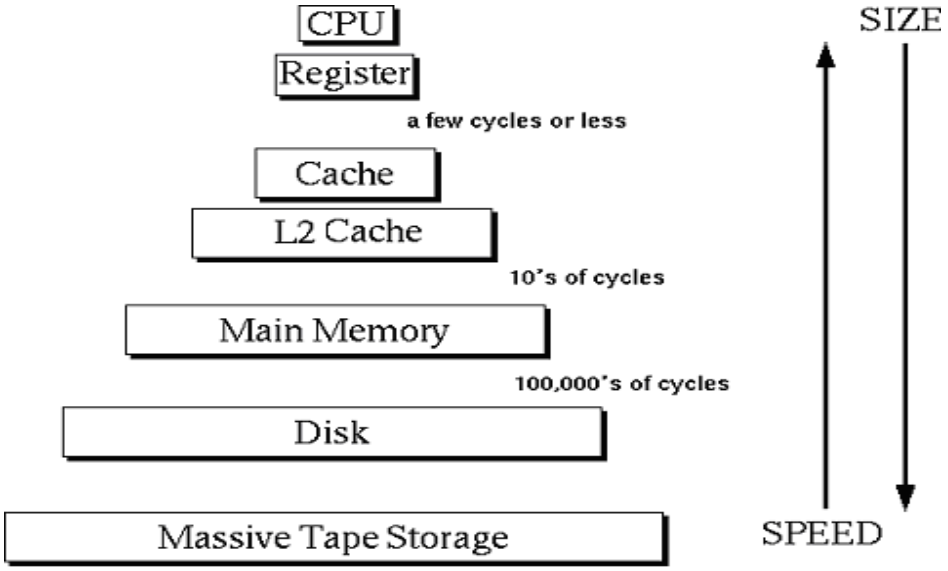


Figure 2.5 Memory Hierarchy

a) CPU

The Central Processor Unit is where the execution of computer instructions takes place. Modern CPUs usually contain several different functional components, such as floating point units, fixed point (integer) units, branch control units, etc.

b) REGISTER

Registers are the memory units with immediate access by the CPU. Registers comprise the fastest and smallest level of the memory hierarchy. These are usually four to eight bytes in size and less than 100 in number.

c) CACHE

Cache is a very fast, but small, area of memory. Cache is typically measured in kilo or mega bytes and may also be subdivided into layers (L1, L2). CPU access to data in cache usually requires only a single CPU cycle. Cache is usually organized into "lines", with each line being measured in bytes (32, 64, 128, 256...)

d) MAIN MEMORY

Main Memory is much larger than cache. It is usually measured in mega or gigabytes. However, access to main memory is usually an order of magnitude slower than cache, being measure in tens of cycles. Main memory is organized into "pages" (4096 bytes on the SP).

e) DISK (Virtual Memory)

Disk is usually several orders of magnitude slower and larger than main memory. Accessing data on disk can cost 100,000s of cycles.

f) MASS STORAGE (Tape)

Mass storage is virtually unlimited in size. Access to data on tape is measured in a very large amount of time compared to main memory.

2.2.2.1.2 EFFICIENT MEMORY USE

Programs should be designed so that a high percentage of accesses are made to the higher levels of memory. To accomplish this, the programmer should strive for two important program characteristics:

a) Spatial Locality: If location X in memory is currently being accessed, it is likely that a location near X will be accessed next.

b) Temporal Locality: If location X in memory is currently being accessed, it is likely that location X will soon be accessed again.

Taking advantage of spatial and temporal locality translates will minimize cache misses, TLB misses, and page faults through data reuse.

2.2.2.2 ARRAY AND MEMORY MANAGEMENT OPTIMIZATIONS

2.2.2.2.1 STRIDE MINIMIZATION

In a loop, stride is defined as the distance between successively accessed elements of a matrix in successive iterations of the loop. Processing matrices with minimal stride takes advantage of spatial locality.

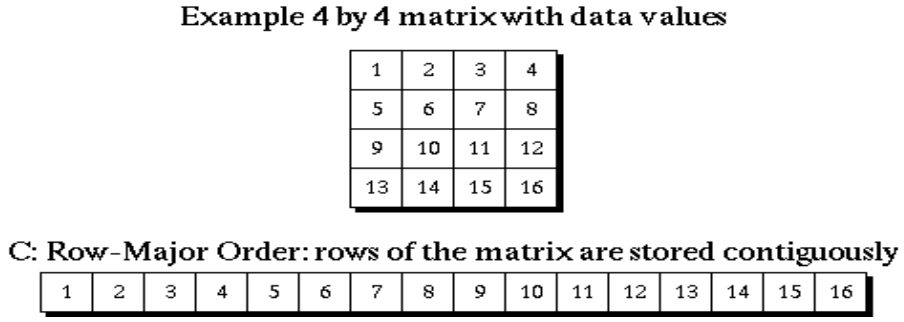


FIGURE 2.6 Array Allocation in C

When any element is referenced, and needs to be brought into cache from memory, an entire cache line worth of data is brought with it. Small stride exploits the extra data brought in with the cache line, since the next data to be processed is already in cache. Large stride does not exploit the extra data brought in with the cache line, and in fact, may require a new cache line to be loaded for each element accessed. To ensure minimal stride in C, the innermost loop's induction variable should be the rightmost array subscript.

2.2.2.2.2 ARRAY PADDING

Array Padding is a technique that can be used to reduce cache misses due to set associativity in arrays that are sized by a power of two. To pad an array, simply increase the last dimension in C/C++. The optimal amount to pad varies, but should be at least the number of elements that fit in one line of cache.

2.2.2.3 LOOP OPTIMIZATIONS

2.2.2.3.1 LOOP FUSION

Loop Fusion involves the merging of the statements in several loops into a single loop. In this way, the loop overheads are reduced, better instruction overlap is allowed and Cache misses can be decreased if both loops reference the same array. But the disadvantage is that it has the potential to *increase* cache misses due to cache set associativity effects when the fused loops contain references to multiple arrays and the starting elements of those arrays map to the same cache line and when the fused loops access arrays that take up a large portion of cache.[3]

2.2.2.3.2 INVARIANT IF CODING FLOATING

The IF statements that do not change from iteration to iteration should be moved out of the loop. The compiler can usually detect and perform this optimization except when;

- The loops contain calls to procedures and the compiler is pessimistic about aliasing.
- And also when Variable-bounded loops that may never get entered.
- Complex loops where the invariance can not be determined by the compiler.

2.2.2.3.3 LOOP DEFACTORIZATION

Loops involving multiplication by a constant can be rewritten so that the multiplication takes place outside the loop. The defactorized loop allows the power processor to perform a single FMA instruction. For more complex loops, defactorizing should target balancing additions and multiplications to provide the floating point unit with a stream of FMA's.

2.2.2.3.4 LOOP UNROLLING

This technique is used where a loop runs for a known number of times. By using this technique, the compiler does not have to check the loop condition again and again and the

number of operations for a loop is significantly decreased, hence achieving optimization. Also, data dependence delays can be reduced or eliminated. Loop overheads may be reduced.

2.2.3 INPUT/OUTPUT OPTIMIZATIONS

Most of the previously discussed optimizations will be of little benefit if your program is I/O bound. I/O slows down your program because:

- I/O is to the order of magnitude slower than internal memory accesses
- I/O routines consume CPU cycles themselves

Therefore, all unnecessary I/O should be eliminated. All I/O statements should be moved outside the loops. Unformatted binary should be used wherever possible because:

- Formatted I/O requires that library calls be made to convert the binary representation to human readable format and then converted back again to binary format when the processor must process the data.
- Formatted I/O can also result in lost precision and rounding errors.
- Binary data is smaller - requires less physical I/O time to process and less disk space to store.

Also the data should be accessed from the memory. All the data should be read from the memory before the program starts. [4]

CHAPTER 3

AN OVERVIEW OF H.264

3.1 INTRODUCTION AND BACKGROUND

H.264, also known as “MPEG-4 part 10” is a high compression video codec standard. It a standard which has been jointly developed by the International Telecommunication Union(ITU) Video Coding Experts group and the International Standards Organization (ISO) Moving Picture Expert Group. The main goals of the H.264/AVC standardization effort have been enhanced compression performance and provision of a "network-friendly" video representation addressing "conversational" i.e real-time applications for example video conferencing and "non-conversational" applications like storage, broadcast, and streaming of video. H.264 has achieved a significant improvement in rate-distortion efficiency relative to existing standards [1].

Broadcast television and home entertainment have been revolutionized by the advent of digital TV and DVD-video. These applications and many more were made possible by the standardization of video compression technology. The next standard in the MPEG series, MPEG4, is enabling a new generation of internet-based video applications whilst the ITU-T H.263 standard for video compression is now widely used in videoconferencing systems.

MPEG4 (Visual) and H.263 are standards that are based on video compression (“video coding”) technology from circa. 1995. The groups responsible for these standards, the Motion Picture Experts Group and the Video Coding Experts Group (MPEG and VCEG) are in the final stages of developing a new standard that promises to significantly outperform MPEG4 and H.263, providing better compression of video images together with a range of features supporting high-quality, low-bitrate streaming video. The history of the new standard, “Advanced Video Coding” (AVC), goes back at least 7 years.

After finalizing the original H.263 standard for video-telephony in 1995, the ITU-T Video Coding Experts Group (VCEG) started work on two further development areas: a “short-term” effort to add extra features to H.263 (resulting in Version 2 of the standard) and a “long-term” effort to develop a new standard for low bit rate visual communications. The long-term effort led to the draft “H.26L” standard, offering significantly better video compression efficiency than previous ITU-T standards. In 2001,

the ISO Motion Picture Experts Group (MPEG) recognized the potential benefits of H.26L and the Joint Video Team (JVT) was formed, including experts from MPEG and VCEG. JVT's main task is to develop the draft H.26L "model" into a full International Standard. In fact, the outcome will be two identical) standards: ISO MPEG4 Part 10 of MPEG4 and ITU-T H.264. The "official" title of the new standard is Advanced Video Coding (AVC); however, it is widely known by its old working title, H.26L and by its ITU document number, H.264 [5].

3.2 H.264 CODEC

Following are some of the main features that make H.264 superior over some of the previous video coding standards.

Quarter-sample-accurate motion compensation: H.264 codec uses quarter-sample-accurate motion compensation as in H.263 with further enhancements and reduced complexity.

Display order and referencing independency: The decoder may choose the most efficient way of displaying pictures for motion compensation referencing improving overall performance.

Weighted prediction: Motion-compensated prediction signal may be weighted and offset by the encoder, improving performance in scenes containing fades.

Small block-size transform: H.264 is based primarily on 4x4 transform, which positively influences the quality of certain scenes.

Hierarchical block-size: Even though the default block-size transform is 4x4, the standard is flexible enough for bigger block-size transforms, such as 8x8 or 16x16, for improved performance in certain scenes.

Short word-length transform: H.264 reduces computation complexity requiring only 16-bit processing.

Exact-match inverse transform: As opposed to most previous standards, all decoders processing video stream encoded using H.264 will produce exactly the same picture.

Arithmetic and context-adaptive entropy coding: H.264 codec uses advanced entropy coding methods improving overall performance.

Parameter set structure: The separation and of the parameter set structure from the remaining data and special handling makes it less prone to information loss.

In common with earlier standards (such as MPEG1, MPEG2 and MPEG4), the H.264 draft standard does not explicitly define a CODEC (encoder / decoder pair). Rather, the standard defines the syntax of an encoded video bit-stream together with the method of decoding this bit-stream. In practice, however, a compliant encoder and decoder are likely to include the functional elements shown in Figure and Figure 2-2. Whilst the functions shown in these Figures are likely to be necessary for compliance, there is scope for considerable variation in the structure of the CODEC. The basic functional elements (prediction, transform, quantization, entropy encoding) are little different from previous standards (MPEG1, MPEG2, MPEG4, H.261, H.263); the important changes in H.264 occur in the details of each functional element. The Encoder (Figure 2-1) includes two dataflow paths, a "forward" path (left to right, shown in blue) and a "reconstruction" path (right to left, shown in magenta). The dataflow path in the Decoder (Figure 2-2) is shown from right to left to illustrate the similarities between Encoder and Decoder.

3.2.1 H.264 ENCODER (forward path)

An input frame F_n is presented for encoding. The frame is processed in units of a macroblock (corresponding to 16x16 pixels in the original image). Each macroblock is encoded in **intra** or **inter** mode. In either case, a prediction macroblock P is formed based on a reconstructed frame. In Intra mode, P is formed from samples in the current frame n that have previously encoded, decoded and reconstructed (uF'_n in the Figures; note that the **unfiltered** samples are used to form P). In Inter mode, P is formed by motion-compensated prediction from one or more reference frame(s). In the Figure, the reference frame is shown as the previous encoded frame F'_{n-1}; however, the prediction for each macroblock may be formed from one or two past or future frames (in time order) that have already been encoded and reconstructed.

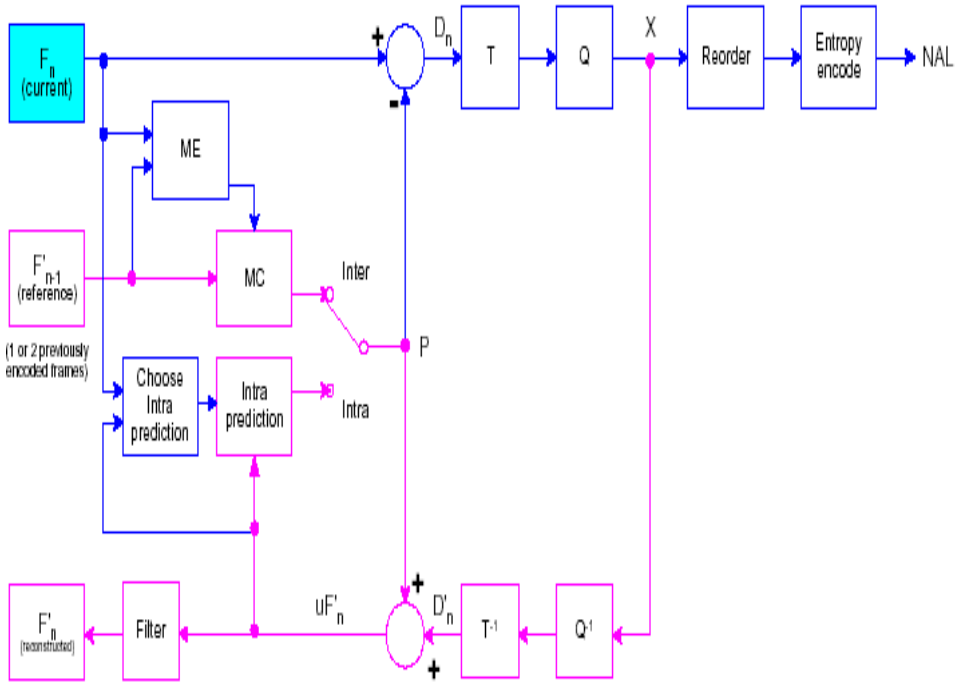


Figure 2.1 Encoder

The prediction P is subtracted from the current macroblock to produce a residual or difference macroblock D_n. This is transformed (using a block transform) and quantized to give X, a set of quantized transform coefficients. These coefficients are re-ordered and entropy encoded. The entropy encoded coefficients, together with side information required to decode the macroblock (such as the macroblock prediction mode, quantizer step size, motion vector information describing how the macroblock was motion-compensated, etc) form the compressed bitstream. This is passed to a Network Abstraction Layer (NAL) for transmission or storage.

3.2.2 H.264 ENCODER (reconstruction path)

The quantized macroblock coefficients X are decoded in order to reconstruct a frame for encoding of further macroblocks. The coefficients X are re-scaled (Q-1) and inverse transformed (T-1) to produce a difference macroblock $D_{n'}$. This is not identical to the original difference macroblock D_n; the quantization process introduces losses and so $D_{n'}$ is a distorted version of D_n. The prediction macroblock P is added to $D_{n'}$ to create a reconstructed macroblock uF'_n (a distorted version of the original macroblock). A filter is

applied to reduce the effects of blocking distortion and reconstructed reference frame is created from a series of macroblocks F'_n.

3.2.3 H.264 DECODER

The decoder receives a compressed bitstream from the NAL. The data elements are entropy decoded and reordered to produce a set of quantized coefficients X. These are rescaled and inverse transformed to give $D_{n'}$ (this identical to the $D_{n'}$ shown in the Encoder). Using the header information decoded from the bitstream, the decoder creates a prediction macroblock P, identical to the original prediction P formed in the encoder. P is added to $D_{n'}$ to produce uF'_n which this is filtered to create the decoded macroblock F'_n.

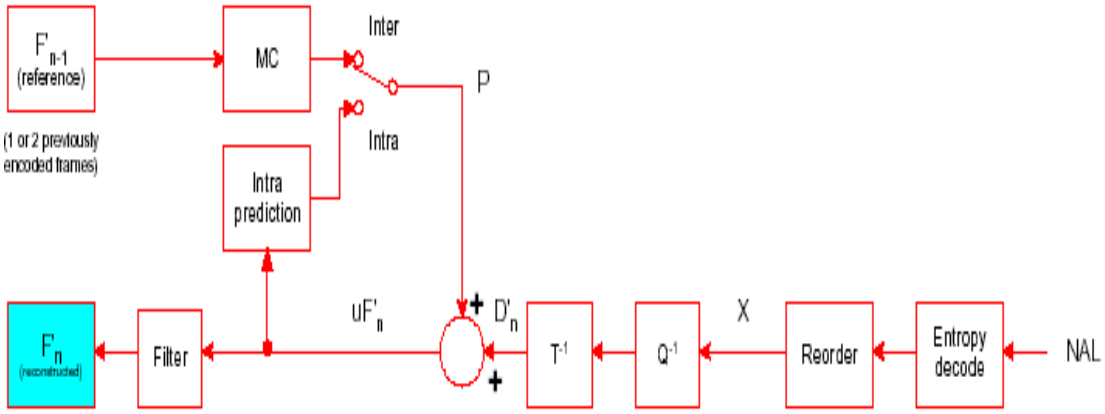


Figure 2.2 H.264 Decoder

It should be clear from the Figures and from the discussion above that the purpose of the reconstruction path in the encoder is to ensure that both encoder and decoder use identical reference frames to create the prediction P. If this is not the case, then the predictions P in encoder and decoder will not be identical, leading to an increasing error or "drift" between the encoder and decoder.[6]

CHAPTER 4

AN OVERVIEW OF TRIMEDIA TM 1300

4.1 INTRODUCTION

TM1300 is a media processor for high-performance multimedia applications that deal with high-quality video and audio. These applications can range from low-cost, dedicated systems such as video phones, video editing, digital television, security systems or set-top boxes to reprogrammable, multipurpose plug-in cards for personal computers. TM1300 easily implements popular multimedia standards such as MPEG-1 and MPEG-2, but its orientation around a powerful general-purpose CPU (called the DSPCPU) makes it capable of implementing a variety of multimedia algorithms, both open and proprietary. TM1300 is also easily configured in multiple processor configurations for very high-end applications.

4.2 TRIMEDIA VLIW ARCHITECTURE

TriMedia's DSPCPU family delivers exceptional performance and high-level language programmability for multimedia applications through the use of its VLIW (very long instruction word) architecture. TriMedia's VLIW architecture combines innovations in compiler and software design with advances in logic design. Rather than supporting only general purpose code as other vendors do, TriMedia's VLIW architecture supports multimedia-specific code that takes full advantage of the architecture.

This code (along with a C/C++ compiler, Debugger, and other tools) is supplied to you in the form of application libraries as part of the TriMedia Software Development Environment (SDE).

4.2.1 OVERVIEW

The TriMedia architecture is based on a three-level hierarchy of operators:

- Instructions
- Operations
- RISC operations

One instruction may contain five operations. Each operation may execute multiple arithmetic operations.

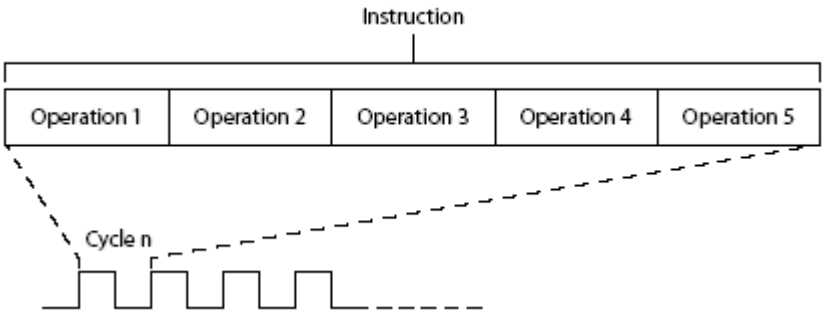


Figure 4.1 Operationd In A Clock Cycle

An example of one such operation is the command ifir(a,b). This command contains a total of three arithmetic operations: *two m*ultiplications and *one* addition (a_{HI} * a_{LO} + b_{HI} * b_{LO}). Up to five operations including two ifir commands can be issued in each machine cycle.

The ability of TriMedia's VLIW architecture to execute multiple operations in parallel gives it a big advantage over traditional RISC and CISC architectures found in current mass-market microprocessors.

4.2.2 FUNCTIONAL UNIT ASSIGNMENT

Although the VLIW architecture allows for five operations to be executed per instruction, most operations cannot use just any of the slots because each functional unit of the TriMedia CPU is assigned to specific issue slots in a TriMedia VLIW instruction. Table 1 gives the functional unit assignments.

Because of the number of available functional units and their assignment, some operations may have to wait for one or more cycles before they are executed. This means that in some cases not all issue slots are used. Good use of issue slots is one of the purposes for software code optimization.

Functional Unit	Quantity	Latency[A]/Delay[B]	Recovery[C]	Slot Assignment				
				1	2	3	4	5
Constant	5	1	1	✓	✓	✓	✓	✓
Integer ALU	5	1	1	✓	✓	✓	✓	✓
Load/Store	2	3	1				✓	✓
DSP ALU	2	2	1	✓		✓		
DSP MUL	2	3	1		✓	✓		
Shifter	2	1	1	✓	✓			
Branch	3	3	1		✓	✓	✓	
Int/Float MUL	2	3	1		✓	✓		
Float ALU	2	3	1	✓			✓	
Float Compare	1	1	1			✓		
Float sqrt/div	1	17	16		✓			

A. Clock cycles between the issuance of an operation and availability of its results
B. Clock cycles between the execution of a branch instruction and the branching taking place
C. Minimum number of clock cycles between successive operations

Table 4.1 Functional Unit Assignments

4.3 ADVANTAGES OF VLIW ARCHITECTURE

The beginning instruction set architecture (the processor programming model) must be distinguished from implementation (the physical chip and its characteristics). VLIW microprocessors and superscalar implementations of traditional instruction sets share some characteristics such as multiple execution units and the ability to execute multiple operations simultaneously.

The techniques used to achieve high performance are different because the parallelism is explicit in VLIW instructions, but must be discovered by hardware at run time by superscalar processors. VLIW implementations are simpler for very high performance. Just as RISC architectures permit simpler, cheaper high-performance implementations than do ClSC architectures, VLIW architectures are simpler and cheaper than RlSC architectures because of hardware simplifications. However, they require more compiler support.

The TriMedia VLIW architecture optimizes parallelism at compile time using its sophisticated compilation system. This makes the core less complicated (and feasible) and does not require specialized scheduling hardware at run-time. As a result, the TriMedia processors have much simpler control logic than other parallel designs and can run at higher clock speeds.

4.4 PERFORMANCE

The best performance measurement is actual application measurement. Experience shows that complex DSP algorithms ported to TriMedia take between 1.0 and 0.5 times the number of instruction cycles as on other Digital Signal Processors. However, while these algorithms are optimized in C for TriMedia, DSPs are traditionally optimized in assembly language.

In some algorithms, very high peak performance can be achieved. This is one of the keys to the optimization of the MPEG decoding algorithm. Because as many as 4 “RISC” operations can be performed in one TM operation, and because up to five TM operations can be executed at the cycle rate of 100 MHz, peak execution rates of 500 million operations per second are possible.

The mixture of programmability in C and efficient signal processing loops makes Tri- Media particularly suited for the complex signal processing required by today’s multimedia.

4.5 TRIMEDIA SOFTWARE STREAMING ARCHITECTURE

In a software-driven system like TriMedia, it is important that the authors of diverse components agree upon some ground rules governing the connections between components. This set of rules has been made available in the TriMedia Software Streaming Architecture (TSSA).

The TSSA:

- Defines formats and data structures to describe common multimedia data types.
- Defines procedures to start up and shut down components.
- Defines a method of connection between data sources and data sinks.
- Describes rules to determine who owns memory allocated by components.

4.6 TRIMEDIA HARDWARE COMPONENTS

This section describes the hardware components of the TriMedia system:

- The TriMedia Processor
- The TriMedia Board

4.6.1 THE TRIMEDIA PROCESSOR

The TriMedia processor can be used as a stand-alone processor, as a coprocessor to a more traditional CPU, or as one of a group of TriMedia chips arranged in a multi-processing configuration. The 32-bit variants of the TriMedia processor have a number of special features that help to accelerate target applications. Their DMA-driven I/O units operate independently to format data. Peripheral units are included for video in/out and audio in/out. Additionally, independent DMA driven coprocessors are available to perform key multimedia operations.

For example, the image coprocessor copies images from the synchronous DRAM (SDRAM) to the host's video frame buffer while simultaneously performing scaling and color space conversion. A peripheral block for Variable Length Decoding (VLD) assists in the decompression of MPEG-1 and MPEG-2 data streams.

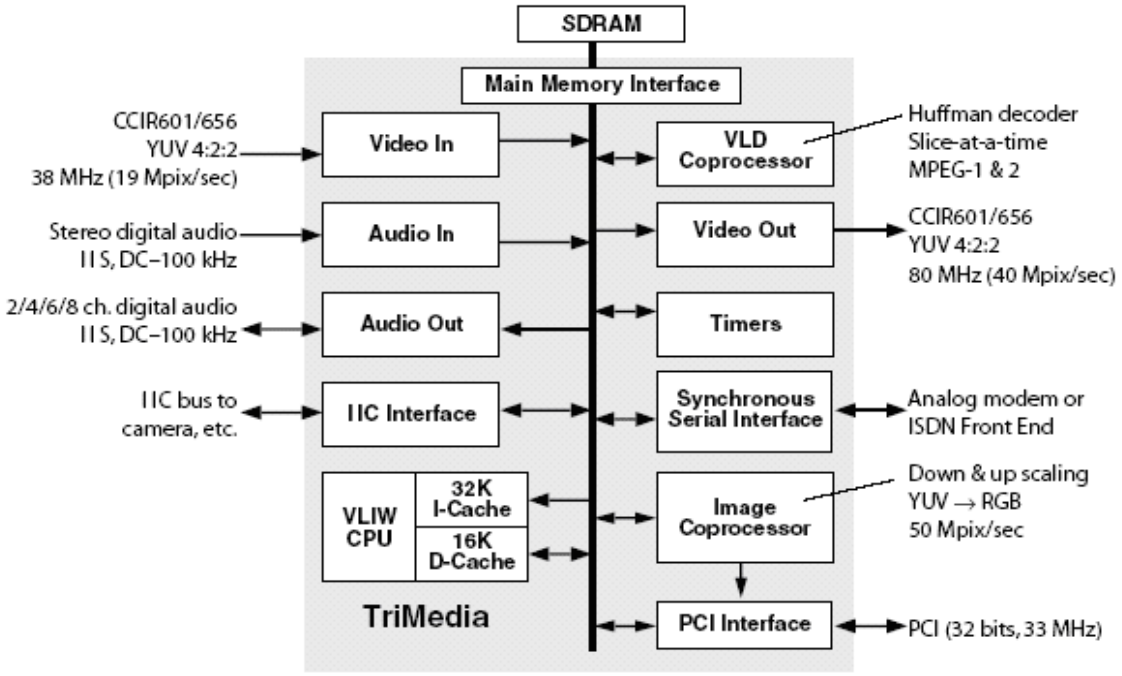


Figure 4.2 TriMedia Block Diagram

4.6.2 TRIMEDIA BOARD

The TriMedia board allows Windows 95, Windows NT, and Macintosh platforms to take advantage of the TriMedia processor. The TriMedia board is installed in an available PCI slot in the target platform.

The TriMedia SDE includes a board support package (BSP) that functions as a driver interface for the TriMedia board. If you create your own board, you will have to create your own BSP.

The BSP enables you to run all the TriMedia examples that come with the TriMedia SDE.

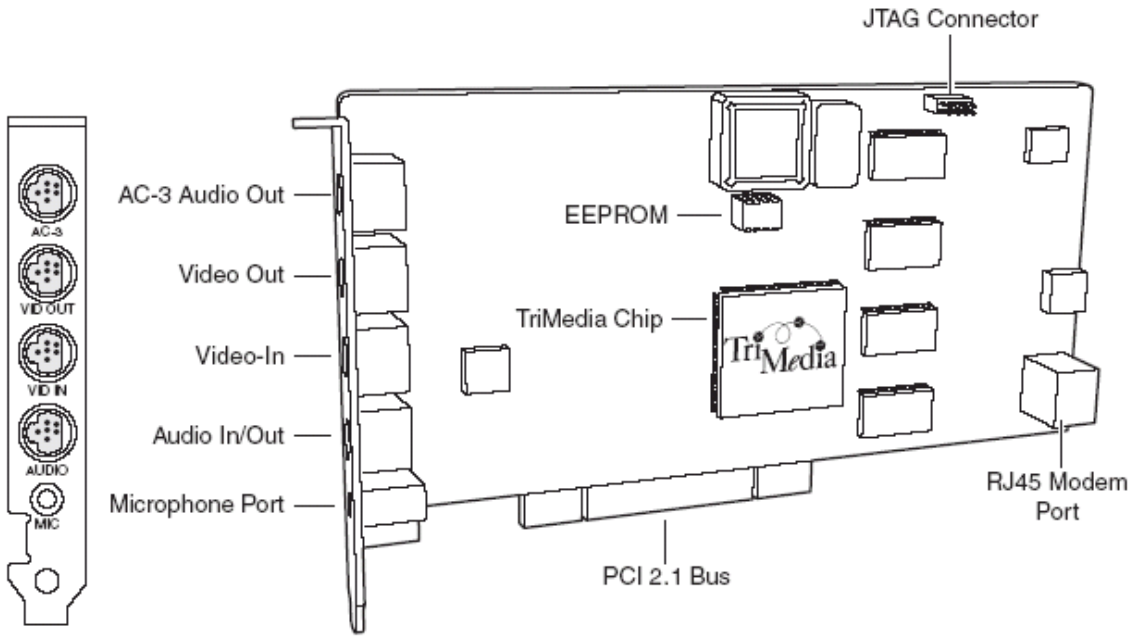


Figure 4.3 The TriMedia Iref Boards And Ports

4.6.3 TRIMEDIA SOFTWARE COMPONENTS

The TriMedia processor is supported by a robust, open TriMedia Software Development Environment (SDE) that speeds creation of highly optimized multimedia applications entirely in the C and C++ programming languages. The TriMedia SDE provides a comprehensive suite of multimedia libraries and system software tools to compile and

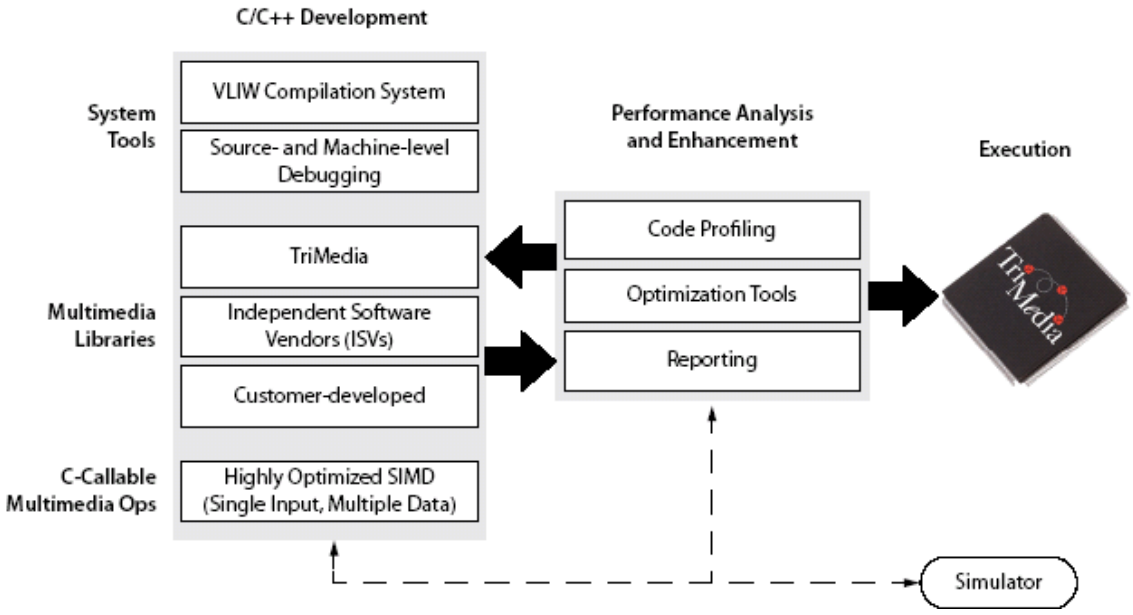


Figure 4.4 TriMedia Software Components

debug multimedia applications, analyze and optimize performance, and simulate execution on the TriMedia processor.[7]

CHAPTER 5

OPTIMIZATION

In this chapter, we shall discuss the methods with the help of which the optimization process was carried out. All the previous chapters were meant to provide the reader with a strong knowledge base to understand the complex procedures carried out. These procedures are discussed in detail in this chapter. This chapter has been written in such a way that first, the method of implementation of a certain block is given. After that, the methodology that has been followed by us to achieve optimization has been discussed.

5.1 OPTIMIZATION OF VLD

5.1.1 VARIABLE LENGTH DECODING IN H.264

In H.264, above the slice layer, syntax elements are coded as fixed or variable length binary codes. At the slice layer and below, elements are coded using either variable-length codes (VLCs) or context-adaptive arithmetic coding (CABAC) depending on the entropy encoding mode. However, we shall limit our discussion to VLCs.

5.1.2 VARIABLE LENGTH CODING (VLC)

When entropy_coding_mode is set to 0, residual block data is coded using a context-adaptive variable length coding (CAVLC) scheme. This is the method used to encode residual, zig-zag ordered 4x4 (and 2x2) blocks of transform coefficients. CAVLC is designed to take advantage of several characteristics of quantized 4x4 blocks:

1. After prediction, transformation and quantization blocks are typically sparse (containing mostly zeros). CAVLC uses run-level coding to compactly represent strings of zeros.

2. The highest non-zero coefficients after the zig-zag scan are often sequences of +/-1. CAVLC signals the number of high-frequency +/-1 coefficients (“Trailing 1s” or “T1s”) in a compact way.

3. The number of non-zero coefficients in neighboring blocks is correlated. The number of coefficients is encoded using a look-up; the choice of look-up table depends on the number of non-zero coefficients in neighboring blocks.

4. The level (magnitude) of non-zero coefficients tends to be higher at the start of the reordered array (near the DC coefficient) and lower towards the higher frequencies.

CAVLC takes advantage of this by adapting the choice of VLC look-up table for the "level" parameter depending on recently-coded level magnitudes.

CAVLC encoding of a block of transform coefficients proceeds as follows.

5.1.2.1 PARSING PROCESS FOR TOTAL NUMBER OF TRANSFORM COEFFICIENT LEVELS AND TRAILING ONES

Inputs to this process are bits from slice data, a maximum number of non-zero transform coefficient levels maxNumCoeff, the luma block index luma4x4BlkIdx or the chroma block index chroma4x4BlkIdx of the current block of transform.

Outputs of this process are TotalCoeff(coeff_token) and TrailingOnes(coeff_token).

The syntax element coeff_token is decoded using one of the five VLCs specified in five right-most columns of Table A (See Appendix). Each VLC specifies both TotalCoeff(coeff_token) and TrailingOnes(coeff_token) for a given codeword coeff_token. The choice of table depends on the number of non-zero coefficients in upper and left-hand previously coded blocks N_U and N_L. A parameter N is calculated as follows:

If blocks U and L are available (i.e. in the same coded slice), $nC = (N_U + N_L)/2$. If only block U is available, $nC = N_U$; if only block L is available, $nC = N_L$; if neither is available, $nC = 0$.

nC selects the look-up table and in this way the choice of VLC adapts depending on the number of coded coefficients in neighbouring blocks (**context adaptive**). Num-VLC0 is "biased" towards small numbers of coefficients; low values of TotalCoeffs (0 and 1) are assigned particularly short codes and high values of TotalCoeff particularly long codes. Num-VLC1 is biased towards medium numbers of coefficients (TotalCoeff values around 2 – 4 are assigned relatively short codes), Num-VLC2 is biased towards higher numbers of coefficients and FLC assigns a fixed 6-bit code to every value of TotalCoeff.

5.1.2.2 PARSING PROCESS FOR LEVEL INFORMATION

Inputs to this process are bits from slice data, the number of non-zero transform coefficient levels TotalCoeff(coeff_token), and the number of trailing one transform coefficient levels TrailingOnes(coeff_token).

Output of this process is a list with name level containing transform coefficient levels.

Initially an index i is set equal to 0. Then the following procedure is iteratively applied TrailingOnes(coeff_token) times to decode the trailing one transform coefficient levels (if any):

- A 1-bit syntax element trailing_ones_sign_flag is decoded and evaluated as follows.
- If trailing_ones_sign_flag is equal to 0, the value +1 is assigned to level[i].
- Otherwise (trailing_ones_sign_flag is equal to 1), the value -1 is assigned to level[i].
- The index i is incremented by 1.

Following the decoding of the trailing one transform coefficient levels, a variable suffixLength is initialised as follows.

- If TotalCoeff(coeff_token) is larger than 10 and TrailingOnes(coeff_token) is less than 3, suffixLength is set equal to 1.
- Otherwise (TotalCoeff(coeff_token) is less than or equal to 10 or TrailingOnes(coeff_token) is equal to 3), suffixLength is set equal to 0.

The following procedure is then applied iteratively (TotalCoeff(coeff_token) – TrailingOnes(coeff_token)) times to decode the remaining levels (if any):

- The syntax element level_prefix is decoded using the VLC specified in Tables (See Appendix).
- The variable levelSuffixSize is set equal to the variable suffixLength with the exception of the following two cases.
- When level_prefix is equal to 14 and suffixLength is equal to 0, levelSuffixSize is set equal to 4.
- When level_prefix is equal to 15, levelSuffixSize is set equal to 12.

The syntax element level_suffix is decoded as follows.

- If levelSuffixSize is greater than 0, the syntax element level_suffix is decoded as unsigned integer representation u(v) with levelSuffixSize bits.
- Otherwise (levelSuffixSize is equal to 0), the syntax element level_suffix shall be inferred to be equal to 0.
- A variable levelCode is set equal to (level_prefix << suffixLength) + level_suffix.

- When level_prefix is equal to 15 and suffixLength is equal to 0, levelCode is incremented by 15.
- When the index i is equal to TrailingOnes(coeff_token) and TrailingOnes(coeff_token) is smaller than 3, levelCode is incremented by 2.

The variable level[i] is derived as follows.

- If levelCode is an even number, the value (levelCode + 2) >> 1 is assigned to level[i].

- Otherwise, the value (-levelCode – 1) >> 1 is assigned to level[i].
- When suffixLength is equal to 0, suffixLength is set equal to 1.
- When the absolute value of level[i] is larger than (3 << (suffixLength – 1)) and suffixLength is less than 6, suffixLength is incremented by 1.
- The index i is incremented by 1.

5.1.2.3 PARSING PROCESS FOR RUN INFORMATION

Inputs to this process are bits from slice data, the number of non-zero transform coefficient levels TotalCoeff(coeff_token), and the maximum number of non-zero transform coefficient levels maxNumCoeff.

Output of this process is a list of runs of zero transform coefficient levels preceding non-zero transform coefficient levels called run.Initially, an index i is set equal to 0.

The variable zerosLeft is derived as follows.

- If the number of non-zero transform coefficient levels TotalCoeff(coeff_token) is equal to the maximum number of non-zero transform coefficient levels maxNumCoeff, a variable zerosLeft is set equal to 0.
- Otherwise (the number of non-zero transform coefficient levels TotalCoeff(coeff_token) is less than the maximum number of non-zero transform coefficient levels maxNumCoeff), total_zeros is decoded and zerosLeft is set equal to its value.

The VLC used to decode total_zeros is derived as follows:

- If maxNumCoeff is equal to 4 one of the VLCs specified in Table D (See Appendix) is used.
- Otherwise (maxNumCoeff is not equal to 4), VLCs from Table 9-7 and Table 9-8 (See Appendix) are used.

The following procedure is then applied iteratively (TotalCoeff(coeff_token) – 1) times:

The variable run[i] is derived as follows.

- If zerosLeft is larger than zero, a value run_before is decoded based on Table 9-10 (See Appendix) and zerosLeft. run[i] is set equal to run_before.
- Otherwise (zerosLeft is equal to 0), run[i] is set equal to 0.
- The value of run[i] is subtracted from zerosLeft and the result assigned to zerosLeft. The result of the subtraction shall be larger than or equal to 0.

- The index i is incremented by 1.

Finally the value of zerosLeft is assigned to run[i].

5.1.3 IMPLEMENTATION OF VLD IN REFERENCE DECODER

We shall now discuss the implementation of VLD and its sub modules in the reference decoder. The basic algorithm used by the decoder in the implementation of all the implementations is the same.

5.1.3.1 CALCULATION FOR TOTAL NUMBER OF TRANSFORM COEFFICIENT LEVELS AND TRAILING ONES

The process takes bits from slice data and the presence or absence of neighbouring blocks in the form of variable **‘vlcnum’** as input. The vlcnum value corresponds to the table to be used for decoding. The code book given in Table A (See Appendix) is hard coded in the form of two tables. One holds the length of each code and the other holds the numerical value of each code. These values are such placed in the two dimensional array that the column of the element in the array gives the value of Total Coefficients and the row gives the value of Trailing Ones. The flow chart of the implementation is given below.

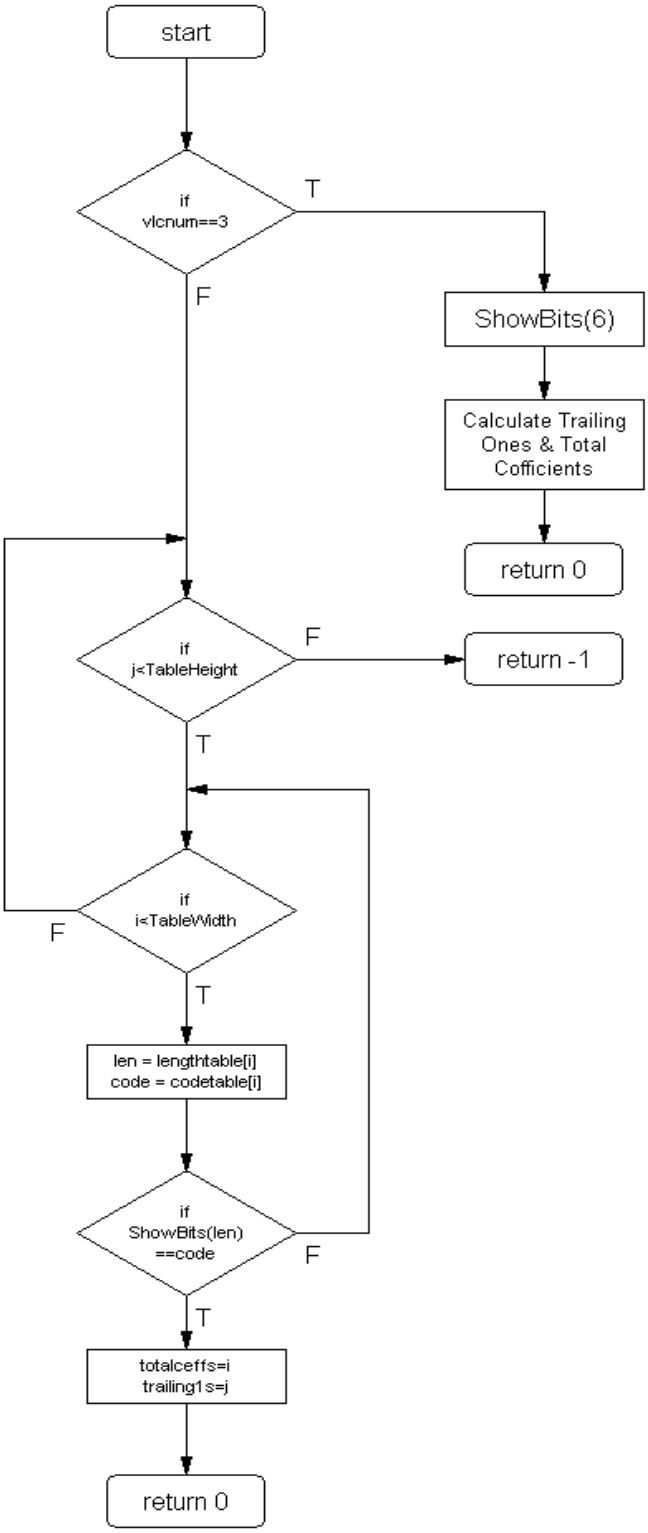


Figure 5.1 FlowChart for the calculation of Trailing Ones and Total Coefficients in Reference Decoder.

The case vlcnum = 3, is taken as a special case as it is the case of fixed length code. It reads six bits from the slice data. The values of Total Coeffs and Trailing ones are calculated as follows:

```
Trailing Ones = code & 3;
Total Coeffs = (code >> 2);
        if (!Trailing Ones && Total Coeffs == 3)
                Total Coeffs = 0;
        else
                Trailing Ones++;
```

If vlcnum is not equal to 3, i.e. it is either 0 or1 or 2, then it iteratively searches for a match between the values of the table and the bits read from the stream. Once a match is found it returns 0 and exits from the loops. If it parses the entire table and no match is found it returns -1 indicating error.

5.1.3.2 CALCULATION OF TOTAL ZERO COEFFICIENTS

The process takes bits from slice data and the value of Total Coeffs in the form of variable **'vlcnum'** as input. The algorithm and method of implementation is the same to that of the Trailing Ones and Total Coeffs calculations. The only difference is that the tables used for the calculation of Total Zeros are different from the tables of Trailing Ones and Total Coeffs. Total Zeros are calculated using Tables B and C (See Appendix). These tables are hard coded in a similar fashion as the Trailing Ones table. The flow chart or the remaining algorithm is exactly the same as for Total Coeffs and Trailing Ones. The difference is that only one value is extracted i.e. of Total Zeros instead of two values. And vlcnum = 3, is not treated as a special case and the same procedure is followed as for the other cases.

5.1.3.3. CALCULATION OF RUN INFORMATION

The run information process takes the same inputs as the Total Zeros calculation process. The output is the list of run of zero coefficient levels preceding the non zero

coefficient levels. The table used in the process is Table 9.10 (See Appendix). This table is hard coded and used in the same algorithm used by the previous two processes. The process is overall completely similar to the previous process.

5.1.4 IMPLEMENTATION OF VLD FOR TM – 1300

5.1.4.1 OPTIMIZATION OF TRAILING ONES AND TOTAL COEFFS FOR THE CAVLC

5.1.4.1.1 TABLE GENERATION

The process used in determining the values of Trailing Ones and Total Coeffs has substantial loads due to repetitive checking. An alternate approach was used in which all the tables were self generated instead of hard coding them. The tables were generated on the basis of two properties of the code words.

1. Leading Zeros of each code word
2. The three binary values after the '1' following the leading zeros

The first Three Tables were generated on the basis of the above two mentioned parameters such that the leading zeros corresponded to the table row and the value after the '1' corresponded to the column of the table.

The value of Total Coeffs and Trailing Ones corresponding to each table element is determined by running the original reference decoder code which checks each element value and length with the hard coded table value and length.

Once the match is found four types of information is packed into the int type table element each occupying 8 bits of the 32-bit integer.

1. The value of Total Coeffs is stored in the first 8 bits.
2. The value of Trailing Ones is stored in the next 8 bits i.e. from bit 8 to 15.
3. The length of the code word is stored from bits 16 to 23.
4. The last 8 bits are used as a check and hold zero if the code exists. If the code doesnot exist it holds FF. This check is necessary since many of the combinations generated in the table donot exist i.e. are defined as legitimate code words in the code book given in the standard.

Error Byte	Length of Code Word	Trailing Ones	Total Coeffs
31 23		15 7	0

Table 5.1 Information in each Table Element

The load of table generation is only once in the whole execution of the code. These code tables are generated at the very start of the execution of the decoder. So it has no effect on the decoding speed of the decoder. It is just considered as an initialization step. So it basically is considered as a no load operation.

5.1.4.1.2 IMPLEMENTATION

The decoding of the trailing ones is carried out by calling the same function as was in the reference code. If vlcnum==3 i.e. nC>=8 then the values are calculated in a similar fashion as was in the reference decoder. Same also applies for the chroma case.

In case vlcnum==0, 1 or 2, then the function:

readSyntaxElement_TrailingOnes(sym,dP,temp_code);

is called with varying argument of temp_code, which corresponds to its respective generated table.

16 bits are read from the stream by ShowBits.

5.1.4.1.3 CALCULATION OF LEADING ZEROS

The number of Leading Zeros in the 16 bits read and the value after '1' has to be found in order to determine the position of the table element needed to be read.

IEEE 754 Standard deals with the Floating Point numbers representation in computers. The method of storage of floating point numbers defined in this standard can be greatly used to our advantage to determine the Leading Zeros and the value of numbers after '1' terming them as mantissa.

5.1.4.1.4 IEEE 754 FLOATING POINT STANDARD

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the fraction and an implicit leading digit (explained below). The exponent base is implicit and need not be stored.

The following figure shows the layout for a single (32-bit) and double (64-bit) precision floating-point values. The number of bits for each field are shown (bit ranges are in square brackets):

	Sign	Exponent	Fraction	Bias
Single Precision	1 [31]	8 [30-23]	23 [22-00]	127
Double Precision	1 [63]	11 [62-52]	52 [51-00]	1023

Table 5.2 Layout For A Single And Double Precision Floating-Point Values

- **The Sign Bit**

The sign bit is as simple as it gets. 0 denotes a positive number; 1 denotes a negative number. Flipping the value of this bit flips the sign of the number.

- **The Exponent**

The exponent field needs to represent both positive and negative exponents. To do this, a '**bias'** is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of (200-127), or 73.

- **The Mantissa**

The mantissa, also known as the '**significand'**, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits.

To find out the value of the implicit leading bit, consider that any number can be expressed in scientific notation in many different ways. For example, the number five can be represented as any of these:

5.00×10^{0}

0.05×10^{2}

5000×10^{-3}

In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in **normalized** form. This basically puts the radix point after the first non-zero digit. In normalized form, five is represented as 5.0×10^{0}. A nice little

optimization is available to us in base two, since the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits.

So, to sum up:

1. The sign bit is 0 for positive, 1 for negative.
2. The exponent's base is two.
3. The exponent field contains 127 plus the true exponent for single-precision.
4. The first bit of the mantissa is typically assumed to be 1.*f*, where *f* is the field of fraction bits.

Now since the value of exponential gives the number of digits following the most significant number so if we substract that value from the total length of the float we will get the leading zeros.

So in the code it is calculated as:

```
Out=142-((0x7F800000&(*pival))>>23);
```

The value is subtracted from 142 because of being weighted by 127 and the value passed to the function is aligned to the 16^{th} bit rather than the 32^{nd} bit. It means that the most significant 16 bits do not contain any info.

127 + 15 = 142

The value of mantissa gives the value of the three bits following ‘1’.The value is simply obtained as:

```
cx = (*iptr & 0x007FFFFF)>>20;
```

Now since the Leading zeros and cx have been determined they are used to retrieve data from the table element corresponding to the position equivalent to their values. The process is shown in the flow chart given below:

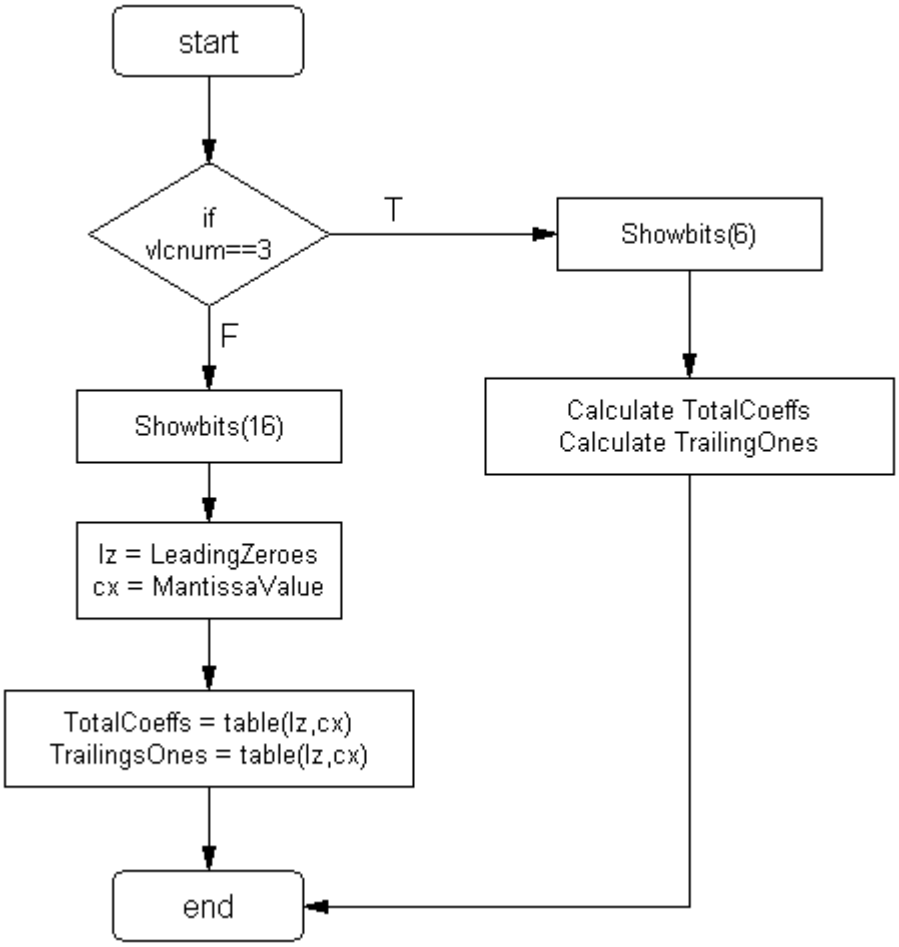


Figure 5.2 Flow Chart of Trailing Ones And Total Coefficients (Own Implementation)

5.1.4.5 CALCULATION OF TOTAL ZERO COEFFICIENTS

The inputs are the bits from the slice data and TotalCoeffs value. The output is the number of Total Zero Coefficients.

The value of Total Coeffs in the standard vary from 1 – 15, however in the code the value varies from 0 – 14. It is important to clarify it here as the value Total Coeffs determines the table to be selected of the code book given in Tables B and C (See Appendix). So in the algorithm, if the value of Total Coeffs is 0, it will correspond to table 1 of Table B. Different procedures are followed for the calculation of Total Zeros for different values of Total Coeffs. Hence decision is made on the value of Total Coeffs. This is clear in the Flow Chart given below:

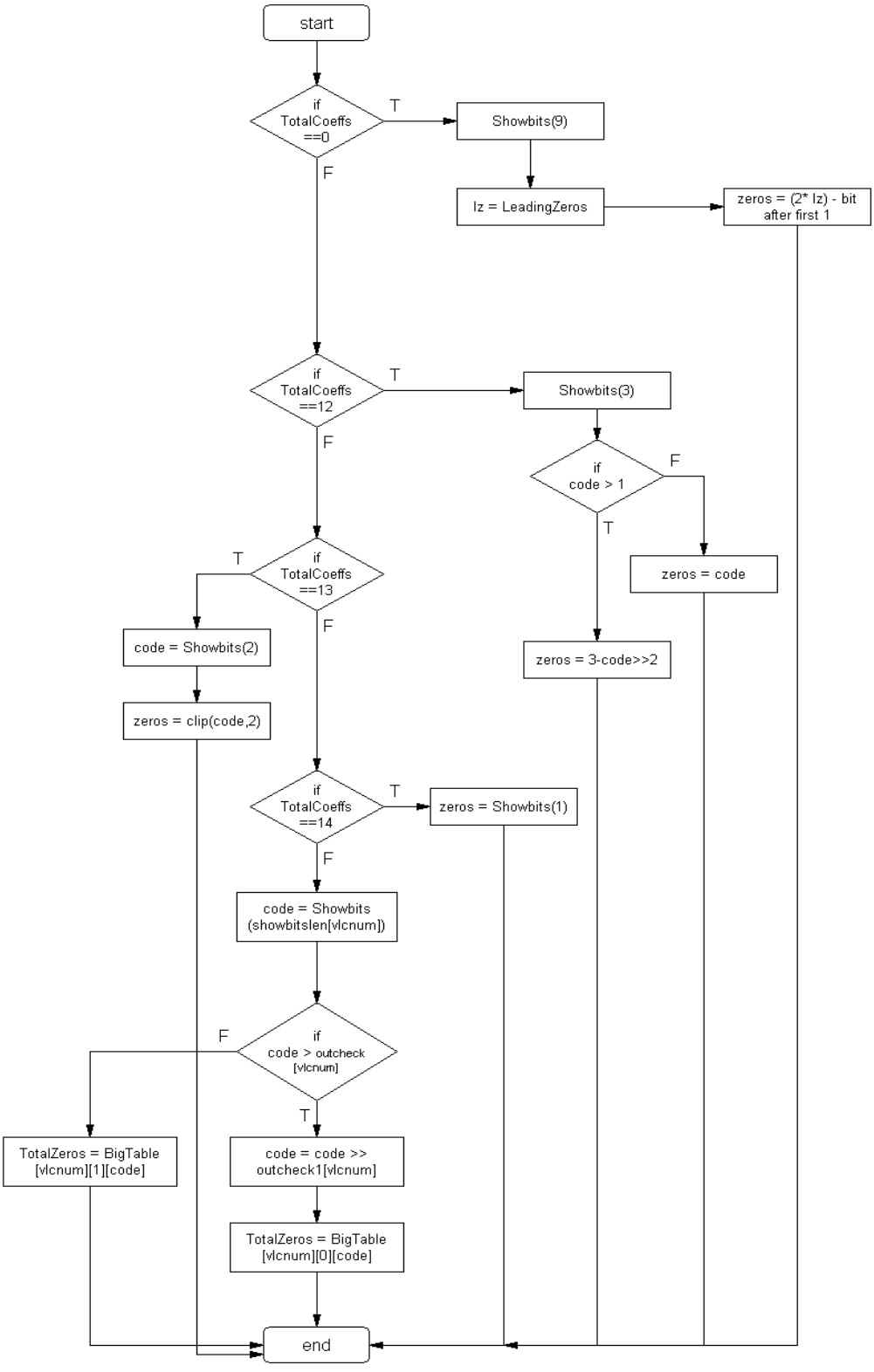


Figure 5.3 Calculations of Total Zeros for Different Values of Total Coefficients

If the value of Total Coeffs is equal to zero, then the decoder reads 9 bits from the stream. It then finds the leading zero bits using the same process used in the calculation of Leading Zeros for Total Coeffs and Trailing Ones.

Now to understand the method of calculating the Total Zeros value lets consider a code as an example.

e.g code = 0000010

Leading zero bits =5 Bit after 1st one = 0

So output is Zeros Left = (2 * leading zero bits) - bit after 1st one

= 2*5 +0 = 10

A special check is placed for the last value of zeros left i.e 15. As it has no bit after 1st one so the value calculated may exceed 15. Hence a clipping macro is called which clips the values greater than 15 to 15.

If the value of Total Coeffs in not equal to 0, then it checks if it is equal to 12. If the condition is satisfied it reads 3 bits from the bitstream. It then checks if the numerical value the three bits read is greater than 1 or not. If the condition is satisfied it calculates the value Total Zeros as follows:

Total Zeros = 3 – (code>>2)

Now, the values given in Table C for this case are: 000, 001, 1xx, 01x

So in the above case we are considering 3rd and 4th values. Shifting them left by 2, gives values 1 and 0 respectively from 3rd and 4th. Subtracting these values from 3 simply gives the value of Zeros left. If the numerical value of the 3 bits is not greater than one, the value of Total Zeros is simply the numerical value of the 3 bits.

If the value of Total Coeffs is neither 0 nor 12, we check if it is equal to 13. If the condition is true we read two bits from the stream. The code read is clipped to the value of ‘2’. The value obtained after clipping is simply the value of Total Zeros.

If the value of Total Coeffs does not satisfy the above if statement we check if it is equal to 14. If the condition is true, only one bit is read from the stream. The bit read is the value of Total Zeros.

If the Total Coeffs fails all the above conditions, then we deal all the other cases on the basis of the same algorithm.

An array showbitlen is passed through the function which contains the maximum length of a code in each table. The max length value of each table is placed in the array at the same position as corresponding to the table no. e.g For table No 5 of the standard the length can be accessed as showbitlen[4]. Bits equal to the value given in the array for each table are read from the stream.

Now the table for each given input is mathematically divided into two, i.e. if the code value read is bigger than a certain value it is calculated in one way and if not it is calculated in another way.

The bench mark values which decide whether it is to be calculated by 1^{st} or 2^{nd} are also defined in an array in a similar way as the max length for each table.

These values are defined in the array **outcheck[ ]**.

Now

If (code > outcheck [Total Coeffs]

out = code >> outcheck1[Total Coeffs]

outcheck1 is also an array very much similar to **outcheck** which defines the amount each value of code for a particular table is to right shifted.

Then the value of the Total Zeros is obtained as follows:

Total Zeros = Big table [Total Coeffs –1] [0] [out]

Big table is a self defined table of dimensions 11 x 2 x 16.

11 = number of tables

2 = upper or lower part of the table. 0 for code>outcheck[] and 1 for less.

16 = code can have values from 0 to 15.

The value at a particular point in the array ‘Big table’ specifies the value of Total Zeros for that value of code. e.g.

In Table 4 (Total Coeffs = 0 to 14) code =3 (011xx) corresponds to Total Zeros of 7.

Now as it is right shifted by 1 so code = 011x i.e 0110 or 0111 (6 or 7)

So Big table[4][0][6] =7 or Big table[4][0][7] =7

5.1.4.6 CALCULATION OF RUN INFORMATION

The process takes value of Total Zeros from the previous decoding step as input along with the slice data bitstream. The output of this process is a list of run of zero transform coefficient levels preceding non zero transform coefficient levels called run.

The tables corresponding to each of the Total Zeros values are given in Table D (See Appendix). The Total Zeros value obtained as input is incremented by 1 as it can be seen from the table that values of Total Zeros start from 1.

The implementation of the calculation of the Run information is shown in the Flow chart below.

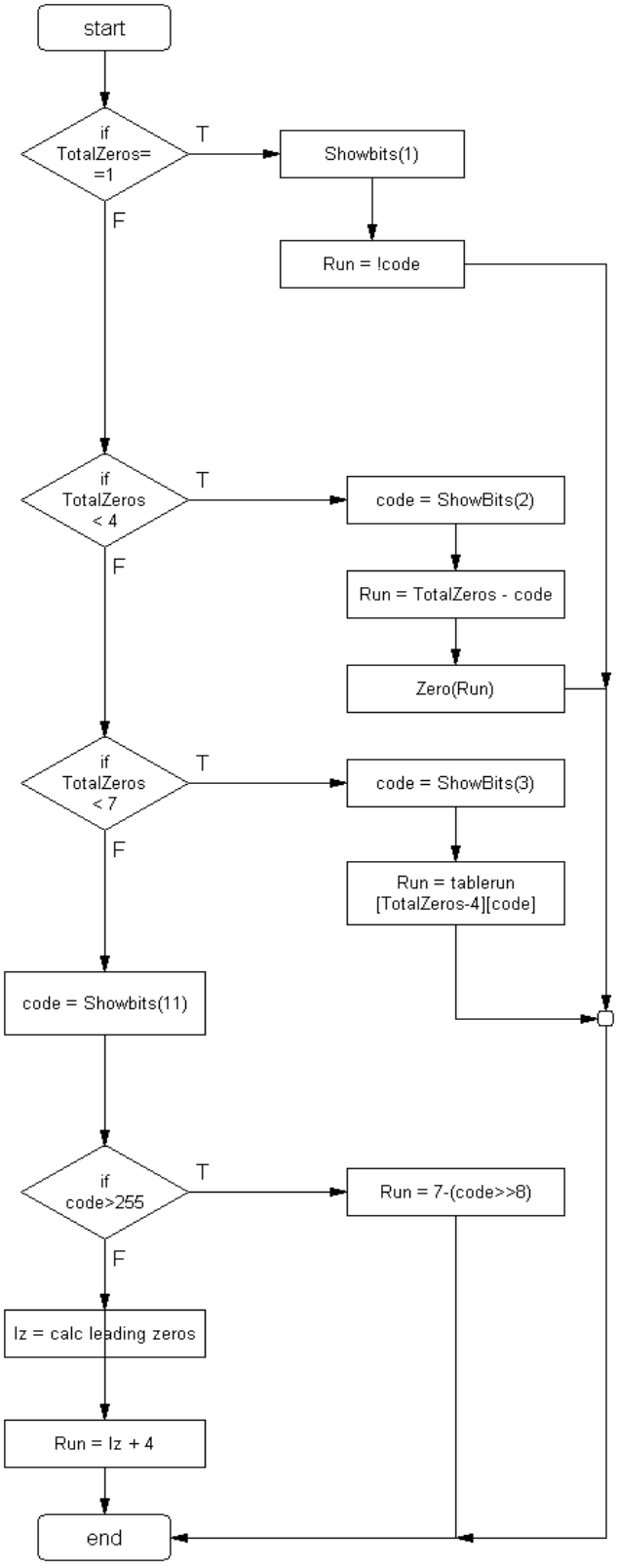


Figure 5.4 FlowChart of Own Implementation of Run Information

The Total Zeros value obtained from the previous step determines the table to be used. First we check if Total Zeros value is equal to 1. If the condition is true then it reads 1 bit from the bitstream. The complement of the bit read gives the run information.

If the above if statement is false, then it is checked if the value of Total Zeros is less than 4. If it is less than 4, the decoder reads 2 bits from the stream. The value of the run is obtained as:

Run = Total Zeros – the 2 bits read

If both the above conditions fail, it checks if the value of Total Zeros is less than 7, if the condition is satisfied 3 bits are read from the stream.

Two tables namely Table Run and Table RunLen of sizes 3x8 are predefined. The first one contains the values of **'run'** while the 2nd one contains the length of the code corresponding to that run information.

The value of Run is obtained as follows:

Run Before = Table Run [Total Zeros – 4] [3 bits read]

If all the above conditions fail, then we read 11 bits from the stream. If the value of the 11 bits read is greater than 255, the value of run is calculated as:

Run = 7 – (11 bits Read >> 8)

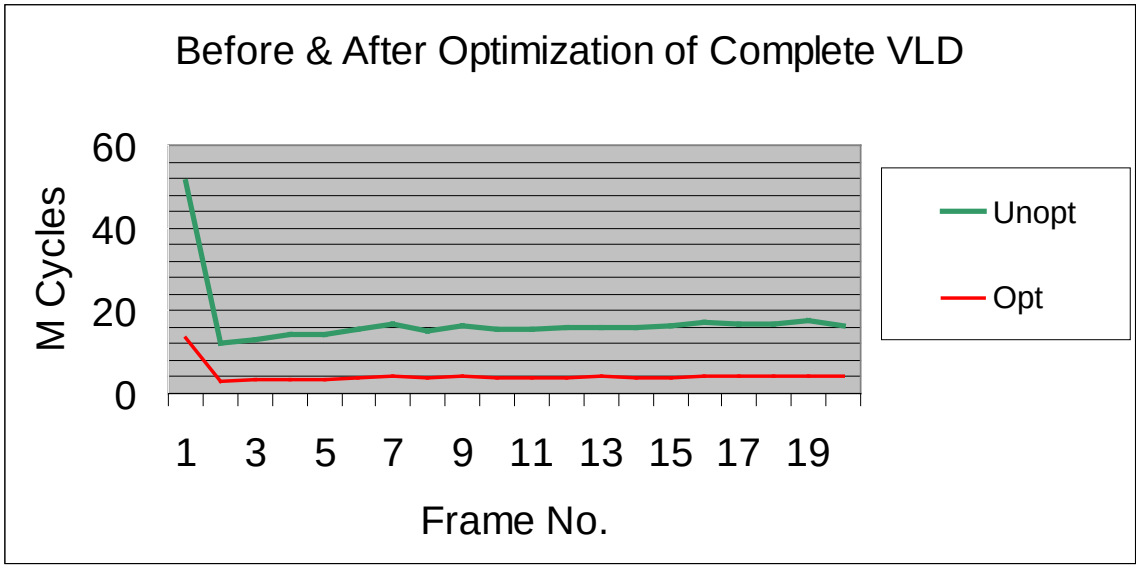
If the value of 11 bits read is less than 255, leading zero bits are found for the eleven bits read. The Run information is found as:

Run Before = leading Zero Bits +4

5.1.5 RESULTS OF OPTIMIZATION

The original algorithm used nested for loops to find the values from the code book in each process. The nested for loops resulted in the breaking of the decision tree and results in enormous loads. Our approach of implementation focused on the decrease in the

breaking of decision trees and fast approaches to reach to the results. This resulted in substantial decrease in loads on the processor. Originally the process of VLD was taking 15 Mega Cycles. After the optimization of algorithms the load was reduced to 3 Mega Cycles resulting in an enormous decrease of 500 %.



5.2 OPTIMIZATION OF MOTION COMPENSATION

5.2.1 MOTION COMPENSATION OF LUMINANCE IN H.264

H.264 supports motion compensation block sizes ranging from 16x16 to 4x4 luminance samples with many options between the two. The luminance component of each macroblock (16x16 samples) may be split up in 4 ways as shown in Figure 5.5: 16x16, 16x8, 8x16 or 8x8. Each of the sub-divided regions is a macroblock partition. If the 8x8 mode is chosen, each of the four 8x8 macroblock partitions within the macroblock may be split in a further 4 ways as shown in Figure 5.5: 8x8, 8x4, 4x8 or 4x4 (known as macroblock sub-partitions).

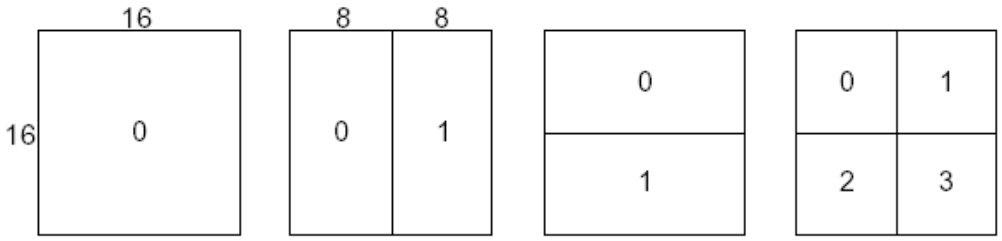


Figure 5.5(a) Macroblock partitions 16x16, 8x16, 16x8, 8x8

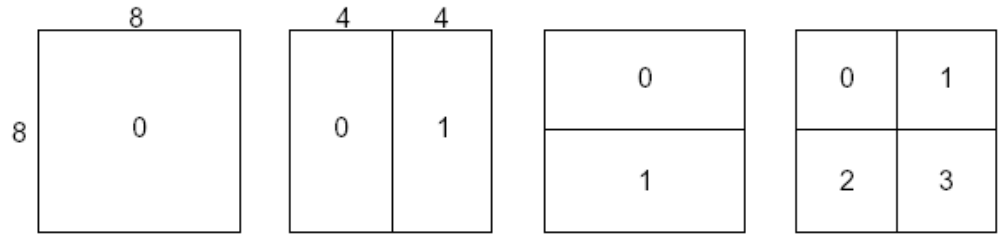


Figure 5.5(b) Macroblock sub-partitions 8x8, 4x8, 8x4, 4x4

Each partition in an inter-coded macroblock is predicted from an area of the same size in a reference picture. The offset between the two areas (the motion vector) has ¼-pixel resolution. The luma samples at sub-pixel positions do not exist in the reference picture and so it is necessary to create them using interpolation from nearby image samples. The sub-pixel samples at half-pixel positions are generated first and are interpolated from neighbouring integer-pixel samples using a 6-tap Finite Impulse Response filter. This means that each half-pixel sample is a weighted sum of 6 neighbouring integer samples. Once all the half-pixel samples are available, each quarter-pixel sample is produced using bilinear interpolation between neighbouring half- or integer-pixel samples.

5.2.1.1 LUMA SAMPLE INTERPOLATION PROCESS

Interpolation takes following parameters as input.

- A luma location in full-sample units (xIntL, yIntL),
- A luma location offset in fractional-sample units (xFracL, yFracL),and
- The luma sample array of the selected reference picture refPicLXL

Output of this process is a predicted luma sample value predPartLXL[xL, yL].

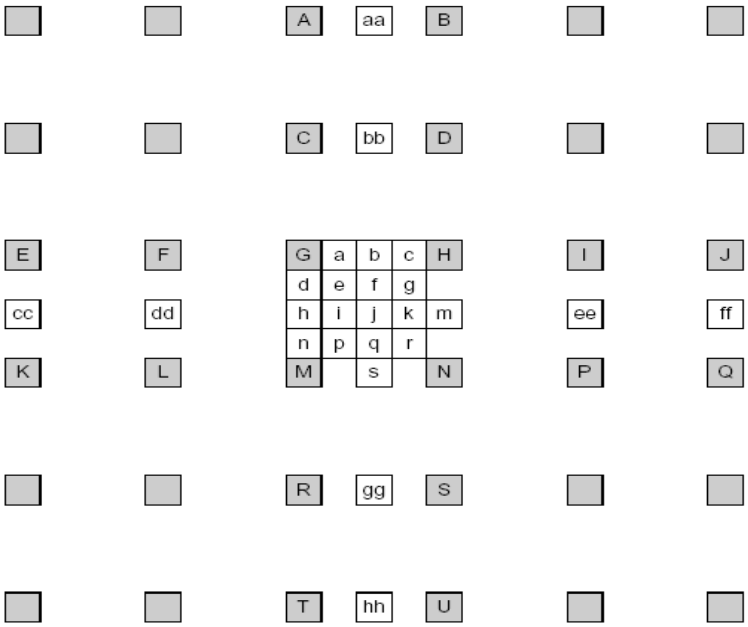


Figure 5.6 Integer samples (shaded blocks with upper case letters) and fractional sample positions (unshaded blocks with lower case letters) for quarter sample luma interpolation

In Figure 5.6, the positions labelled with upper-case letters within shaded blocks represent luma samples at full-sample locations inside the given two-dimensional array refPicLXL of luma samples. These samples may be used for generating the predicted luma sample value predPartLXL[xL, yL]. The locations (xZL, yZL) for each of the corresponding luma samples Z, where Z may be A, B, C, D, E, F, G, H, I, J, K, L, M, N, P, Q, R, S, T, or U, inside the given array refPicLXL of luma samples are derived as follows:

xZL = Clip3(0, PicWidthInSamples – 1, xIntL + xDZL)

yZL = Clip3(0, PicHeightInSamples – 1, yIntL + yDZL)

Table 8-10 specifies (xDZL, yDZL) for different replacements of Z.

Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	P	Q	R	S	T	U
xDZ_L	0	1	0	1	-2	-1	0	1	2	3	-2	-1	0	1	2	3	0	1	0	1
yDZ_L	-2	-2	-1	-1	0	0	0	0	0	0	1	1	1	1	1	1	2	2	3	3

Table 5.3 Differential full-sample luma locations

Given the luma samples ‘A’ to ‘U’ at full-sample locations (xAL, yAL) to (xUL, yUL), the luma samples ‘a’ to ‘s’ at fractional sample positions are derived by the following rules. The luma prediction values at half sample positions are derived by applying a 6-tap filter with tap values (**1, -5, 20, 20, -5, 1**). The luma prediction values at quarter sample positions are derived by averaging samples at full and half sample positions. The process for each fractional position is described below.

- The samples at half sample positions labelled **b** is derived by first calculating intermediate values denoted as **b1** by applying the 6-tap filter to the nearest integer position samples in the horizontal direction. The samples at half sample positions labelled **h** is derived by first calculating intermediate values denoted as **h1** by applying the 6- tap filter to the nearest integer position samples in the vertical direction:

b1 = (E – 5 * F + 20 * G + 20 * H – 5 * I + J)

h1 = (A – 5 * C + 20 * G + 20 * M – 5 * R + T)

The final prediction values b and h are derived using:

b = Clip1((b1 + 16) >> 5)

h = Clip1((h1 + 16) >> 5)

- The samples at half sample position labelled as **j** is derived by first calculating intermediate value denoted as **j1** by applying the 6-tap filter to the intermediate values of the closest half sample positions in either the horizontal or vertical direction because these yield an equal result.

j1 = cc – 5 * dd + 20 * h1 + 20 * m1 – 5 * ee + ff, or

j1 = aa – 5 * bb + 20 * b1 + 20 * s1 – 5 * gg + hh

where intermediate values denoted as **aa**, **bb**, **gg**, **s1** and **hh** are derived by applying the 6-tap filter horizontally in the same manner as the derivation of **b1** and intermediate values denoted as **cc**, **dd**, **ee**, **m1** and **ff** are derived by applying the 6-tap filter vertically in the same manner as the derivation of **h1**. The final prediction value **j** is derived using:

j = Clip1((j1 + 512) >> 10)

- The final prediction values **s** and **m** are derived from **s1** and **m1** in the same manner as the derivation of **b** and **h**, as given by:

s = Clip1((s1 + 16) >> 5)

m = Clip1((m1 + 16) >> 5)

- The samples at quarter sample positions labelled as **a**, **c**, **d**, **n**, **f**, **i**, **k**, and **q** are derived by averaging with upward rounding of the two nearest samples at integer and half sample positions using:

a = (G + b + 1) >> 1

c = (H + b + 1) >> 1

d = (G + h + 1) >> 1

n = (M + h + 1) >> 1

f = (b + j + 1) >> 1

i = (h + j + 1) >> 1

k = (j + m + 1) >> 1

q = (j + s + 1) >> 1.

- The samples at quarter sample positions labelled as **e**, **g**, **p**, and **r** are derived by averaging with upward rounding of the two nearest samples at half sample positions in the diagonal direction using

e = (b + h + 1) >> 1

g = (b + m + 1) >> 1

p = (h + s + 1) >> 1

r = (m + s + 1) >> 1.

The luma location offset in fractional-sample units (xFracL, yFracL) specifies which of the generated luma samples at full-sample and fractional-sample locations is assigned to the predicted luma sample value predPartLXL[xL, yL]. This assignment is done according to Table 8-11. The value of predPartLXL[xL, yL] shall be the output.

$xFrac_L$	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
$yFrac_L$	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
$predPartLX_L[x_L, y_L]$	G	d	h	n	a	e	i	p	b	f	j	q	c	g	k	r

Table 5.4 Assignment of the luma prediction sample

5.2.1.2 IMPLEMENTATION OF LUMA MC IN REFERENCE DECODER

The decoder takes the position of the starting pixel of the block as input. The values are scaled by a factor of 4, so the subpixel positions also from integer values.

The inputs as defined in the standard are calculated as follows:

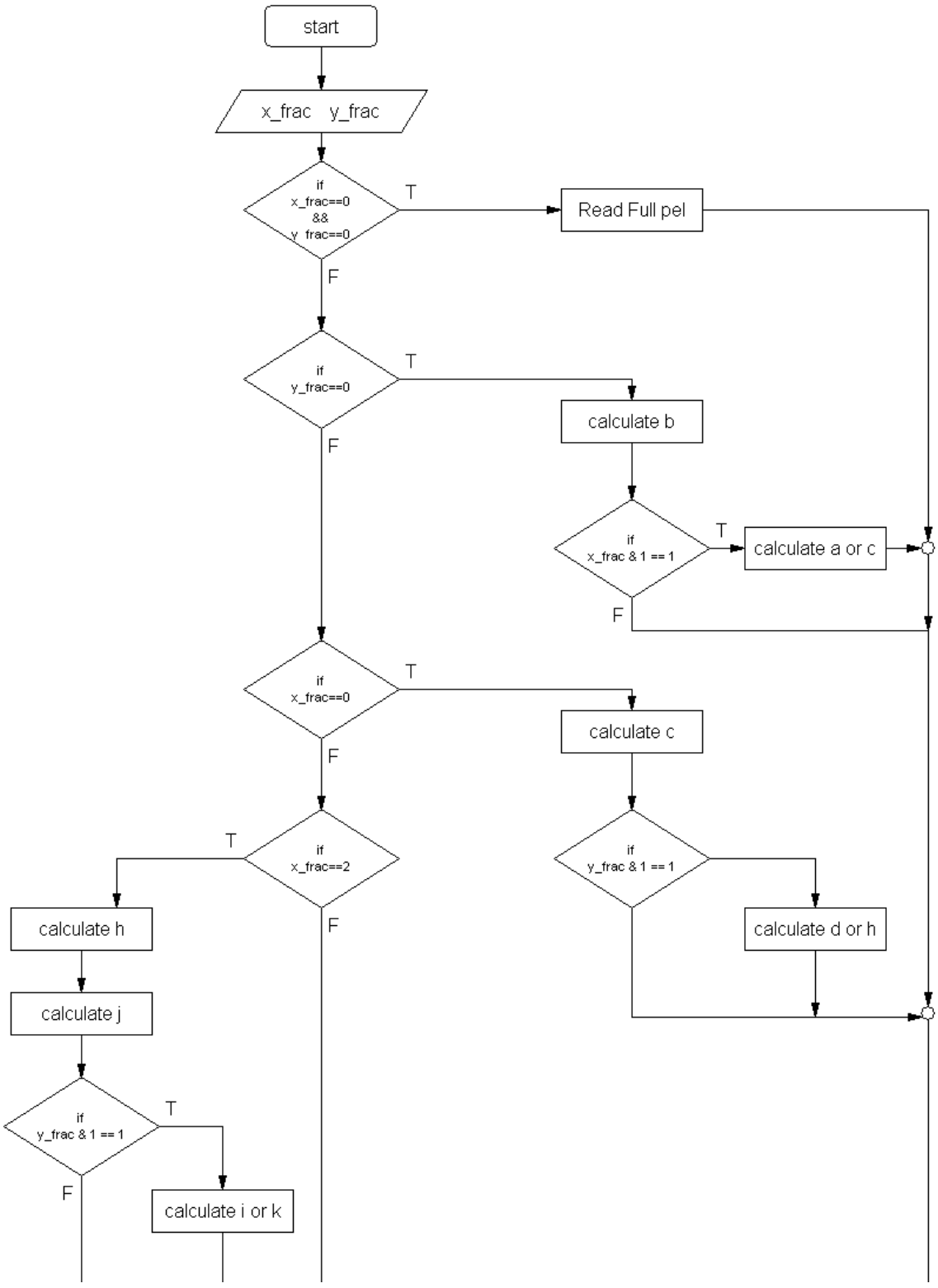
xInt = xL&0x3

yInt = yL&0x3

xFrac = (xL – xInt) / 4

yFrac = (yL – yInt) / 4

The implementation of the reference decoder is shown in the flow chart.


start
x_frac y_frac
if x_frac==0 && y_frac==0
T
Read Full pel
F
if y_frac==0
T
calculate b
F
if x_frac & 1 == 1
T
calculate a or c
F
if x_frac==0
T
calculate c
F
if x_frac==2
T
F
if y_frac & 1 == 1
calculate d or h
calculate h
calculate j
if y_frac & 1 == 1
T
F
calculate i or k

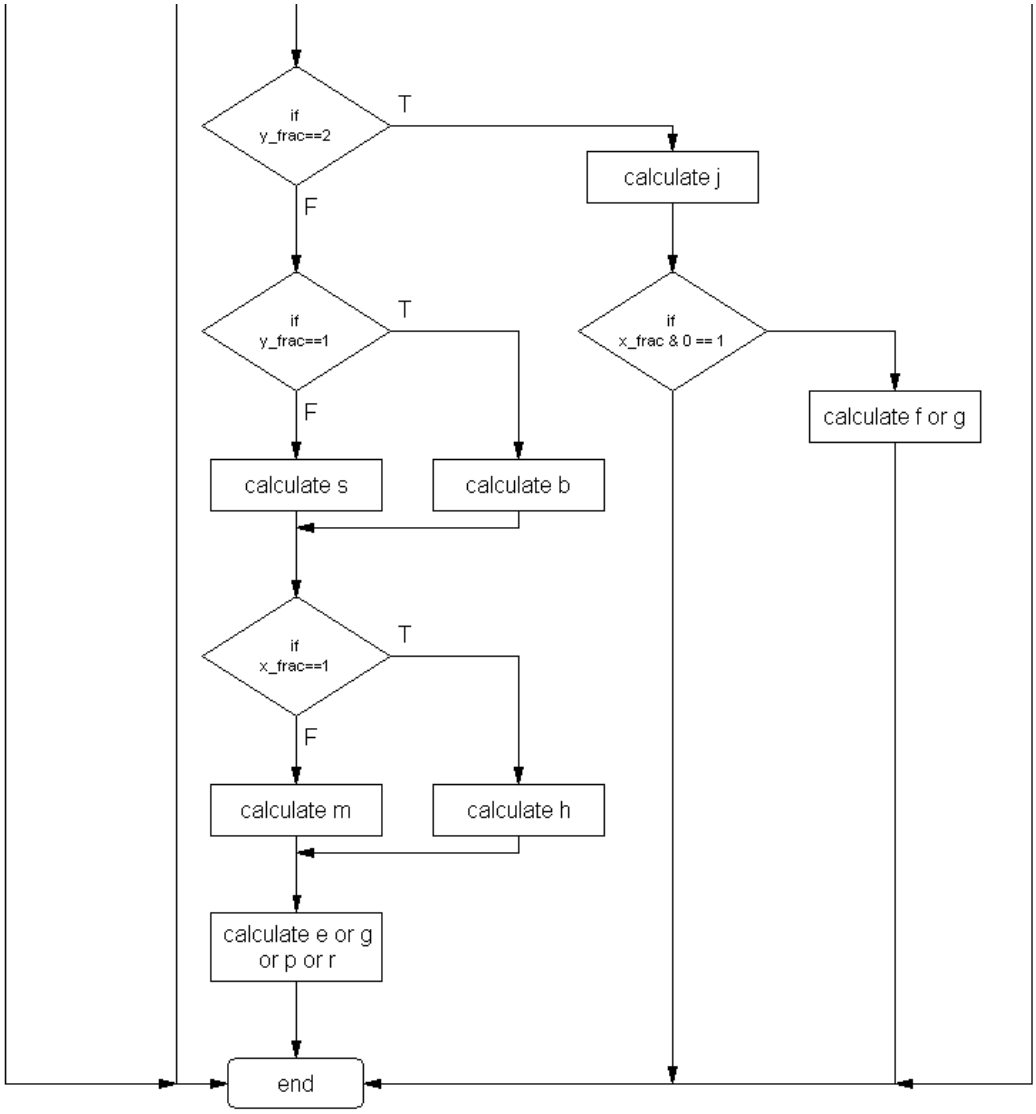


Figure 5.7 Flow Chart of Reference Decoder Motion Compensation

The decoder first checks for the full pel case. If the condition is satisfied then it just reads the required block from the buffer and exists. If it is false then it checks the condition for half sample case i.e. (**b** sample). If it is true it calculates the values of the entire block by applying the filter to find the value of each pixel. Then it goes ahead to check if it is just the **b** case or do **a** or **c** also need to be calculated. After that it exists from the loop. If the **b** condition was false it checks for **h** and if its true for **d** or **n**. If **h** is also false it checks for **j** and further its interdependent cases. Finally if all the above cases are false it calculates the respective blocks needed for **e**, **g**, **p** and **r** and calculates their values.

5.2.1.3 FLAWS IN THE ALGORITHM

The above discussed algorithm has some serious flaws in terms of resource and memory utilization.

1. There are a lot **'if'** statements which result in the breaking of decision tree.
2. The memory access is highly inefficient and the same buffer location is accessed repeatedly for different operations.
3. Another flaw which is considered as a part of the calculation of sub luma interpolation positions is that the algorithm checks each pixel if it is inside the frame i.e. the motion vector is pointing outside the frame. Due to which it has to applying clipping to the pixel coordinates every time it reads from the frame buffer.

5.2.2 IMPLEMENTATION OF LUMA MC FOR TM – 1300

5.2.2.1 THE CONCEPT OF PRE COMPUTATION

In all of the cases of subpixel motion compensation except for the full pel case, **b**, **h** or **j** are needed; either as a final result or as an intermediate value for final calculations. Now for the calculation of **b** for a 4x4 block, a 9x4 block i.e. (56 pixels) needs to read. Now if two 4x4 blocks exist side by side as shown in the figure 5.8, so the above algorithm would read two 9x4 blocks i.e. (112 pixels) out which only 4 pixels would be new.

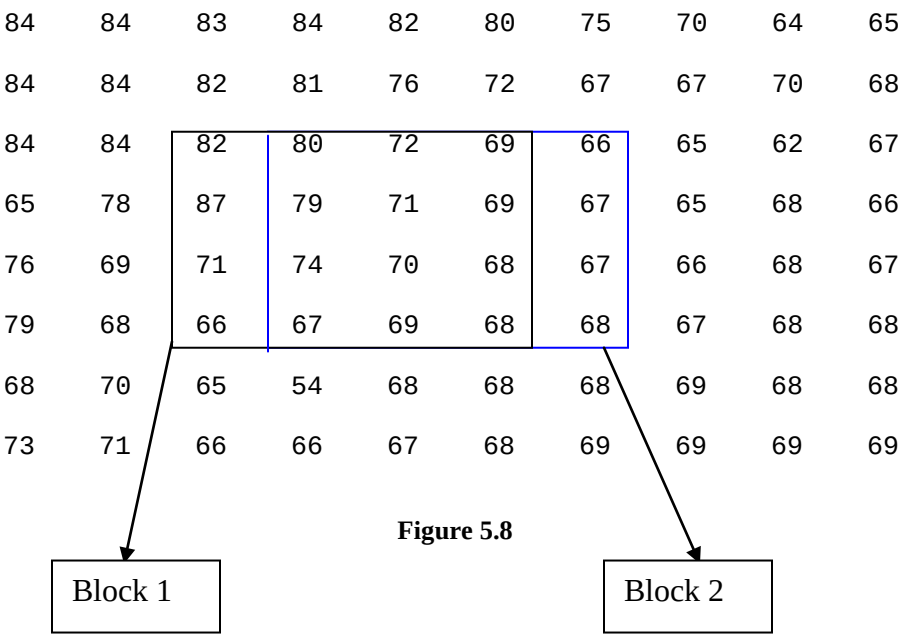


Figure 5.8

This results in very heavy loads on the processor. As each memory operations needs 3 cycles, so 112 pixels would need 112 * 3 = 336 cycles. Out of these 336 cycles 156 cycles are those pixels which are being read again. So basically we are wasting 156 cycles.

5.2.2.2 STATISTICAL OCCURRENCE OF ALL SUBPIXEL CASES

The ratio of the occurrences of various subpixel cases was analyzed over a number of streams, with motion compensation either at 16x16 i.e. (macroblock level) or 4x4 (block level). The results are shown below.

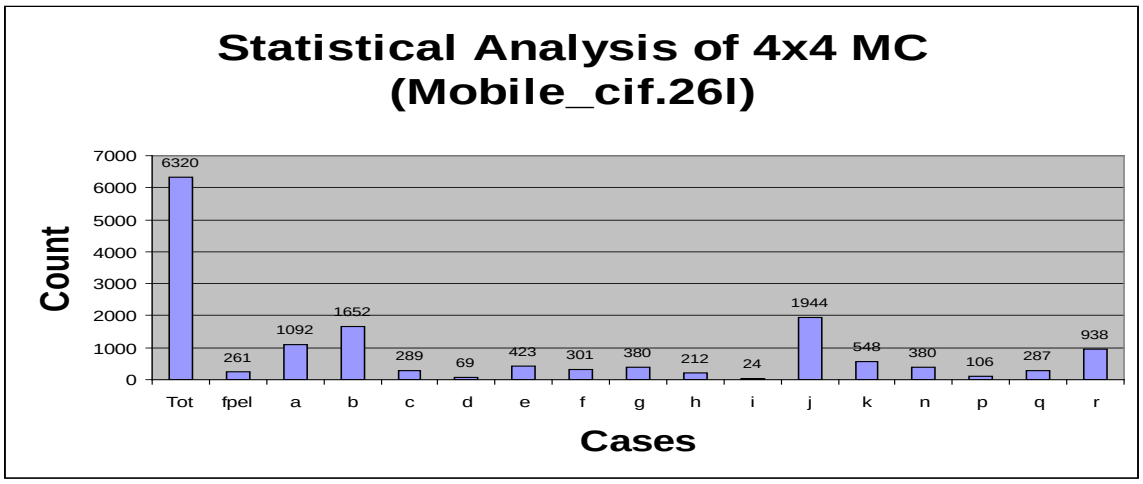


Figure 5.9 Statistical Analysis of 4x4 Motion Compensation

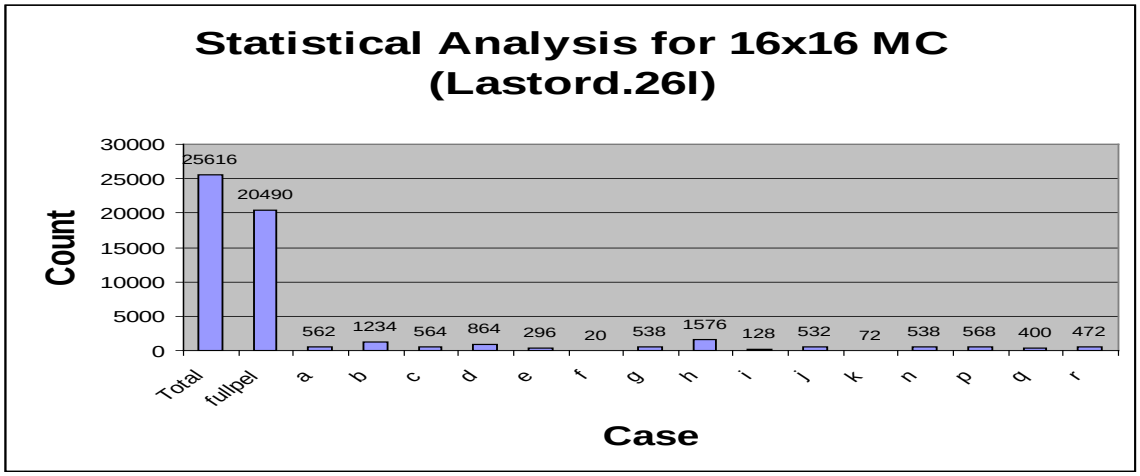


Figure 5.10 Statistical Analysis for 16x16 Motion Compensation

From the graph we see very high ratios of **b**, **h** and **j** in the 4x4 block motion compensation case as these are also used in other cases. The full pel case occurrence is not very often. In the 16x16 macroblock motion compensation the occurrence of full pel case is very frequent but still **b**, **h** and **j** are substantial. Furthermore the graphs also make it clear that the occurrence of intra blocks in inter frames is very rare and most of the blocks would have to predicted from previous frame.

On the basis of the above results we conclude that if pre-compute all the values of **b**, **h** and **j** it can result in the substantial reduction of load. This will avoid re-accessing the same memory locations. It will also decrease the breaking of the decision tree and the algorithm would be greatly simplified.

5.2.2.3 EDGE CREATION AROUND IMAGES

Whenever the motion vector is pointed out of the frame it is basically pointing at a data which does not exist, but would have had a particular value if it would have existed. The value is that which needs to be predicted. In order to avoid erroneous memory access the algorithm has inculcate clipping for memory access operation so that it does not access the wrong memory location. So due to clipping no matter how much the motion vector points out of the picture, the code will always read the memory at the edge of the picture.

Clipping has huge loads as two clipping operations are being executed for each memory access into the picture buffer. This has substantial load on the processor and results in the slowness of the operation.

To avoid the loads of clipping an edge or border in created around the picture in the memory buffer. The width of the edge is set equal to 16 pixels which correspond to the macroblock width. The 16 pixel border is of the same values that of the pixel on the edge of the actual image. This is illustrated in the pictures below.

Figure 5.11(a) Image with No Borders

Figure 5.11(b) Image with 16 Pixel Edges on Each Side

The above picture clearly shows the border created around the image. This simple technique results in a remarkable decrease of load on the processor.

5.2.2.4 IMPLEMENTATION

5.2.2.4.1 PRE-COMPUTE

The first step in the implementation of the above discussed concepts is the creation of the borders of the original image data. This is done by copying the entire image data in another image and borders are created by extending the pixels at the edges.

- **Calculation of 'b'**

Once the border has been created, we calculate **b** for the entire image. The value of **b** is calculated on block to block basis, i.e. one iteration results in 16 **b** value pixels or in other words a 4x4 block. The data is read using an integer pointer. Basically 4 integer pointer hold the 4x4 data.

67	66	68	67	**Int* 1**
79	68	66	67	**Int* 2**
68	70	65	54	**Int* 3**
73	71	66	66	**Int* 4**

Figure 5.12 Data of a 4x4 block in 4 int*

Now the data needed for the calculation of **b** half samples for a 4x4 block is a block of 9x4. As we use integer pointers to read data, we read data equal to 12x4.This data is sent to a macro named **filter**.

The algorithm of the macro **filter** is developed to calculate the **b** half sample values of a single row of a 4x4 block in a single iteration. The filter is based on Trimedia custom op **IFIR8UI (arg1, arg2)**, which byte wise multiply arg1 with arg2 and take there sum.

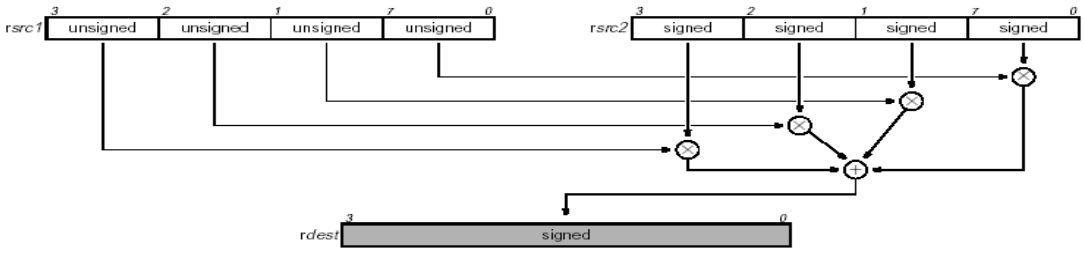


Figure 5.13 Operation of IFIR8UI Custom Op

The filter weights are defined as pre-processor directives and are shifted to satisfy each case as follows:

#define one 1616FB01

#define two 000001FB

#define three 16FB0100

#define four 0001FB16 and so on…..

These values are byte wise multiplied by the image data resulting in the **b** half sample value.

Value = IFIR8UI (one, int1) + IFIR8UI (two, int2)

b = (Value+16) / 32

This process is repeated until all the values of **b** half sample are calculated.

Figure 5.14 Image composed of all 'b' half samples of an image

- **Calculation of 'h'**

For the calculation of **h**, we need to apply the filter vertically on the data. Now data cannot be accessed vertically using integer pointer. This can be a serious drawback to the implementation. To overcome it we take the Transpose of each 4x4 block of the entire image.

84	84	83	84	82	80	75	70
84	84	83	83	81	77	72	71
84	84	82	81	76	72	67	67
84	84	82	80	72	69	66	65
65	78	87	79	71	69	67	65
76	69	71	74	70	68	67	66
79	68	66	67	69	68	68	67
68	70	65	54	68	68	68	69

Figure 5.15 8x8 data outlined box corresponding to a 4x4 Block

84	84	84	84	82	81	76	72
84	84	84	84	80	77	72	69
83	83	82	82	75	72	67	66
84	83	81	80	70	71	67	65
65	76	79	68	71	70	69	68
78	69	68	70	69	68	68	68
87	71	66	65	67	67	68	68
79	74	67	54	65	66	67	69

Figure 5.16 Transposed 8x8 block on 4x4 basis Inset 4x4 block transposed

The transpose of a 4x4 block involve 12 operations. He process is illustrated in the figure below.

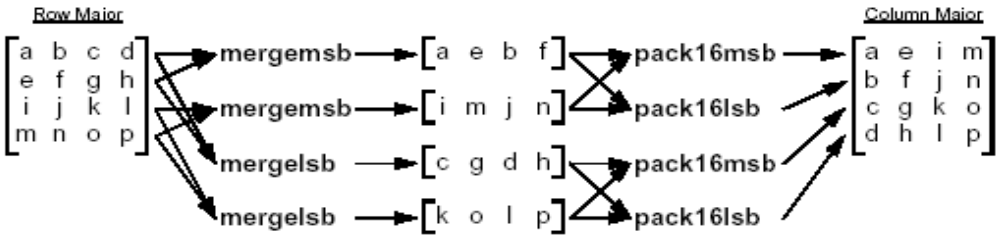


Figure 5.17 Transpose of 4x4 block

Once the data has been transposed, we read data for the first column from 0th row, 4th row and 8th row. The values we get from **filter** are the four values of the first column of the block. The hardware architecture of Trimedia is such designed that accessing the memory locations in rows has less load as compared to jumping to whole stride. Keeping this in view we read the next 4 bytes on the same row. The values returned are the four values of the first column of the next block. Once the end of the row is reached, the pointers start accessing data from the below it, for the case being discussed these would be 1st, 5th and 9th. This will give data of 2nd column of each block. This process is continued until the 1st pointer reaches the end of the 3rd row. When this is done the 1st pointer jumps to the 12th row and so on. This process is continued until we get the full image composed of **h** half samples.

Figure 5.18 Image composed of all 'h' samples values of an image

- **Calculation of j**

The subpixel values of **j**, are calculated in the same as way as the values of **h**. The **j** is calculated by applying filter to the **b** subpixel data. The procedure followed is completely similar to that of the calculation of **h**. The image completely constructed of **j** values is shown in the figure below.

Figure 5.19 Image composed of all 'h' samples values of an image

- **Flaw in the calculation of j**

The implementation of **j** deviates from that of the standard which results in discrepancies in the calculated values. According to the standard, **j** can be calculated from either $\mathbf{b_1}$ or $\mathbf{h_1}$ i.e. the values of **b** or **h** that have not been rounded, averaged and clipped. The values obtained from **filter** which is just the sum of products may be a negative number. As it is clipped so it becomes zero. When **j** is calculated from these values it results in erroneous result. Furthermore, the rounding and averaging also introduces a discrepancy of +-1 in the final result.

The ratio of occurrence of these errors is around **20%**. However, they do not have substantial effect on the quality of image for smaller number of frames. The effects

become visible if all frames of the stream are **P** except for 1st and that too after 150 to 200 frames. In normal streams it is usual practice to send an **I** frame after around 100 to 150 frames. So they effect of this discrepancy becomes negligible.

5.2.2.4.2 GETBLOCK

The implementation of motion compensation becomes very simple after the pre-computation of **b, h** and **j**. All the motion compensation cases are divided int two categories i.e. the ones that do not need further calculation and those who do. Full pel, **b**, **h** and **j** are those cases which do not need further calculation and there data can read directly from there respective buffers.

Now we need to read data from different buffers in different cases. The address of the buffers needed for each case is stored in a number of arrays. For the first case in which no further calculation is involved only one buffer is needed for each case. The address of the different buffers needed in each of the four cases is stored at a position calculated by the formula:

Value = yFrac<<2 + xFrac

So, the values for full pel, **b**, **h** and **j** would correspond to 0, 2, 8 and 10. Accessing the array ptr at these locations would give the starting address of their respective buffers.

5.2.2.4.3 BYTE ALIGNMENT

Once we get the starting address of the buffer, we access data using int*. Now int*'s can only access data from memory location addresses divisible by 4. As the buffers are char buffers, and the motion vector can point at any location so this creates a problem. In order to access data properly we need to make it byte aligned. So we read data from the address before the desired address that is divisible by 4. We also calculate a parameter called **shift** which give the value by which the data has to be shifted to get the desired value.

Shift = xFrac&0x3

Now as we have to access a 4x4 block, we read a block of size 8x4. The desired data is extracted from two int* which hold 8 values of a row. This is done as follows:

ans=((*one<<shift)&(*two>>(4-shift)));

The value we get ans are the required four values of a row of a block.

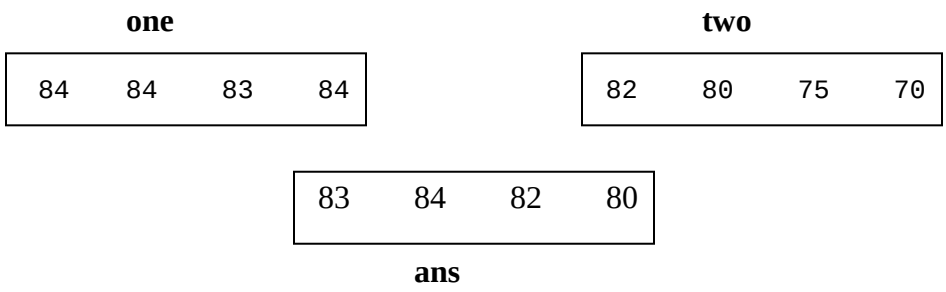


Figure 5.20 Byte Alignment for a case where shift = 2

After byte alignment the data is simply given as output. For the other cases data needs to be read from two buffers. So, two arrays hold the addresses of the required buffers at their respective positions. Once the data has been read using the above procedure, the two 4x4 block values are added and averaged on a pixel by pixel basis. This operation is implemented using the custom op **QUADAVG (arg1, arg2)**. The operation of QUADAVG is shown in the figure 5.21 below.

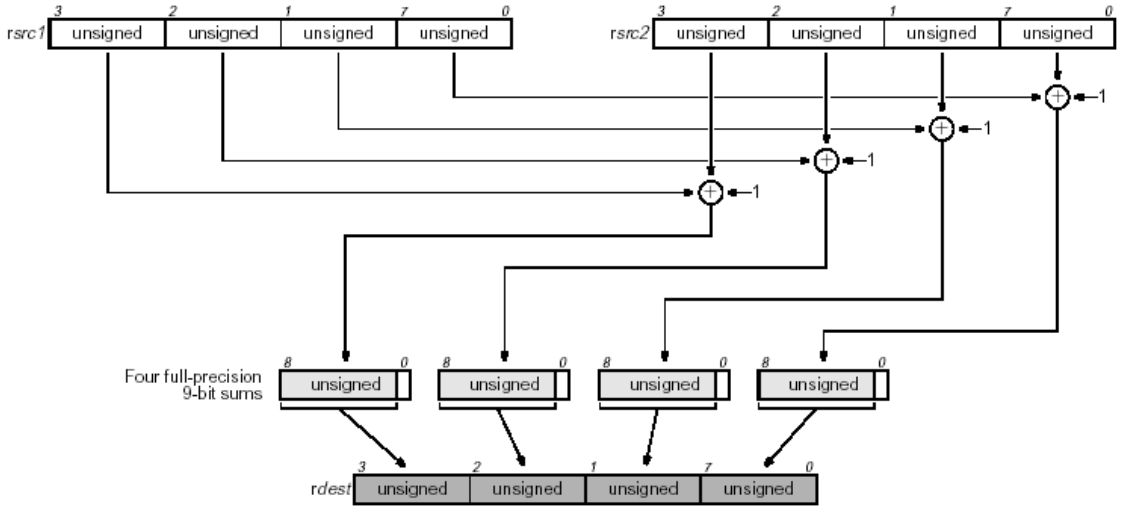


Figure 5.21 Custom Operation QUADAVG

The result is the output of the function. The flowchart of the above discussed process is given below:

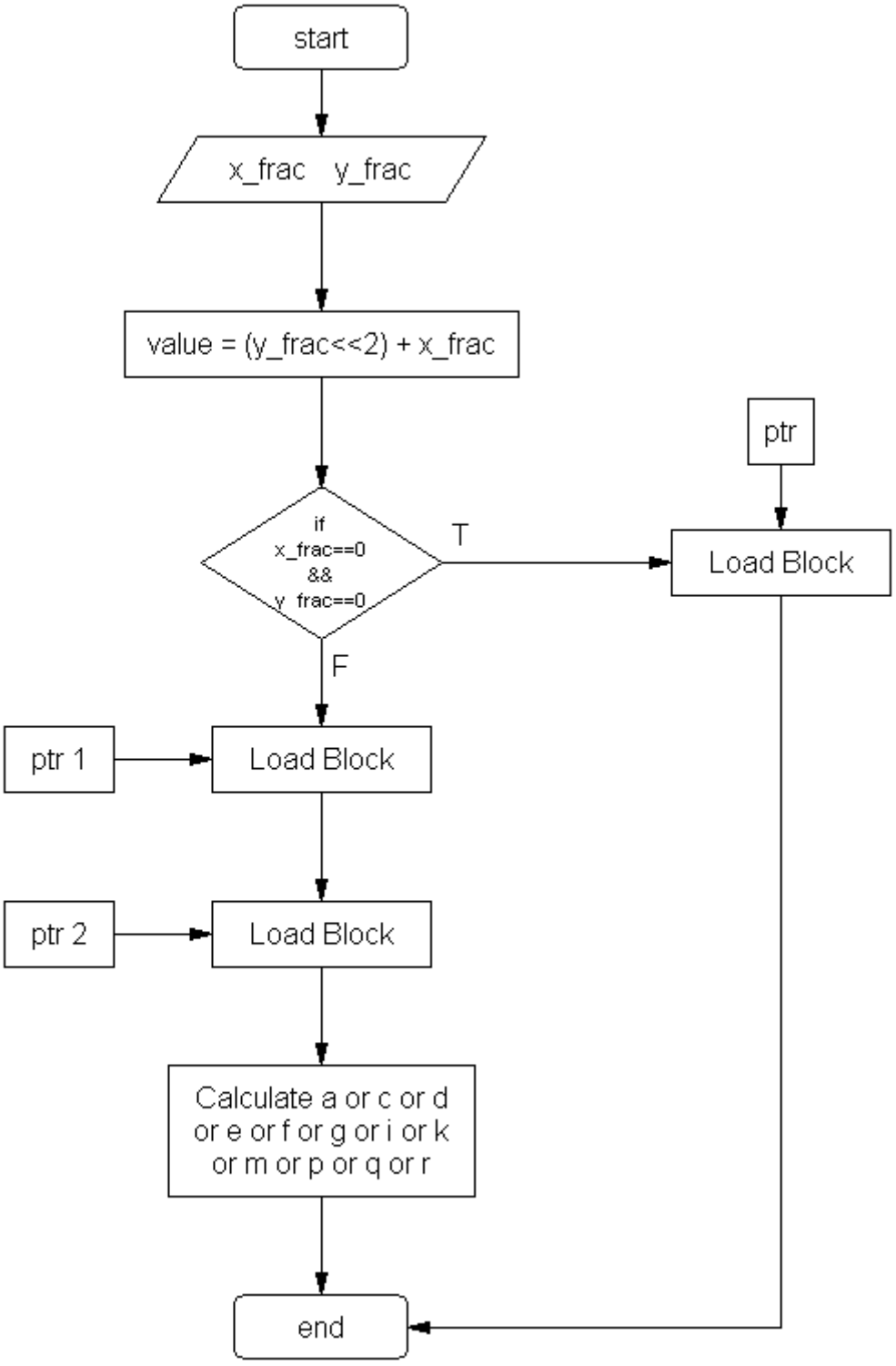


Figure 5.22 Flowchart of implementation on TM 1300

5.2.3 RESULTS OF OPTIMIZATION

There was a substantial decrease in the overall loads of the process of motion compensation. Initially the whole process was taking around **24 Mega Cycles** per frame. After optimization it was reduced to around **5 Mega Cycles** which is almost **5** times improvement in the processing speed.

F	Unoptimized	Getbloc	PRECOMP	Total Optimized
1	0	0	2.6	2.6
2	23.112	2.718	2.599	5.317
3	23.422	2.705	2.6	5.305
4	23.744	2.815	2.595	5.41
5	24.637	2.823	2.598	5.421
6	22.252	2.781	2.611	5.392
7	24.646	2.836	2.598	5.434
8	23.915	2.736	2.603	5.339
9	23.876	2.818	2.601	5.419
10	24.506	2.818	2.599	5.417
11	24.498	2.778	2.597	5.375
12	23.761	2.829	2.599	5.428
13	24.012	2.781	2.612	5.393
14	22.742	2.811	2.597	5.408
15	25.193	2.795	2.599	5.394
16	23.985	2.818	2.596	5.414
17	25.228	2.789	2.6	5.389

Figure 5.23 Before and After Optimizing Motion Compensation Luma

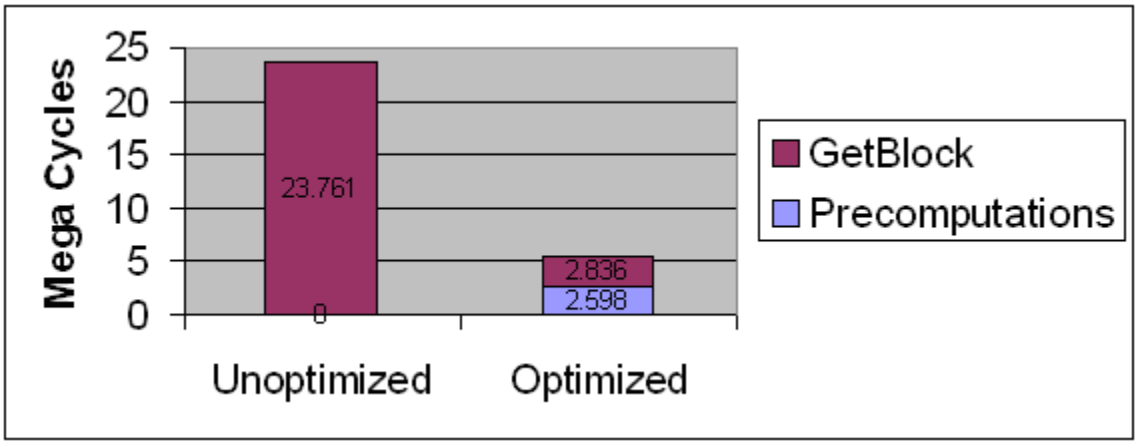


Figure 5.24 Optimization of Motion Compensation of Luma

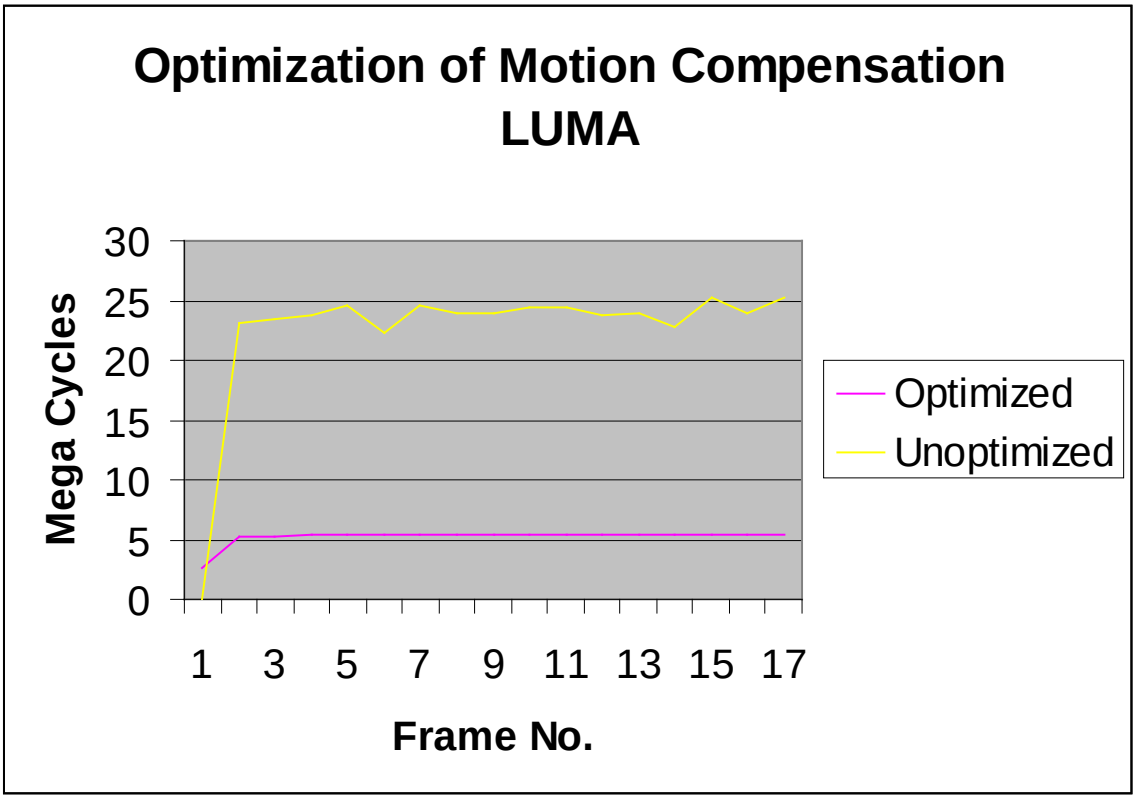


Figure 5.25 Optimization of Motion Compensation of Luma

5.3 OPTIMIZATION OF MOTION COMPENSATION OF CHROMA

5.3.1 MOTION COMPENSATION OF CHROMINANCE IN H.264

The resolution of each chroma component in a macroblock (Cr and Cb) is half that of the luminance (luma) component. Each chroma block is partitioned in the same way as the luma component, except that the partition sizes have exactly half the horizontal and vertical resolution (an 8x16 partition in luma corresponds to a 4x8 partition in chroma; an 8x4 partition in luma corresponds to 4x2 in chroma; and so on). The horizontal and vertical components of each motion vector (one per partition) are halved when applied to the chroma blocks.

The Motion Compensation of a chroma pixel goes upto $1/8^{th}$ of a pixel, in correspondence with ¼ of the luma component. The interpolation of the chroma pixel is

carried out by taking the weighted sum of the pixel itself and the three neighbouring pixels.

Inputs to this process are similar to that of Luma case which are:

1. A chroma location in full-sample units (xIntC, yIntC),

2. A chroma location offset in fractional-sample units (xFracC, yFracC), and

3. Chroma component samples from the selected reference picture refPicLXC.

Output of this process is a predicted chroma sample value predPartLXC[xC, yC].

In Figure 5.26, the positions labelled with A, B, C, and D represent chroma samples at full-sample locations inside the given two-dimensional array refPicLXC of chroma samples.

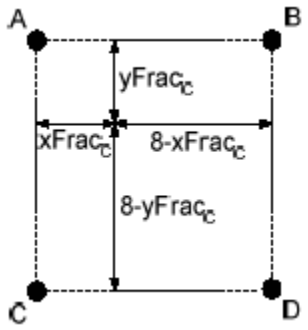


Figure 5.26 Fractional sample position dependent variables in chroma interpolation and surrounding integer position samples A, B, C, and D.

These samples may be used for generating the predicted chroma sample value predPartLXC[xC, yC].

xAC = Clip3(0, PicWidthInSamplesC – 1, xIntC)

xBC = Clip3(0, PicWidthInSamplesC – 1, xIntC + 1)

xCC = Clip3(0, PicWidthInSamplesC – 1, xIntC)

xDC = Clip3(0, PicWidthInSamplesC – 1, xIntC + 1)

yAC = Clip3(0, PicHeightInSamplesC – 1, yIntC)

yBC = Clip3(0, PicHeightInSamplesC – 1, yIntC)

yCC = Clip3(0, PicHeightInSamplesC – 1, yIntC + 1)

yDC = Clip3(0, PicHeightInSamplesC – 1, yIntC + 1)

Given the chroma samples A, B, C, and D at full-sample locations, the predicted chroma sample value predPartLXC[xC, yC] is derived as follows:

predPartLXC[xC, yC] = ((8 – xFracC) * (8 – yFracC) * A + xFracC * (8 – yFracC) * B + (8 – xFracC) * yFracC * C + xFracC * yFracC * D + 32) >> 6

5.3.2 IMPLEMENTATION OF MC CHROMA IN REFERENCE DECODER

Interpolation of Chroma samples in the reference decoder is implemented on pixel calculation basis. The decoder calculates the value of one subpixel in a single iteration. Thus, the calculations for a complete macroblock would require 64 iterations (8* 8 = 64). In each iteration, it checks if the motion vector is not pointing out of the picture so it has to apply clipping to all the four memory access. It also calculates the weights of filter coefficients in every iteration. This results in a large amount of load on the processor. The flow chart of the process is given below.

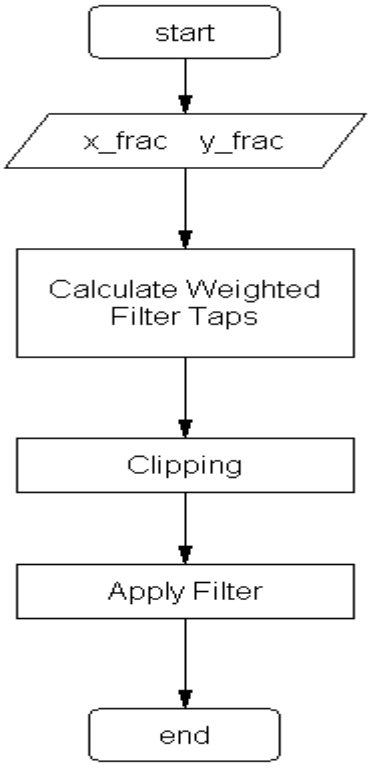


Figure 5.27 Implementation in Reference Decoder

5.3.3 IMPLEMENTATION OF CHROMA MC FOR TM – 1300

5.3.3.1 IMPLEMENTATION

The implementation of the chroma motion compensation uses the same concepts and ideas as for the luma case. Edges are created around the chroma portion of the image i.e. both for Cb and Cr. As there are no fixed weights of the filter coefficients therefore the concept of pre-computation cannot be applied here.

As currently we are only dealing with the 16x16 macroblock case of Motion Estimation at our encoder so instead of calculating the prediction values on pixel basis we carry out our calculation at 4x4 block basis. A 4x4 chroma block corresponds to a 8x8 Luma block. The limit is set at 8x8 and not 16x16 in terms of Luma because the output from the motion compensation is passed to IDCT. IDCT has a bottle neck and it operates at 4x4 blocks. So a 4x4 chroma just satisfies the requirement. We could have carried our calculations on an 8x8 block case but the bottle neck would have come at IDCT.

To calculate the interpolated values of the pixels at 4x4 block level we need actual image block size of 5x5. We read an 8x5 block using int*'s. An 8x5 block is read instead of a 5x5 block because we cannot read five bytes using int*'s. Furthermore it is helpful as we need to carry byte alignment of the read due to similar reasons as in case of Luma motion compensation.

The weighted coefficients of the filter are calculated and all the four values are packed in an integer. To calculate the value of one pixel we need its value along with three neighboring pixels. Now these pixels are not in one integer. So these have to be packed in one integer to make operations simple.

Different techniques are used for each column of pixels in a block. For the first column we use the custom op PACK16LSB (arg1, arg2). The custom op packs the Least Significant 16 bits of the two integers.

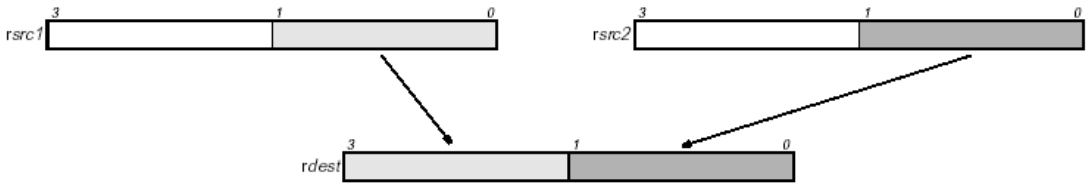


Figure 5.28 Packing of 16 LSB's of the two integers

The least significant bits are packed for the first column due to endian invert i.e. we are using the little endian scheme for our code. The code is as follows:

```
one=PACK16LSB(b,a);
```

For column 2, the data is aligned as:

```
int one1,two2,sort=0;
one1=FUNSHIFT1(a,sort);
two2=FUNSHIFT1(b,sort);
one=PACK16MSB(two2,one1);
```

Only the data to be used is moved to most significant 16 bits and the rest is discarded. It is then packed to achieve the result. For column 3, a procedure similar to column 1 is used except most significant 16 bits are packed. For column 4, the procedure is similar to column 2, except the shift is of 3 i.e. FUNSHIFT3 (arg1, arg2) is used.

To apply the filter we use the custom op **IFIR8UI** which gives the byte wise product of the coefficients and pixel values and finally sums them up.

Finally each interpolated pixel value is rounded and averaged. The overall effect of this process is a great reduction of the overall loads of the operations due no breaking of decision trees due to excessive nested loops.

Flow Chart of our implementation of Motion Compensation for Chroma is given below:

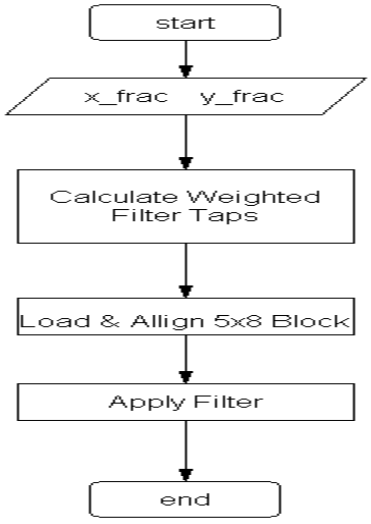


Figure 5.29 Implementation of Motion Compensation of Chroma on TM 1300

5.3.4 OPTIMIZATION RESULTS

The result of this optimization was a remarkable decrease in the loads of this process. Initially the loads of the chroma interpolation were around **17 Mega Cycles** per frame. After optimization these were reduced to around **2.5 Mega Cycles**. Hence, there was a **680%** improvement in the processing speed of the chroma motion compensation.

Frame No.	Unoptimized	Optimized
1	9.963	0
2	17.462	2.452
3	17.371	2.446
4	17.533	2.481
5	17.534	2.501
6	17.544	2.501
7	17.583	2.527
8	17.469	2.484
9	17.508	2.504
10	17.576	2.504
11	17.534	2.502
12	17.532	2.507
13	17.515	2.51
14	17.568	2.517
15	17.516	2.494
16	17.555	2.528
17	17.486	2.5

Table 5.5 Before and After Optimizing Motion Compensation Chroma

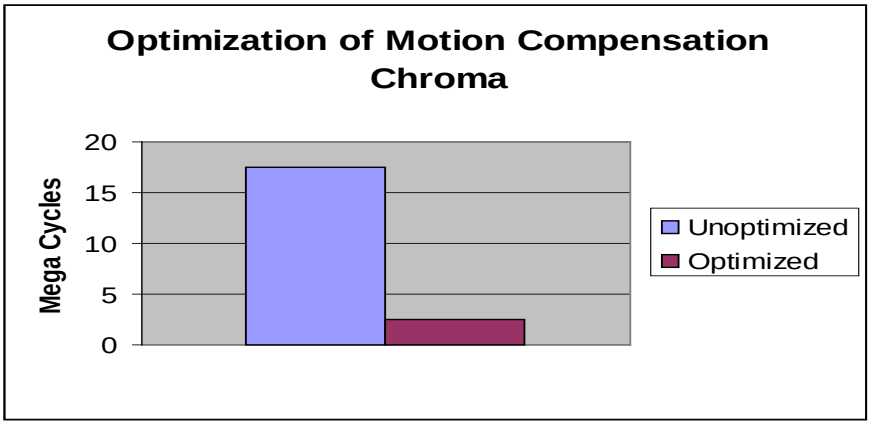


Figure 5.30 Optimization of Motion Compensation of Chroma

5.4 OPTIMIZATION OF IDCT

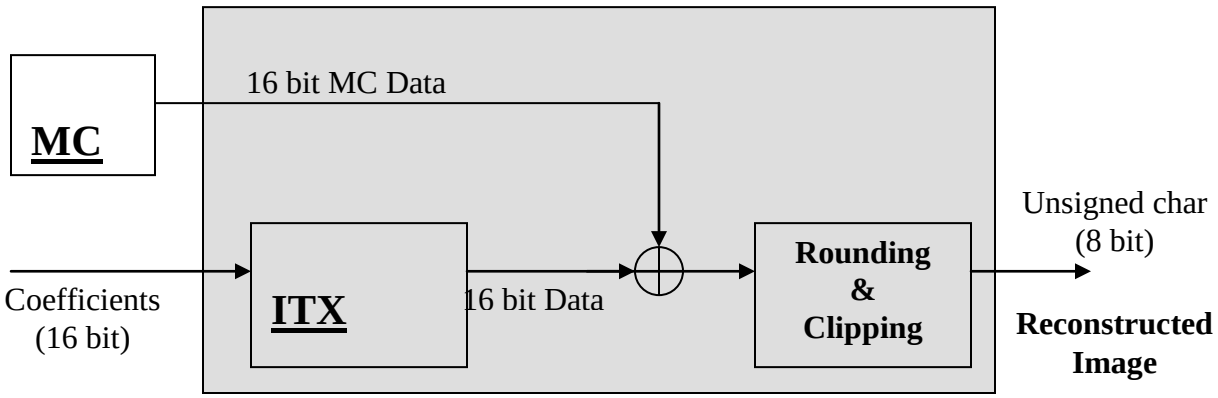


Figure 5.31 Block Diagram of a portion of an H264 Decoder

The portion in gray was implemented in the function itrans().A4x4 DCT of an input array A is given by:

$Y = TAT^T$

The input coefficients passed in the function are already transposed. Thus:

$Y = TA^T T^T$ or

$Y = T(TA)^T$

Its implementation is carried out by determining the intermediate values Y_1 and then calculating the final result Y.

STEP 1:

$Y_1 = TA$

$$\begin{pmatrix} y_100 & y_101 & y_102 & y_103 \\ y_110 & y_111 & y_112 & y_113 \\ y_120 & y_121 & y_122 & y_123 \\ y_130 & y_131 & y_132 & y_133 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 0.5 & 0 & -1 \\ 0 & 1 & 0 & 0.5 \end{pmatrix} \begin{pmatrix} a0 & b0 & c0 & d0 \\ a1 & b1 & c1 & d1 \\ a2 & b2 & c2 & d2 \\ a3 & b3 & c3 & d3 \end{pmatrix}$$

$$= \begin{pmatrix} a0 + a2 & b0 + b2 & c0 + c2 & d0 + d2 \\ a0 - a2 & b0 - b2 & c0 - c2 & d0 - d2 \\ a1/2 - a3 & b1/2 - b3 & c1/2 - c3 & d1/2 - d3 \\ a1 + a3/2 & b1 + b3/2 & c1 + c3/2 & d1 + d3/2 \end{pmatrix}$$

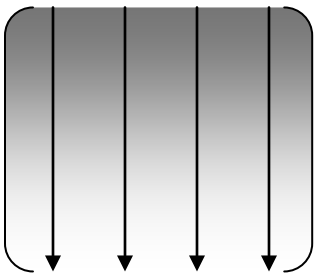
STEP 2:

$Y = TY_1^T$

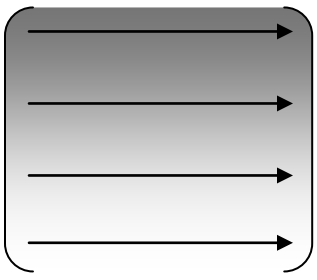
$$\begin{pmatrix} y00 & y_101 & y_102 & y_103 \\ y10 & y_111 & y_112 & y_113 \\ y20 & y_121 & y_122 & y_123 \\ y30 & y_131 & y_132 & y_133 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 0.5 & 0 & -1 \\ 0 & 1 & 0 & 0.5 \end{pmatrix} \begin{pmatrix} y_100 & y_110 & y_120 & y_130 \\ y_101 & y_111 & y_121 & y_131 \\ y_102 & y_112 & y_122 & y_132 \\ y_103 & y_113 & y_123 & y_133 \end{pmatrix}$$

$$= \begin{pmatrix} y_100 + y_102 & y_100 + y_102 & y_100 + y_102 & y_100 + y_102 \\ y_100 - y_102 & y_100 - y_102 & y_100 - y_102 & y_100 - y_102 \\ y_101/2 - y_103 & y_101/2 - y_103 & y_101/2 - y_103 & y_101/2 - y_103 \\ y_101 + y_103/2 & y_101 + y_103/2 & y_101 + y_103/2 & y_101 + y_103/2 \end{pmatrix}$$

Inverse transform is applied sequentially on all four columns and then the rows of the input 4x4 block. Both step 1 and step 2 and performed for one 1D transform.
1D transforms are supposed to be applied in the following 2 steps.



Vertical



Horizontal

Direct Implementation:

```
#define H264trans1D(sa,sb,sc,sd)\
{\
unsigned short se,sf,sg,sh;\
    se=sa+sc;\                  //STEP 1
    sf=sa-sc;\
    sg=(sb>>1)-sd;\
    sh=sb+(sd>>1);\
\
    sa=se+sh;\                  //STEP 2
    sb=sf+sg;\
    sc=sf-sg;\
    sd=se-sh;\
}

// executed 8 times (4 columns & 4 rows) for complete
// 4x4 block transformation
```

5.4.1 OUR IMPLEMENTATION

As the input data is in shorts, we pack the shorts into integers and apply the transform on two columns at a time, rather than the above process, and then repeat the procedure for the remaining two columns. The columns ‘a’ and ‘b’ are combined as ‘ab’ and ‘c’ and ‘d’ are combined as ‘cd’

The above equations in the form of integers become:

STEP 1:

$$\begin{pmatrix} y_100 & y_101 \\ y_110 & y_111 \\ y_120 & y_121 \\ y_130 & y_131 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 0.5 & 0 & -1 \\ 0 & 1 & 0 & 0.5 \end{pmatrix} \begin{pmatrix} ab0 & cd0 \\ ab1 & cd1 \\ ab2 & cd2 \\ ab3 & cd3 \end{pmatrix}$$

$$\begin{pmatrix} ab0 + ab2 & cd0 + cd2 \\ ab0 – ab2 & cd0 - cd2 \\ ab1/2 – ab3 & cd1/2 – cd3 \\ ab1 + ab3/2 & cd1 + cd3/2 \end{pmatrix}$$

=

STEP 2:

$$\begin{pmatrix} y00 & y_1 01 \\ y10 & y_1 11 \\ y20 & y_1 21 \\ y30 & y_1 31 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 0.5 & 0 & -1 \\ 0 & 1 & 0 & 0.5 \end{pmatrix} \begin{pmatrix} y_1 00 & y_1 10 \\ y_1 01 & y_1 11 \\ y_1 02 & y_1 12 \\ y_1 03 & y_1 13 \end{pmatrix}$$

$$= \begin{pmatrix} y_1 00 + y_1 02 & y_1 00 + y_1 02 \\ y_1 00 - y_1 02 & y_1 00 - y_1 02 \\ y_1 01/2 - y_1 03 & y_1 01/2 - y_1 03 \\ y_1 01 + y_1 03/2 & y_1 01 + y_1 03/2 \end{pmatrix}$$

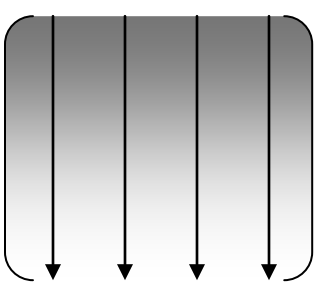
The shorts packed into integers are added, subtracted and shifted using DSPIDUAL functions, discussed in the end of this chapter, which process the 16LSBs and 16MSBs of an integer individually.

Code Ref:

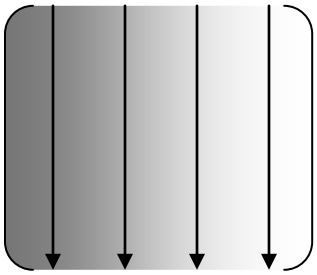
```
#define H264trans1D(sa,sb,sc,sd)\
{\
unsigned int se,sf,sg,sh;\
    se=DSPIDUALADD(sa,sc);\           //STEP 1
    sf=DSPIDUALSUB(sa,sc);\
    sg=DSPIDUALSUB(DUALASR(sb,1),sd);\
    sh=DSPIDUALADD(sb,DUALASR(sd,1));\
\
    sa=DSPIDUALADD(se,sh);\           //STEP 2
    sb=DSPIDUALADD(sf,sg);\
    sc=DSPIDUALSUB(sf,sg);\
    sd=DSPIDUALSUB(se,sh);\
}

// executed 4 times for a complete 4x4 block
// transformation. Half processing than the
// previous method.
```

After applying the process on the columns, the same process is then applied on all the four rows of the resulting matrix after transposing it. Transpose is taken by individually taking four 2x2 transposes using the PACKLSB and PACKMSB custom operations of TM1300 (discussed in the end of this chapter) and moving the results to their proper registers.



Vertical



Horizontal

Code Ref:

```
H264trans1D(ab1,ab2,ab3,ab4);      //vertical
H264trans1D(cd1,cd2,cd3,cd4);

Transpose2x2(ab1,ab2,sab1,sab2);   //transposition
Transpose2x2(ab3,ab4,sab3,sab4);
Transpose2x2(cd1,cd2,scd1,scd2);
Transpose2x2(cd3,cd4,scd3,scd4);


H264trans1D(sab1,sab2,scd1,scd2);  //horizontal
H264trans1D(sab3,sab4,scd3,scd4);
```

5.4.1.1 RECONSTRUCTION

Reconstructed sample values (section 8.6.1.1) have to be calculated using the result of the inverse transform and the predicted block.

Inputs:

- Index of the 4x4 inverse transformed block.
- Inter prediction samples for the current macroblock.
- Prediction residual transform coefficient levels.

Output:Decoded samples of the current macroblock prior to the deblocking filter process.

A 4x4 block is picked from the 16x16 block of scaled predicted values. For this, the predicted values are picked up using a 1 dimensional pointer initially pointing at the offset value of the predicted 16x16 array and later by giving the suitable increments to access the subsequent elements of the required 4x4 block.

Predicted values are required to be scaled up by a factor of 64. This is done by using the custom operation `DSPIDUALMUL(a,b)` of TM1300. This custom operation basically multiplies the 16 LSBs and the 16 MSBs of an integer ‘a’ with the 16 LSBs and the 16 MSBs of integer ‘b’ respectively. Thus scaling our 2 shorts of 16 bits in one operation. Multiplication with 0x00000040 is equivalent to scaling up by 64.

Code reference:

```
DSPIDUALMUL(mab,0x00400040);
```

Where mab is an integer value, which holds two motion compensated shorts of a 4x4 array which are scaled up by 64 by individual multiplication with 16bit 0x0040.

The corresponding 16 LSBs and 16MSBs of the scaled predicted values, the LevelScale (explained in the equation 8-254 in the standard) and the inverse transformed values are added by the using the custom operation `DSPIDUALADD(a,b)`.

Code reference:

```
DSPIDUALADD(DSPIDUALADD(y00,RESULT),0x00200020)
```

*Where RESULT is the scaled motion compensated values calculated from the previous code reference and y00 is transformed values.

In accordance with equation 8-287 of the standard, the computed value is shifted to right by 6 bits and then clipped in the range of 0-255

The scaling down is done by using the custom operation `DUALASR(a,b)` that shifts the 16 MSBs and 16 LSBs of an integer ‘a’ by ‘b’ bits individually.

The clipping is achieved by the custom operation `DUALUCLIPI(a,b)` that clips the 16 MSBs and 16 LSBs in the range of 0-255 individually.

Code reference:

```
DUALUCLIPI(DUALASR(RESULT2,6),255);
```

(Where `RESULT2` is the sum calculated in the previous code reference.)

This gives us the unsigned character array of reconstructed block prior to the deblocking filter and completes the inverse transformation and the reconstruction process.

5.4.1.2 CUSTOM OPERATIONS USED

The custom operations of the TriMedia processor that have been used in the optimization of the Inverse Discrete Cosine Transform are Dspidualadd, Dspidualsub, Dspidualmul, Pack16msb,Pack16lsb, Dualuclipi and Dualasr. The details of these custom operations can be viewed in the appendix.

5.4.1.3 RESULTS

The following graphs show the obtained results after the optimization of Inverse DCT.

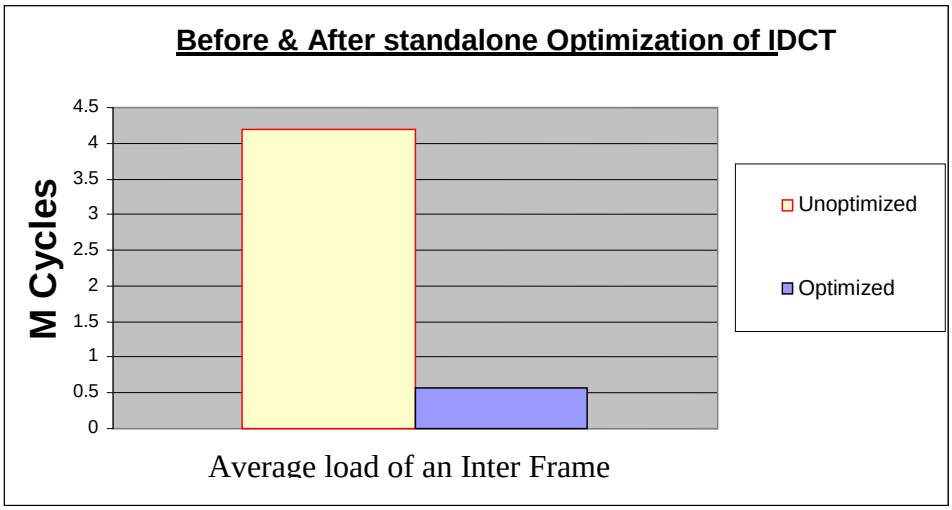


Figure 5.31 Before and After the Stand Alone Implementation of IDCT

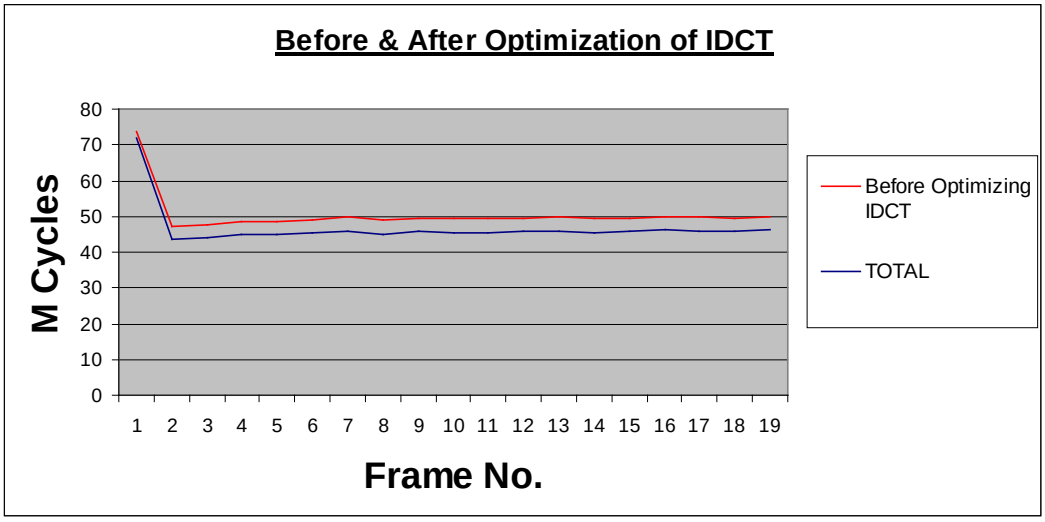


Figure 5.32 Before and After Optimization of IDCT

CHAPTER 6

DEVELOPMENT OF DISPLAYS

6.1 INTRODUCTION TO DISPLAY DRIVERS

The image after decoding is in raw YUV 4:2:0 format. It can either be stored to the hard disk or it can be sent to a rendering device such as a television or monitor. The YUV data decoded, by the decoder, does not have the necessary information, required by the CCIR 656 standard, to be displayed on the television. In order to convert data to the desired format, an on board chip, SAA-7125 is used along with the TriMedia processor. The software module built to use the SAA-7125 chip is called a video out driver. The onboard device is initialized once at startup and is then used to display the decoded image on the television. This chapter will shed light on the basic operation of the television scanning and the working of the video out driver.

Images on a television screen are made up of pixels present in horizontal lines. An image is formed by the electron gun moving horizontally from left to right and top to bottom one line after another. Hence an electron gun moves both from left to right as well as from top to bottom. It simply means that images seen on a television screen are made by an electron gun moving from the top left corner of the screen to the bottom right corner. Another important fact is that one image as seen on the television is not created in a single scan of the electron gun, from the top left corner to the bottom right corner, but by two such scans. Each of these scans is called a field. Hence we can say that two fields scanned by the electron gun form a frame of video or an image. Let us consider that a television screen has ten lines then in the first scan the gun would scan odd numbered lines namely 1,3,5,7,9 and in the second scan even numbered lines 2,4,6,8,10, thus completing one frame. Such an image, formed from the combination of two fields, is called an interlaced image. It also implies that the field rate is always twice the frame rate. If we have a frame rate of 20 frames per second, the field rate would be 40 fields per second and the video will give the same effect of that of 40 frames per second.An illustrated example of an interlaced image is shown in figure 6.1 below:

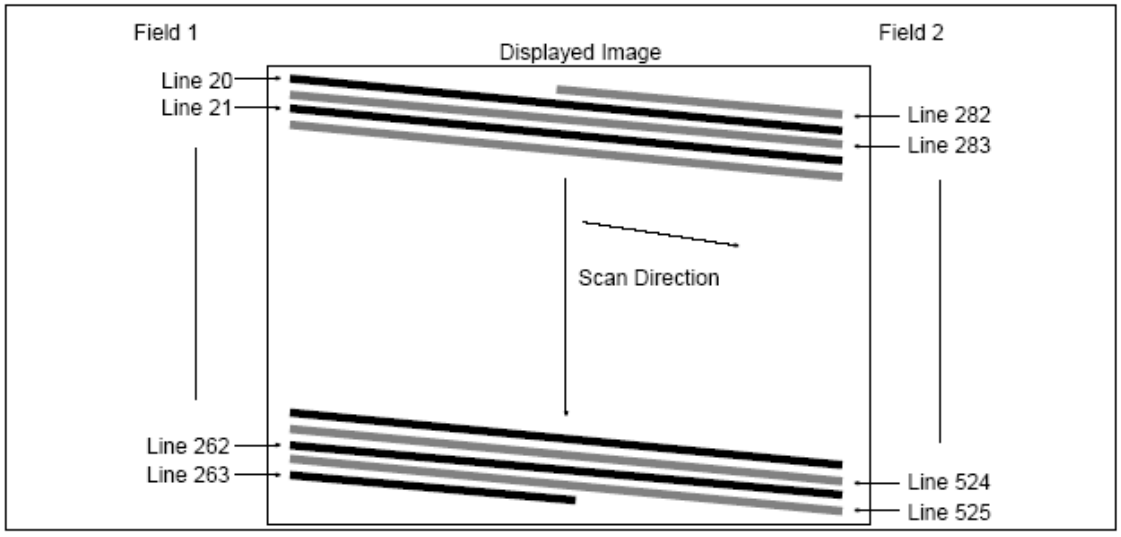


Figure 6.1 An Interlaced Image Scan Pattern

6.2 DEVELOPMENT OF DISPLAY FOR TELEVISION

In order to display this data on a television the CCIR 656 standard was defined. The standard defines that for PAL standard the number of pixels and lines is 720 x 525 and incase of NTSC is 720 x 480. The standard defines that each line is to be preceded by a SAV (Start Active Video) and the completion of a single line is to be followed by an EAV (End Active Video) code. The advantage of using SAA-7125 is that once it is initialized it adds the SAV and EAV information to the video frames itself.

6.3 STEPS

In order to make the SAA-7125 work properly a device driver or in other words a video out driver, software module, needs to be created. The device driver consists of two parts one part is concerned with the proper initialization of the video out unit SAA-7125 and the other part is concerned with the working. The two parts can be defined by four distinct steps:

STEP1 – Open up an instance of the video out driver.

STEP2 – Setup up an instance of the video out driver.

STEP3 – Setup the video settings for the video out driver.

STEP4 – Start the video out driver.

The first three steps are part of the initialization process and the final step is related to the working of the video out driver.

6.3.1 INITIALIZATION PROCESS FOR VIDEO DRIVER

In the first step we initialize an instance of the video out driver. This is done by opening an instance of the video out driver using the device library api

6.3.1.1 STEP 1

voOpen( )

This function checks if there is a video out driver unit attached to the TriMedia processor and in case of success returns an instance of the video out driver.

6.3.1.2 STEP 2

In the second step we initialize an instance of the video out driver. This is done by calling the device library api function

voInstanceSetup( ) and passing the **voInstanceSetup_t** structure to the function

The **voInstanceSetup** structure consists of the following fields:

typedef struct **voInstanceSetup**_t

{

Bool	**hbeEnable**;
Bool	**underrunEnable**;
UInt32	**ddsFrequency**;
intPriority_t	**interruptPriority**;
void	(***isr**)(void);
tmVideoAnalogStandard_t	**videoStandard**;
tmVideoAnalogAdapter_t	**adapterType**;

}

voInstanceSetup_t, ***pvoInstanceSetup_t**;

hbeEnable True enables interrupts for highway bandwidth errors.

underrunEnable True enables interrupts when an under-run occurs.

ddsFrequency Frequency, in Hertz, to be set to 27000000 for TriMedia .

interruptPriority VO interrupt priority.

isr Pointer to the interrupt service routine.

videoStandard Video standard to which the video-out decoder on the board must be programmed. NTSC or PAL.

adapterType The adaptor type (either CVBS and Svideo)

This structure is used as the common initializing structure for all video-out modes of operation, including YUV images and raw data streaming modes. It is passed to the voInstanceSetup function to perform an initial setup of the video-out peripheral.

The setting used by us are

hbeEnable = True

underrunEnable = True

ddsFrequency = 27000000

interruptPriority = intPRIO_3 (Medium Priority)

isr = voTestISR (Function name to be called to process isr)

videoStandard = NTSC / PAL (depending upon the height of the image)

adapterType = None (This means video is output on both the CVBS and SVideo Connectors

6.3.1.3 STEP 3

In the third step we define the YUV settings for the video out driver. This is done with the following device library api function

voYUVSetup ( ) and passing the **voYUVSetup_t** structure to the function.

The **voYUVSetup_t** structure consists of the following fields

typedef struct **voYUVSetup_t**

{

Bool **buf1emptyEnable**;

Bool **yThresholdEnable**;

voYUVModes_t **mode**;

UInt **imageVertOffset**, **imageHorzOffset**;

UInt **imageWidth**, **imageHeight**;

UInt **yThreshold**;
UInt **yStride**, **uStride**, **vStride**;
Pointer **yBase**, **uBase**, **vBase**;
}
voYUVSetup_t, ***pvoYUVSetup_t**;

buf1emptyEnable True enables interrupt when buffer 1 is empty.
yThresholdEnable True enables Y threshold interrupt.
mode The image mode of operation. vo420_UNSCALED, vo422_COSITED_UNSCALED, vo420_SCALED to name a few.
imageVertOffset, **imageHorzOffset** Vertical and horizontal start of the upper left corner of the output.
imageWidth, **imageHeight** Image width and height in samples.
yThreshold When the yThresholdEnable flag is true, an interrupt will be generated when the line counter reaches this value.
yStride, **uStride**, **vStride** Number of bytes from the start of one line to the start of the next line. The values depend on the YUV image mode.
yBase, **uBase**, **vBase** Pointers to the start of the YUV data.

The structure is passed to the voYUVSetup function to set up the video-out peripheral in image mode.

The setting that have been used are;
buf1emptyEnable = True
yThresholdEnable = False
mode =.vo420_UNSCALED,
imageVertOffset = 120
imageHorzOffset = 176
imageWidth = 352
imageHeight = 240 / 288 (Depending on the height of the image)
yThreshold = False

yStride = 704

uStride = 352

vStride = 352

yBase, **uBase**, **vBase** = Pointers to the start of the YUV data. These are the buffer pointers that are used by the video out unit to display the data on the television.

Now the video out driver is initialized and ready to be started.

6.3.2 WORKING OF VIDEO DRIVER

6.3.2.1 STEP 4

In the fourth step we start the video out driver. This is done using the device library api function

voStart( )

This function starts the video out driver unit attached to the TriMedia processor and waits for interrupts to occur. Each interrupt that occurs sets a pre-defined flag in the MMIO registers (Memory Mapped Device Registers) of the TriMedia processor. Each flag represents a different interrupt and after checking these flags the appropriate function can be called to handle this interrupt. The video out driver runs on this principle.

Once the driver starts running interrupts occur. These interrupts are processed in the **voTestISR** function. This function handles the following interrupts in this video out driver:

voAckYTR_ACK( )

voAckURUN_ACK( )

voAckHBE_ACK( )

voAckBFR2_ACK( )

voAckBFR1_ACK( )

An acknowledge means that the interrupt has been handled or processed by the video out driver and reset the MMIO register. In this driver we only process the case of

voAckBFR1_ACK( ). Incase of any other interrupt the interrupt is acknowledged but no action is taken against it.

We check the status of the video out status buffer. If the status is VO_BUF1EMPTY then this means that the previous buffer containing the video image has been displayed and the video driver is waiting for a new image. Incase if the new image is not available the video out driver is repeatedly going to display the same image. If the status is VO_BUF1EMPTY then check if the field last displayed is the first (odd field) or second (even field). If the field is the first field then don't change the image. Keep the old data pointers and wait till the second field is displayed by the driver. This is necessary because both fields combined together would form the complete image. If the buffer is changed after only one field is rendered then this would result in an image in which we would see an image with alternate black lines. If the field displayed is the second field then get a new image, as implemented in the driver it means acquire new pointers for the image. Now that we have the new image we can start all over displaying the first field and then the second field.

In this version of the decoder we only decode CIF (352 x 288) / SIF (352 x 240) size images. The buffers allocated to hold the image to display are however 704 x 480 size buffers. This means that a small image of 352 x 288/240 can placed anywhere in the buffer of dimension 704 x 480. Using the **imageHorzOffset** and **imageVertOffset** we can place the image as we choose. We also set the stride of the image to two times 352, 704. This means that the image is going to be placed in the 704 x 480 size buffer line after line. This means that the driver is going to display the first line of the image and then the third line and not the second. After the third line fifth, seventh, ninth and so on, only odd lines. This happens because of the interlaced nature of the video out unit. Thus it would first display one field. Next the interrupt occurs indicating that the first field has been displayed and the second field is to be displayed. Since we don't change the buffer the video out unit starts reading and displaying the even lines of the image. In this way the whole image, both even and odd lines, is displayed on the television screen.

CONCLUSION

The aim of the project was to optimize the H.264 video Decoder to a level that it could support web based video applications. These applications normally send video streams at 6 to 8 fps. After optimizing the H.264 video decoder we were able to decode the streams at 6 fps, hence achieving the benchmark frame rate for web based video applications.

In our project we targeted the computationally intensive modules of H.264 namely Variable Length Decoding (VLD), Inverse Discrete Cosine Transform (IDCT) and Motion Compensation (MC). Also apart from developing new and more efficient algorithms for these modules we also carried out small level optimization by improving the memory access procedures. The code was also restructured and various operations were related with the Custom Ops of the DSP to fully utilize the processing capabilities of the process.

The project provided us the opportunity to understand the various processes involved in the optimization of a code. The code restructuring is an essential part of the optimization process which was a new idea to us. The process of modifying existing algorithms and developing more efficient algorithms was completely new experience for us.

There is still room for improvement and further optimizations can be made to De-blocking filter, Intra Prediction, initialization processes and memory management issues. This project can be used as a base to further work on the optimization of H.264 as a future project or to develop a streaming video web application.

REFERENCES:

[1] 'Implementation Of Real Time Mpeg-1 Encoder For The Philips Trimedia', Thesis Of Gerrit Wenig, Erstellt Im Systems Laboratory, Hamburg

[2] http://www.mpeg.org

[3] "SP parallel programming workshop: optimization topics" George Gusciora, MHPCC.

[4] "Optimization and Tuning Guide for Fortran, C, and C++". AIX Version 4. IBM Corporation. June 1996.

[5] ITU-T Rec. H.264 / ISO/IEC 11496-10, "Advanced Video Coding", Final Committee Draft, Document JVT-E022, September 2002

[6] *www.vcodex.com H.264 / MPEG-4 Part 10 : Overview*

[7] Philips TriMedia TM 1300 Databook.

Appendix A

TrailingOnes (coeff_token)	TotalCoeff (coeff_token)	0 <= nC < 2	2 <= nC < 4	4 <= nC < 8	8 <= nC	nC == -1
0	0	1	11	1111	0000 11	01
0	1	0001 01	0010 11	0011 11	0000 00	0001 11
1	1	01	10	1110	0000 01	1
0	2	0000 0111	0001 11	0010 11	0001 00	0001 00
1	2	0001 00	0011 1	0111 1	0001 01	0001 10
2	2	001	011	1101	0001 10	001
0	3	0000 0011 1	0000 111	0010 00	0010 00	0000 11
1	3	0000 0110	0010 10	0110 0	0010 01	0000 011
2	3	0000 101	0010 01	0111 0	0010 10	0000 010
3	3	0001 1	0101	1100	0010 11	0001 01
0	4	0000 0001 11	0000 0111	0001 111	0011 00	0000 10
1	4	0000 0011 0	0001 10	0101 0	0011 01	0000 0011
2	4	0000 0101	0001 01	0101 1	0011 10	0000 0010
3	4	0000 11	0100	1011	0011 11	0000 000
0	5	0000 0000 111	0000 0100	0001 011	0100 00	-
1	5	0000 0001 10	0000 110	0100 0	0100 01	-
2	5	0000 0010 1	0000 101	0100 1	0100 10	-
3	5	0000 100	0011 0	1010	0100 11	-
0	6	0000 0000 0111 1	0000 0011 1	0001 001	0101 00	-
1	6	0000 0000 110	0000 0110	0011 10	0101 01	-
2	6	0000 0001 01	0000 0101	0011 01	0101 10	-
3	6	0000 0100	0010 00	1001	0101 11	-
0	7	0000 0000 0101 1	0000 0001 111	0001 000	0110 00	-
1	7	0000 0000 0111 0	0000 0011 0	0010 10	0110 01	-
2	7	0000 0000 101	0000 0010 1	0010 01	0110 10	-
3	7	0000 0010 0	0001 00	1000	0110 11	-
0	8	0000 0000 0100 0	0000 0001 011	0000 1111	0111 00	-
1	8	0000 0000 0101 0	0000 0001 110	0001 110	0111 01	-
2	8	0000 0000 0110 1	0000 0001 101	0001 101	0111 10	-
3	8	0000 0001 00	0000 100	0110 1	0111 11	-
0	9	0000 0000 0011 11	0000 0000 1111	0000 1011	1000 00	-
1	9	0000 0000 0011 10	0000 0001 010	0000 1110	1000 01	-
2	9	0000 0000 0100 1	0000 0001 001	0001 010	1000 10	-

Table A. coeff_token mapping to Total Coeff (coeff_token) and Trailing Ones (coeff_token)

total_zeros	TotalCoeff(coeff_token)						
	1	2	3	4	5	6	7
0	1	111	0101	0001 1	0101	0000 01	0000 01
1	011	110	111	111	0100	0000 1	0000 1
2	010	101	110	0101	0011	111	101
3	0011	100	101	0100	111	110	100
4	0010	011	0100	110	110	101	011
5	0001 1	0101	0011	101	101	100	11
6	0001 0	0100	100	100	100	011	010
7	0000 11	0011	011	0011	011	010	0001
8	0000 10	0010	0010	011	0010	0001	001
9	0000 011	0001 1	0001 1	0010	0000 1	001	0000 00
10	0000 010	0001 0	0001 0	0001 0	0001	0000 00	
11	0000 0011	0000 11	0000 01	0000 1	0000 0		
12	0000 0010	0000 10	0000 1	0000 0			
13	0000 0001 1	0000 01	0000 00				
14	0000 0001 0	0000 00					
15	0000 0000 1						

Table B. Total_Zero Tables For 4x4 Blocks With Total Coeff (Coeff_Token) 1 To 7

total_zeros	TotalCoeff(coeff_token)							
	8	9	10	11	12	13	14	15
0	0000 01	0000 01	0000 1	0000	0000	000	00	0
1	0001	0000 00	0000 0	0001	0001	001	01	1
2	0000 1	0001	001	001	01	1	1	
3	011	11	11	010	1	01		
4	11	10	10	1	001			
5	10	001	01	011				
6	010	01	0001					
7	001	0000 1						
8	0000 00							

Table C. Total_Zeros Tables For 4x4 Blocks With Total Coeff (Coeff_Token) 8 To 15

run_before	zerosLeft						
	1	2	3	4	5	6	>6
0	1	1	11	11	11	11	111
1	0	01	10	10	10	000	110
2	-	00	01	01	011	001	101
3	-	-	00	001	010	011	100
4	-	-	-	000	001	010	011
5	-	-	-	-	000	101	010
6	-	-	-	-	-	100	001
7	-	-	-	-	-	-	0001
8		-	-	-	-	-	00001
9	-	-	-	-	-	-	000001
10	-	-	-	-	-	-	0000001
11	-	-	-	-	-	-	00000001
12	-	-	-	-	-	-	000000001
13	-	-	-	-	-	-	0000000001
14	-	-	-	-	-	-	00000000001

Table D Tables for run_before

Appendix B

Pack most-significant 16 bits — pack16msb

SYNTAX

```
[ IF rguard ] pack16msb rsrc1 rsrc2 → rdest
```

FUNCTION

```
If rguard then {
    rdest<15:0> ← rsrc2<31:16>
    rdest<31:16> ← rsrc1<31:16>
}
```

ATTRIBUTES

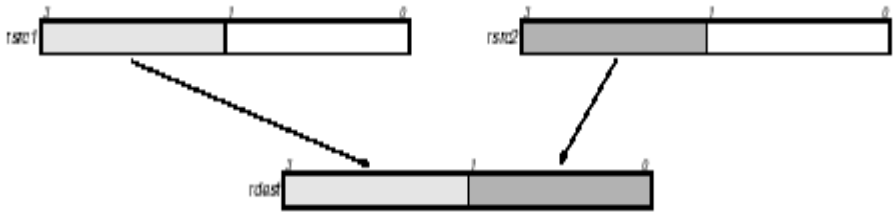
Function unit	alu
Operation code	54
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16lsb packbytes
mergelsb mergemsb

DESCRIPTION

As shown below, the `pack16msb` operation packs the two most-significant halfwords from the arguments *rsrc1* and *rsrc2* into *rdest*. The halfword from *rsrc1* is packed into the most-significant halfword of *rdest*; the halfword from *rsrc2* is packed into the least-significant halfword of *rdest*.



The `pack16msb` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`pack16msb r30 r40 → r50`	r50 ← 0x1234aabb
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 pack16msb r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 pack16msb r40 r30 → r70`	r70 ← 0xaabb1234

pack16lsb

Pack least-significant 16-bit halfwords

SYNTAX

```
[ IF rguard ] pack16lsb rsrc1 rsrc2 → rdest
```

FUNCTION

If *rguard* then {
    rdest<15:0> ← *rsrc2*<15:0>
    rdest<31:16> ← *rsrc1*<15:0>
}

ATTRIBUTES

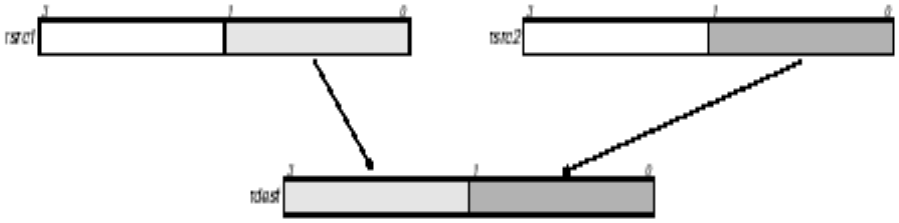
Function unit	alu
Operation code	53
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16msb packbytes
mergelsb mergemsb

DESCRIPTION

As shown below, the `pack16lsb` operation packs the two least-significant halfwords from the arguments *rsrc1* and *rsrc2* into *rdest*. The halfword from *rsrc1* is packed into the most-significant halfword of *rdest*; the halfword from *rsrc2* is packed into the least-significant halfword of *rdest*.



The `pack16lsb` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`pack16lsb r30 r40 → r50`	r50 ← 0x5678ccdd
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 pack16lsb r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 pack16lsb r40 r30 → r70`	r70 ← 0xccdd5678

mergemsb

Merge most-significant byte

SYNTAX

```
[ IF rguard ] mergemsb rsrc1 rsrc2 → rdest
```

FUNCTION

If *rguard* then {
  rdest<7:0> ← *rsrc2*<23:15>
  rdest<15:8> ← *rsrc1*<23:15>
  rdest<23:16> ← *rsrc2*<31:24>
  rdest<31:24> ← *rsrc1*<31:24>
}

ATTRIBUTES

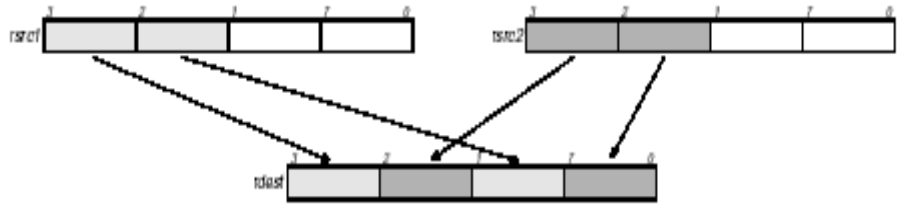
Function unit	alu
Operation code	58
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

pack16lsb pack16msb
packbytes mergelsb

DESCRIPTION

As shown below, the `mergemsb` operation interleaves the two pairs of most-significant bytes from the arguments *rsrc1* and *rsrc2* into *rdest*. The second-most-significant byte from *rsrc2* is packed into the least-significant byte of *rdest*; the second-most-significant byte from *rsrc1* is packed into the second-least-significant byte of *rdest*; the most-significant byte from *rsrc2* is packed into the second-most-significant byte of *rdest*; and the most-significant byte from *rsrc1* is packed into the most-significant byte of *rdest*.



The `mergemsb` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`mergemsb r30 r40 → r50`	r50 ← 0x12aa34bb
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 mergemsb r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 mergemsb r40 r30 → r70`	r70 ← 0xaa12bb34

Merge least-significant byte

mergelsb

SYNTAX

```
[ IF rguard ] mergelsb rsrc1 rsrc2 → rdest
```

FUNCTION

If *rguard* then {
  rdest<7:0> ← *rsrc2*<7:0>
  rdest<15:8> ← *rsrc1*<7:0>
  rdest<23:16> ← *rsrc2*<15:8>
  rdest<31:24> ← *rsrc1*<15:8>
}

ATTRIBUTES

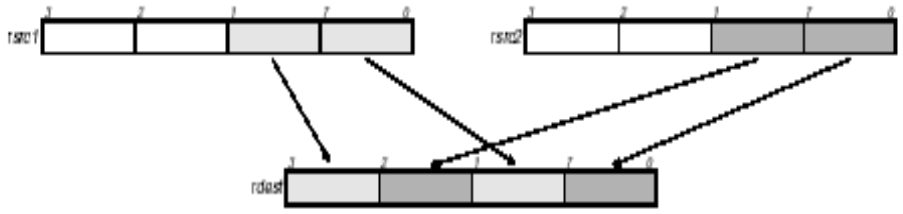
Function unit	alu
Operation code	57
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2, 3, 4, 5

SEE ALSO

packlolsb packlomsb
packbytes mergemsb

DESCRIPTION

As shown below, the `mergelsb` operation interleaves the two pairs of least-significant bytes from the arguments *rsrc1* and *rsrc2* into *rdest*. The least-significant byte from *rsrc2* is packed into the least-significant byte of *rdest*; the least-significant byte from *rsrc1* is packed into the second-least-significant byte of *rdest*; the second-least-significant byte from *rsrc2* is packed into the second-most-significant byte of *rdest*; and the second-least-significant byte from *rsrc1* is packed into the most-significant byte of *rdest*.



The `mergelsb` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is unchanged.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x12345678, r40 = 0xaabbccdd	`mergelsb r30 r40 → r50`	r50 ← 0x56cc78dd
r10 = 0, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r10 mergelsb r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0xaabbccdd, r30 = 0x12345678	`IF r20 mergelsb r40 r30 → r70`	r70 ← 0xcc56dd78

quadavg

Unsigned byte-wise quad average

SYNTAX

```
[ IF rguard ] quadavg rsrc1 rsrc2 → rdest
```

FUNCTION

```
If rguard then {
  temp ← (zero_ext8to32(rsrc1<7:0>) + zero_ext8to32(rsrc2<7:0>) + 1) / 2
  rdest<7:0> ← temp<7:0>
  temp ← (zero_ext8to32(rsrc1<15:8>) + zero_ext8to32(rsrc2<15:8>) + 1) / 2
  rdest<15:8> ← temp<7:0>
  temp ← (zero_ext8to32(rsrc1<23:16>) + zero_ext8to32(rsrc2<23:16>) + 1) / 2
  rdest<23:16> ← temp<7:0>
  temp ← (zero_ext8to32(rsrc1<31:24>) + zero_ext8to32(rsrc2<31:24>) + 1) / 2
  rdest<31:24> ← temp<7:0>
}
```

ATTRIBUTES

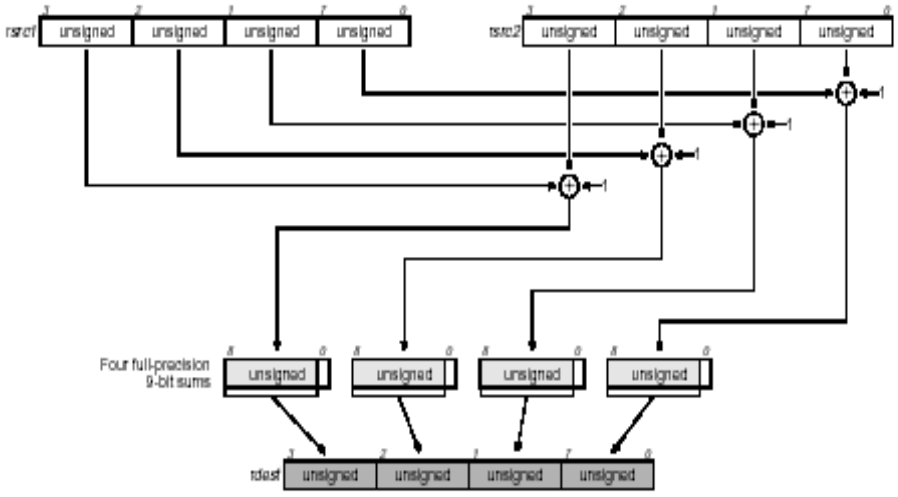
Function unit	dspalu
Operation code	73
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

iavgonep dspuquadaddui ifir8ii

DESCRIPTION

As shown below, the quadavg operation computes four separate averages of the four pairs of corresponding 8-bit bytes of *rsrc1* and *rsrc2*. All bytes are considered unsigned. The least-significant 8 bits of each average is written to the corresponding byte in *rdest*. No overflow or underflow detection is performed.



The quadavg operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x0201000e, r40 = 0xffffff02	`quadavg r30 r40 → r50`	r50 ← 0x81808008
r10 = 0, r60 = 0x9c9c6464, r70 = 0x649c649c	`IF r10 quadavg r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x9c9c6464, r70 = 0x649c649c	`IF r20 quadavg r60 r70 → r90`	r90 ← 0x809c6480

Signed sum of products of unsigned/signed bytes

ifir8ui

SYNTAX

```
[ IF rguard ] ifir8ui rsrc1 rsrc2 → rdest
```

FUNCTION

If *rguard* then

rdest ← zero_ext8to32(*rsrc1*<31:24>) × sign_ext8to32(*rsrc2*<31:24>) +
zero_ext8to32(*rsrc1*<23:16>) × sign_ext8to32(*rsrc2*<23:16>) +
zero_ext8to32(*rsrc1*<15:8>) × sign_ext8to32(*rsrc2*<15:8>) +
zero_ext8to32(*rsrc1*<7:0>) × sign_ext8to32(*rsrc2*<7:0>)

ATTRIBUTES

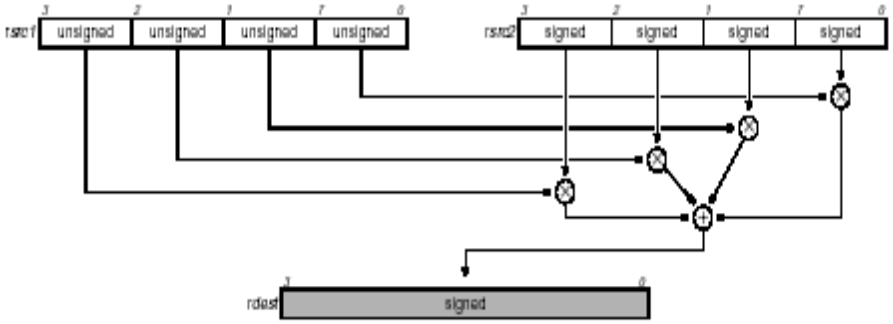
Function unit	dspmul
Operation code	91
Number of operands	2
Modifier	No
Modifier range	—
Latency	3
Issue slots	2, 3

SEE ALSO

ifir8ii ufir8uu ifir16 ufir16

DESCRIPTION

As shown below, the `ifir8ui` operation computes four separate products of the four pairs of corresponding 8-bit bytes of *rsrc1* and *rsrc2*; the four products are summed, and the result is written to *rdest*. The bytes from *rsrc1* are considered unsigned, but the bytes from *rsrc2* are considered signed; thus, the intermediate products and the final sum of products are signed. All computations are performed without loss of precision.



The `ifir8ui` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r70 = 0x0afb14f6, r30 = 0x0a0a1414	`ifir8ui r30 r70 → r90`	r90 ← 0xfa
r10 = 0, r70 = 0x0afb14f6, r30 = 0x0a0a1414	`IF r10 ifir8ui r30 r70 → r100`	no change, since guard is false
r20 = 1, r80 = 0x649c649c, r40 = 0x9c649c64	`IF r20 ifir8ui r40 r80 → r110`	r110 ← 0x2bc0
r50 = 0x80808080, r60 = 0xffffffff	`ifir8ui r60 r50 → r120`	r120 ← 0xfffe0200

Funnel-shift 1byte

funshift1

SYNTAX

```
[ IF rguard ] funshift1 rsrc1 rsrc2 → rdest
```

FUNCTION

If *rguard* then

rdest<31:8> ← rsrc1<23:0>

rdest<7:0> ← rsrc2<31:24>

ATTRIBUTES

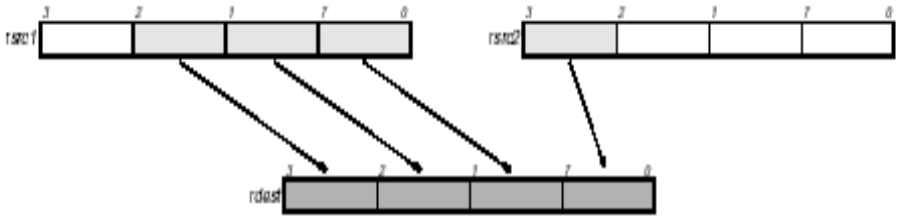
Function unit	shifter
Operation code	99
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

funshift2 funshift3 rol

DESCRIPTION

As shown below, the `funshift1` operation effectively shifts left by one byte the 64-bit concatenation of *rsrc1* and *rsrc2* and writes the most-significant 32 bits of the shifted result to *rdest*.



The `funshift1` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xaabbccdd, r40 = 0x11223344	`funshift1 r30 r40 → r50`	r50 ← 0xbbccdd11
r10 = 0, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r10 funshift1 r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r20 funshift1 r40 r30 → r70`	r70 ← 0x22334aa

funshift2

Funnel-shift 2 bytes

SYNTAX

```
[ IF rguard ] funshift2 rsrc1 rsrc2 → rdest
```

FUNCTION

If *rguard* then

rdest<31:16> ← *rsrc1*<15:0>

rdest<15:0> ← *rsrc2*<31:16>

ATTRIBUTES

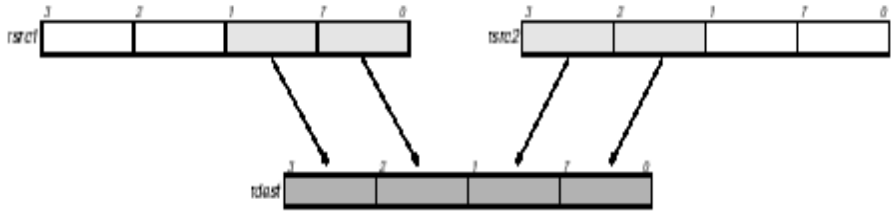
Function unit	shifter
Operation code	100
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

funshift1 funshift3 rol

DESCRIPTION

As shown below, the `funshift2` operation effectively shifts left by two bytes the 64-bit concatenation of *rsrc1* and *rsrc2* and writes the most-significant 32 bits of the shifted result to *rdest*.



The `funshift2` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xaabbccdd, r40 = 0x11223344	`funshift2 r30 r40 → r50`	r50 ← 0xccdd1122
r10 = 0, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r10 funshift2 r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r20 funshift2 r40 r30 → r70`	r70 ← 0x3344aabb

Funnel-shift 3 bytes

funshift3

SYNTAX

```
[ IF rguard ] funshift3 rsrc1 rsrc2 → rdest
```

FUNCTION

If *rguard* then
    rdest<31:24> ← rsrc1<7:0>
    rdest<23:0> ← rsrc2<31:8>

ATTRIBUTES

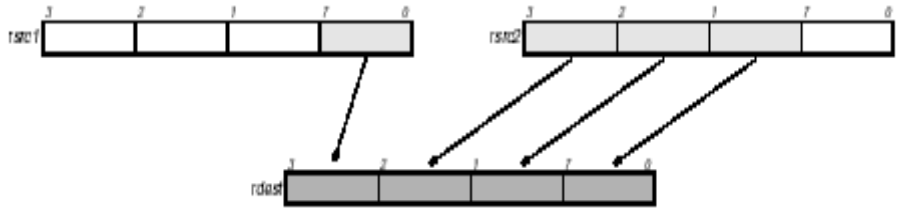
Function unit	shifter
Operation code	101
Number of operands	2
Modifier	No
Modifier range	—
Latency	1
Issue slots	1, 2

SEE ALSO

funshift1 funshift2 rol

DESCRIPTION

As shown below, the `funshift3` operation effectively shifts left by three bytes the 64-bit concatenation of *rsrc1* and *rsrc2* and writes the most-significant 32 bits of the shifted result to *rdest*.



The `funshift3` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0xaabbccdd, r40 = 0x11223344	`funshift3 r30 r40 → r50`	r50 ← 0xdd112233
r10 = 0, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r10 funshift3 r40 r30 → r60`	no change, since guard is false
r20 = 1, r40 = 0x11223344, r30 = 0xaabbccdd	`IF r20 funshift3 r40 r30 → r70`	r70 ← 0x44aabbcc

uclipi

Clip signed to unsigned

SYNTAX

```
[ IF rguard ] uclipi rsrc1 rsrc2 → rdest
```

FUNCTION

If *rguard* then

rdest ← min(max(*rsrc1*, 0), *rsrc2*)

ATTRIBUTES

Function unit	dspalu
Operation code	75
Number of operands	2
Modifier	No
Modifier range	—
Latency	2
Issue slots	1, 3

SEE ALSO

iclipi uclipu imin imax

DESCRIPTION

The `uclipi` operation returns the value of *rsrc1* clipped into the unsigned integer range 0 to *rsrc2*, inclusive. The argument *rsrc1* is considered a signed integer; *rsrc2* is considered an unsigned integer.

The `uclipi` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* is written; otherwise, *rdest* is not changed.

EXAMPLES

Initial Values	Operation	Result
r30 = 0x80, r40 = 0x7f	`uclipi r30 r40 → r50`	r50 ← 0x7f
r10 = 0, r60 = 0x12345678, r70 = 0xabc	`IF r10 uclipi r60 r70 → r80`	no change, since guard is false
r20 = 1, r60 = 0x12345678, r70 = 0xabc	`IF r20 uclipi r60 r70 → r90`	r90 ← 0xabc
r100 = 0x80000000, r110 = 0x3ffffff	`uclipi r100 r110 → r120`	r120 ← 0

ufloatrz

Convert unsigned integer to floating-point with rounding toward zero

SYNTAX

```
[ IF rguard ] ufloatrz rsrc1 → rdest
```

FUNCTION

```
If rguard then {
  rdest ← (float) ((unsigned long)rsrc1)
}
```

ATTRIBUTES

Function unit	falu
Operation code	119
Number of operands	1
Modifier	No
Modifier range	—
Latency	3
Issue slots	1, 4

SEE ALSO

ifloatrz ifloat ufloat
ifixieee ufloatflags

DESCRIPTION

The `ufloatrz` operation converts the unsigned integer value in *rsrc1* to single-precision IEEE floating-point format and writes the result into *rdest*. Rounding is performed toward zero; the IEEE rounding mode bits in PCSW are ignored. This is the preferred rounding mode for ANSI C. If `ufloatrz` causes an IEEE exception, such as inexact, the corresponding exception flags in the PCSW are set. The PCSW exception flags are sticky: the flags can be set as a side-effect of any floating-point operation but can only be reset by an explicit `writepcsw` operation. The update of the PCSW exception flags occurs at the same time as *rdest* is written. If any other floating-point compute operations update the PCSW at the same time, the net result in each exception flag is the logical OR of all simultaneous updates ORed with the existing PCSW value for that exception flag.

The `ufloatrzflags` operation computes the exception flags that would result from an individual `ufloatrz`.

The `ufloatrz` operation optionally takes a guard, specified in *rguard*. If a guard is present, its LSB controls the modification of the destination register. If the LSB of *rguard* is 1, *rdest* and the exception flags in PCSW are written; otherwise, *rdest* is not changed and the operation does not affect the exception flags in PCSW.

EXAMPLES

Initial Values	Operation	Result
r30 = 3	`ufloatrz r30 → r100`	r100 ← 0x40400000 (3.0)
r40 = 0xffffffff (4294967295)	`ufloatrz r40 → r105`	r105 ← 0x4f7fffff (4.294967040e+9), INX flag set
r10 = 0, r50 = 0xffffffd	`IF r10 ufloatrz r50 → r110`	no change, since guard is false
r20 = 1, r50 = 0xffffffd	`IF r20 ufloatrz r50 → r115`	r115 ← 0x4f7fffff (4.294967040e+9), INX flag set
r60 = 0x7fffffff (2147483647)	`ufloatrz r60 → r117`	r117 ← 0x4effffff (2.147483520e+9), INX flag set
r70 = 0x80000000 (2147483648)	`ufloatrz r70 → r120`	r120 ← 0x4f000000 (2.147483648e+9)
r80 = 0x7fffffff1 (2147483633)	`ufloatrz r80 → r122`	r122 ← 0x4effffff (2.147483520e+9), INX flag set