Accelerating (MPEG-4) Video Decoding with an Xtensa Processor

Application Note
Contents

1 Introduction ................................................................................................................ 1

2 Overview of the MPEG-4 Standard ............................................................................. 2

3 MoMuSys MPEG-4 Video Decoder ............................................................................. 3
   The Base MoMuSys Decoder ....................................................................................... 4
   Test Streams .................................................................................................................. 4
   Out-of-the-Box Decoder Performance ......................................................................... 4

4 SIMD TIE Engine ......................................................................................................... 7

5 Optimizing an MPEG-4 Simple Profile Decoder .......................................................... 8
   Motion Compensation .................................................................................................. 9
      General Motion Compensation ................................................................................. 9
      Half Pel Motion Compensation ............................................................................. 11
      Unrestricted Motion Compensation ..................................................................... 13
      Performance of Optimized Motion Compensation ............................................... 14
   Inverse Discrete Cosine Transform (IDCT) ................................................................. 15
   Bitstream Processing ................................................................................................. 17
   Variable Length Decoding ......................................................................................... 18

6 Putting It All Together: An Optimized MPEG-4 Decoder ............................................. 21
   Other Optimizations .................................................................................................. 21
      Dequantization ....................................................................................................... 21
      AC/DC Prediction ................................................................................................. 22
      Color Conversion .................................................................................................... 22
      Post-Filtering ......................................................................................................... 22
      Memory Hierarchy ................................................................................................. 22
   Synthesis Results for the Core and TIE ..................................................................... 23
   Performance Results for Optimized MoMuSys Video Decoder ..................................... 24
   Unimplemented Optimizations .................................................................................... 25

7 Conclusions .................................................................................................................. 25

Appendix A – TIE Code for Bitstream Processing and Variable Length Decoding.............. 26
Figures

Figure 1: Element numbering for the TIE SEL instruction.................................................................8
Figure 2: TIE optimization for MC for (Address mod 8) = 5. ..........................................................11
Figure 3: Half pixel motion compensation.........................................................................................11
Figure 4: Example of Unrestricted motion compensation (full pel)......................................................13
Figure 5: Transpose of a 3x3 matrix....................................................................................................16
Figure 6: Overview of TIE for accelerating Bitstream Processing.....................................................18
Figure 7: Matrix multiplication used for color conversion ...................................................................22

Tables

Table 1: Description of the Test Streams used....................................................................................4
Table 2: Out-of-the-box performance for a base Xtensa configuration with 4K Icache and 4K Dcache. .....6
Table 3: Out-of-the-box performance for Xtensa processor configuration with 16x16 multiplier............7
Table 4: Optimized Motion Compensation results versus original .....................................................15
Table 5: Optimized iDCT performance versus original iDCT with zero checking...............................17
Table 6: Optimization results for Bitstream Processing .....................................................................18
Table 7: Optimized Variable Length Decoding versus original ...........................................................21
Table 8: Optimized MPEG-4 video decoder results .........................................................................24
Abstract

This document describes how video decoding, in general, and MPEG-4 video decoding, in particular, can be accelerated using an Xtensa processor with customer-defined instructions. Xtensa-based application-specific processors are quickly designed and verified. These processors offer performance that rivals pure hardware solutions along with the benefits of flexibility, programmability and ease of verification found with pure software implementations.

In this application note, we show how high-performance implementations can be made for the Inverse Discrete Cosine Transform, Variable length decoding and Motion Compensation using instructions developed with the Tensilica Instruction Extensions (TIE) language. These optimizations are combined to show that the MoMuSys MPEG-4 reference code can be optimized by a factor of 50. The original code required a processor speed of at least 400MHz while the optimized code only requires 9MHz, without writing any assembly code. These optimizations were done in 7 man-months producing a fully verified optimized processor core and software. More optimizations are still possible due to inefficiencies in the original code.

This application note assumes that the reader is familiar with the Tensilica design methodology and Tensilica Instruction Extension (TIE) language. Also, a basic knowledge on video coding is assumed.
Accelerating (MPEG-4) Video Decoding with an Xtensa Processor

1 Introduction

Digital video compression has become an important feature for a wide variety of products. Designing these products that require support for multiple video standards can be challenging. Currently, the main video compression standards are MPEG-1/2, H.263 and MPEG-4. In this document, we focus on MPEG-4 video coding because it is a superset of the techniques defined in the other standards.

A major challenge for system designers is to create a high-performance but flexible architecture that is capable of supporting the large variety of marketing requirements such as different audio and video coding standards and multiple communication protocols. The problem is not only in the design of a solution for any one feature, but in creating a product possessing most or all of the features in a reasonable amount of time. One solution is to use hardware blocks, if they are available. This solution is often not an option for multi-feature products due to the large number of hardware blocks required to support all the features or simply due to ever-changing feature requirements. A solution is required that provides the performance but also the flexibility and fast time to market that is required by modern-day products.

Fixed form programmable processors are commonly available from a number of sources. Most of these processors are capable of handling some level of real-time video decoding. But more often than not, these solutions will use most of the available processor bandwidth. To reduce the processor load by the video processing so that larger resolutions can be supported or to allow other tasks to run on the same processor as well, other solutions must be sought.

This application note shows the performance that can be achieved by extending the core instructions with general SIMD instructions for accelerating video decompression. The advantage of this method is that these SIMD instructions could be used for optimizing other functions and algorithms as well. The focus of this application note is not to demonstrate the best implementation possible using an Xtensa processor, but to show how different parts of video decompression can be accelerated with an Xtensa core and TIE instructions.

This project used the publicly available MoMuSys MPEG-4 video decoder as a reference, which can be downloaded from www.osi.org. This decoder is able to decode the complete visual standard, which is divided in multiple profiles and levels. Due to time constraints this application note focuses on the Simple Profile. Using a reference implementation as a base has a couple of disadvantages:

♦ In general, reference code is written for understandability, not for efficiency.
♦ The reference code can handle any MPEG-4 Natural Video stream and even multiple streams, whereas the Simple Profile is a small subset of MPEG-4 Natural Video coding. Therefore, a large overhead is caused by the support of all MPEG-4 video features.

This application note will show that even these deficiencies may actually highlight the advantages of using a configurable, easily extendable and verifiable core.

Although the project assumes that the whole decoder will actually run on the processor, in some systems this might not be the case. But again the project will show the power that is available to the user to accelerate the parts that will run on a processor core. Plus, it shows that the whole decoder can be run on a single or on multiple Xtensa processors.

Rather than focusing on area and power considerations in this application note, the effort is to show how Tensilica Instruction Extensions (TIE) can be used to accelerate the major parts of video decoding. These optimizations can be used to generate very efficient implementations of higher resolution video decoders (VGA resolution). Using the optimizations shown in this document, a very large design space can be explored by system designers.

This application note has seven sections. The following section explains some basic terminology and provides an overview of the MPEG-4 standard. Section 3 contains information about the MoMuSys video decoder and the test streams that are used throughout this document. In section 4, you will find a high level overview of the TIE SIMD engine that is the basis for the rest of the optimizations. Section 5 presents a technical description of the optimization techniques.
for the main parts of most video decoders. Other optimizations are shown in section 6. This section also combines all optimizations and shows the performance of the resulting optimized MPEG-4 video decoder. The last section contains the conclusions.

2 Overview of the MPEG-4 Standard

In this section, we will give a short summary of the history of MPEG-4 and a high level overview of the MPEG-4 standard. After this we will discuss how video coding works in general and introduce and explain some relevant terminology.

The Motion Picture Experts Group (MPEG) is a standardization group that focuses on coding techniques for moving pictures. MPEG-4 is the third standard produced by this group. The first, MPEG-1, was a standard for coding of VCR quality video and audio on CD’s with a maximum bitrate of about 1.5Mbit/s. The next standard, MPEG-2, extends the MPEG-1 standard to cover coding of general video and audio. The MPEG-2 standard is used as the coding scheme in DVD and Digital Broadcasting. It adds interlaced image support and scalability to the MPEG-1 video standard. An MPEG-3 standard was planned for handling High Definition TV (HDTV) coding, but this standard was canceled when people realized that MPEG-2 was more than capable of handling HDTV coding.

The MPEG-4 standard was focused at first on the coding of low bitrate video. This focus was later changed to a standardized coding for multimedia content. This is a significant difference from MPEG-1 and MPEG-2. The MPEG-1 and 2 standards describe how to decode Natural Audio/Video sequences. MPEG-4, on the other hand, describes how to decode, combine and interact with multimedia content (multiple streams, natural and synthetic content). It does not describe new methods for natural video coding, as this is only a subset of MPEG-4.

The difference between (natural) video coding and multimedia coding is that MPEG-1 and MPEG-2 assume that only natural video exists and that it covers the whole screen area. This is not the case for multimedia because the different screen regions are best coded using different coding methods. Plus, a very important part of multimedia is interactivity. MPEG-4 should therefore be seen as a collection of coding schemes that are combined to generate multimedia streams.

MPEG-4 consists of the following coding methods (called tools):

- **Natural Video:**
  - MPEG-1/MPEG-2 video coding methods
  - H.263 video coding methods
  - Advanced video coding (AVC) (Same as H.264)

- **Synthetic Video:**
  - Face and body animation
  - 2D/3D animation

- **Natural Audio:**
  - Voice coding
  - AAC audio coding

- **Synthetic Audio:**
  - Synthetic music
  - Text to speech

In addition to the above-mentioned coding methods, MPEG-4 contains an extensive section that describes how to combine multiple objects into one image. For more details on the complete MPEG-4 standard, see the MPEG-4 homepage (http://mpeg.telecomitalialab.com/).
In this document, we will focus on natural video coding because it is the most mature tool in the MPEG-4 toolset. Another reason is that this is the main area of interest for commercialization.

MPEG-4 natural video uses the compression techniques found in H.263, MPEG-1/2. These standards compress video by using three different techniques:

♦ Color frame decimation
♦ Spatial compression
♦ Temporal compression

The input to the encoder is a stream of images where the image is represented in YCbCr format. A pixel in the YCbCr color space can be subdivided in three pels where each pel represents one of the color components. In the RGB color space, the three pels represent Red, Green and Blue. The advantage of using an YCbCr color space is that eyes are more sensitive to luminance (Y) than to color information (Cb and Cr). Therefore, video can be compressed by reducing the amount of color information. The encoder splits the image into three frames: Y, Cb and Cr. Now, compression can be achieved by reducing the size of the Cb and Cr frames. MPEG-4 uses a 4:2:0 colorspace, which means that the Cb and Cr frames are a quarter of the size of a Y frame. This produces a compression of a factor of two. Although this technique, color frame decimation, is a very effective compression method, it is not always used.

The second compression method, spatial compression, is always used. Spatial compression is based on the fact that neighboring pixels will have related pixel values. This attribute of natural images enables us to compress data by taking a small region of a frame and by viewing this region as a two-dimensional waveform. For a small region the high frequency components can be zero or will be very small. MPEG-4 uses 8x8 pel regions that are transformed with the discrete cosine transform (DCT). This result may already require less coefficients than pixel values but the real compression is achieved by dividing the frequency coefficients by a quantizer value. The resulting block will contain a lot of zeros, especially in the higher frequencies. The presence of these zeros can then be utilized by zig-zag scanning and run-length encoding. For a more detailed description of these algorithms, see the different video standards or textbooks on video coding.

The last compression technique is based on the knowledge that we are compressing natural video. The first two techniques are based on techniques found in still image compression standards such as JPEG. The third technique is specific to moving pictures; there is a dependency between the consecutive images. Temporal compression uses this dependency by describing the motion (with a motion vector) of a region, instead of the DCT information of this region. The video decoder receives the motion vector that tells the decoder where it can find this region in the previous frame. This process is called motion compensation. If necessary, the error between that block and the original values can be encoded using DCT coding. Temporal compression cannot be used on all frames because there must be frames that do not depend on any other frames. Therefore, two different frame types are used. INTRA frames which uses color decimation and spatial compression but do not use temporal compression. The second type is INTER which uses all three compression techniques.

3 MoMuSys MPEG-4 Video Decoder

The MPEG committee used two software implementations of the standard to prevent ambiguities in the standard and to validate proposals made during the standardization process. Both of these verification models are downloadable from the OSI website.

In this application note, the verification model generated by the Mobile Multimedia Systems (MoMuSys) project was used. This decoder is written in ANSI C and it can be compiled with the GNU compiler. Thus, it can be easily compiled by replacing the GNU compiler by the Xtensa GNU compiler or by the Xtensa C-compiler (XCC).

A number of test streams were used to validate the correctness of the optimizations. These test streams are described in the next section. After which a short overview of the MoMuSys MPEG-4
Accelerating (MPEG-4) Video Decoding with an Xtensa Processor

A decoder structure will be given. After compiling the code for an Xtensa processor, we can run the decoder on the test streams. The resulting cycle counts are discussed in the last section. In this section we will also show how configuring the Xtensa core can improve performance.

The Base MoMuSys Decoder

After running an initial simulation of the decoder, it became clear that the MoMuSys decoder uses a floating-point implementation for the iDCT. Using a floating-point implementation is not required. Ideally, any 8x8 block that is transformed through a DCT and through an iDCT should produce exactly the same block. Minor differences are possible, however, for almost any implementation of the iDCT. To allow for different implementations of the iDCT and at the same time guarantee some quality level, all iDCT implementations should comply with the precision requirements set forward in the IEEE 1180-1990 standard.

For optimization reasons, it is better to have an integer implementation of the iDCT because the rest of the decoder uses integer operations. The floating point iDCT is therefore replaced by the iDCT implementation used in the IJG JPEG decoder. This implementation is based on an algorithm described in C.Loeffler, A.Ligtenberg and G.Moschytz. “Practical Fast 1-D DCT algorithms with 11 Multiplications”\(^1\) The implementation uses 12 multiplies and 32 adds. This implementation was verified to satisfy the minimum precision requirements set forth by the IEEE 1180-1990 standard. Then I verified that the difference with the floating-point implementation between two values is less or equal to 1.

In the rest of this document, we assume that this version of the MoMuSys code with the integer iDCT is the ‘Original’ MoMuSys code.

Test Streams

Table 1 describes the different test streams that were used for verifying the correctness of the decoder and to quantify its performance. The streams were generated by the MoMuSys encoder using no rate control and no skipped frames.

<table>
<thead>
<tr>
<th>Number of frames</th>
<th>Stream Size (KByte)</th>
<th>Avg Bitrate (15fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss America</td>
<td>149</td>
<td>66</td>
</tr>
<tr>
<td>Suzie</td>
<td>149</td>
<td>129</td>
</tr>
<tr>
<td>Foreman</td>
<td>400</td>
<td>724</td>
</tr>
<tr>
<td>Car Phone</td>
<td>381</td>
<td>606</td>
</tr>
<tr>
<td>Monsters Inc</td>
<td>1438</td>
<td>1456</td>
</tr>
</tbody>
</table>

Out-of-the-Box Decoder Performance

To obtain a first estimate of the cycle requirements for the MPEG-4 decoder, a base big endian Xtensa processor configuration is used with 4KB Instruction cache and 4KB Data cache. The

---

decoder source code can then be compiled after building and installing the Xtensa processor configuration.

After the code is compiled, the simulation can be run on the Xtensa Instruction Set Simulator (ISS). Shown below is the command line command for the ISS and the output of the ISS. The ISS performs full pipeline modeling and memory system modeling. It also generates profile information in a gmon.out file.

```
> xt-run -pipe -mem_model -profile=gmon.out vm_dec-xt Suzie_QCIF_dec.ctl
Complexity Estimation DISABLED
Using the neither Data Partitioning nor RVLC.
ACDC Prediction ENABLED
display_time=0,prev_time=-1,vop_time=-1
Start I VOP decoding
display_time=0,prev_time=-1000,vop_time=0
Start P VOP decoding
==============================================================
[1]{curr_time}:0.000000 ->{next_time} = 66.666664
display_time=1,prev_time=0,vop_time=1
Start P VOP decoding
==============================================================
[2]{curr_time}:66.666664 ->{next_time} = 133.333328
display_time=2,prev_time=1,vop_time=2
Start P VOP decoding
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : :
: : : }
Time for Simulation = 22334.35 seconds (user = 22334.35 system = 0.00)
current pc = 0x6006e416

Cache Configuration:
  I cache: 4096 bytes (4KB), direct-mapped, 16-byte line
  icache allocation penalty  6 + system time
  D cache: 4096 bytes (4KB), direct-mapped, 16-byte line
  load   allocation penalty  7 + system time

Events Number Number
            per 100
Instructions 2827121053 ( 100.00 )
Instruction fetches 2297584800 (  81.27 )
    Uncached 954 ( 0.00 )
    CACHED with access mode 1 2297583846 (  81.27 )
Icache misses 27880058 (  0.99 ) 1.21% of ic reads
Unconditional taken branches 82229531 (  2.91 )
Conditional branches 313153294 ( 11.08 )
    Taken 249736916 (  8.33 )
    Not taken 63416378 (  2.24 )
Window Overflows 866157 (  0.03 )
Window Underflows 866155 (  0.03 )
Loads 496049671 ( 17.55 )
    Uncached 10 ( 0.00 )
    CACHED with access mode 1 496049661 ( 17.55 )
Dcache Load misses 59175070 (  2.09 ) 11.93% of dc reads
Stores 330290944 ( 11.68 )
    CACHED with access mode 1 330290944 ( 11.68 )
Dcache Store misses 195227271 (  6.91 ) 59.11% of dc writes
Accelerating (MPEG-4) Video Decoding with an Xtensa Processor

The first part of the simulator output until “Time for Simulation” is the output of the decoder software. The rest is the simulator giving a summary of the simulation run itself. The most important line of the simulation output is the total processor cycles. The above run for the Suzie test stream required 44.469 Giga-cycles. An explanation of the rest of the simulation output can be found in the Xtensa Instruction Set Simulator manual. Table 2 shows the cycle requirements of all test streams for the base Xtensa configuration. The last column of the table reports the average cycle requirement (in millions) per second. This number is not the same as the required clock frequency because some frames require more cycles than this average. A real decoder must be able to decode a stream consisting of only these worst-case frames. This number can be used as an indication of the clock frequency, however.

**TABLE 2: OUT-OF-THE-BOX PERFORMANCE FOR A BASE XTENSA CONFIGURATION WITH 4K ICache AND 4K DCache.**

<table>
<thead>
<tr>
<th>Number of frames</th>
<th>Total cycle count (10^9)</th>
<th>Million Cycles per second (average)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss America</td>
<td>149</td>
<td>4.11</td>
</tr>
<tr>
<td>Suzie</td>
<td>149</td>
<td>4.45</td>
</tr>
<tr>
<td>Foreman</td>
<td>400</td>
<td>13.13</td>
</tr>
<tr>
<td>Car Phone</td>
<td>381</td>
<td>12.06</td>
</tr>
<tr>
<td>Monsters Inc²</td>
<td>1437</td>
<td>40.53</td>
</tr>
</tbody>
</table>

Because profile information has been gathered in the ISS run, we can look at the execution profile of the MoMuSys decoder. Shown below are the top functions for decoding the Suzie test stream.

**Flat profile:**

<table>
<thead>
<tr>
<th>% cumulative cycles</th>
<th>self cycles</th>
<th>self calls</th>
<th>total cycles</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(M)</td>
<td>(M)</td>
<td>(M)</td>
<td></td>
</tr>
<tr>
<td>15.10</td>
<td>671.62</td>
<td>58600</td>
<td>1.01</td>
<td>GetPred_Advanced</td>
</tr>
<tr>
<td>12.58</td>
<td>1231.00</td>
<td>559.38</td>
<td>318.94</td>
<td>__mulsi3</td>
</tr>
<tr>
<td>7.17</td>
<td>1549.94</td>
<td>384.66</td>
<td>11158</td>
<td>Blend</td>
</tr>
<tr>
<td>6.40</td>
<td>1834.60</td>
<td>284.66</td>
<td>11158</td>
<td>GetMBblockdata</td>
</tr>
<tr>
<td>4.90</td>
<td>2052.67</td>
<td>211.53</td>
<td>11158</td>
<td>InterpolateImage</td>
</tr>
<tr>
<td>4.76</td>
<td>2264.20</td>
<td>148</td>
<td>11158</td>
<td>1.43</td>
</tr>
<tr>
<td>3.88</td>
<td>2436.53</td>
<td>592</td>
<td>284.66</td>
<td>ClipImageI</td>
</tr>
<tr>
<td>3.75</td>
<td>2603.40</td>
<td>66948</td>
<td>211.53</td>
<td>BlockIDCT</td>
</tr>
</tbody>
</table>

² The Monsters Inc trailer stream is 18 frames per second instead of 15
The `mulsi` function is a library function that performs integer multiplication. This function is inserted in the original code by the C compiler every time the C multiply operator is used. We can therefore optimize the performance of the code by changing our configuration to include a 16x16 multiplier. The decoder performance for this processor configuration is shown in Table 3.

**TABLE 3: OUT-OF-THE-BOX PERFORMANCE FOR XTENSA PROCESSOR CONFIGURATION WITH 16X16 MULTIPLIER**

<table>
<thead>
<tr>
<th>Number of frames</th>
<th>Total cycle count (10^9)</th>
<th>Million cycles per second (average)</th>
<th>Performance improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss America</td>
<td>149</td>
<td>3.13</td>
<td>314.72</td>
</tr>
<tr>
<td>Suzie</td>
<td>149</td>
<td>3.39</td>
<td>341.21</td>
</tr>
<tr>
<td>Foreman</td>
<td>400</td>
<td>10.05</td>
<td>376.71</td>
</tr>
<tr>
<td>Car Phone</td>
<td>381</td>
<td>9.22</td>
<td>363.09</td>
</tr>
<tr>
<td>Monsters Inc³</td>
<td>1437</td>
<td>29.33</td>
<td>367.35</td>
</tr>
</tbody>
</table>

Using this processor configuration gives almost a 25% improvement over the previous configuration. We will, therefore, use this configuration as the base processor configuration for the rest of this application note.

### 4 SIMD TIE Engine

Two main optimization methods exist. The first one combines multiple operations into one TIE instruction and the second performs an operation on multiple data elements. A combination of these two methods can also be used. For video, the most promising method is to use the second approach. This approach is also called Single Instruction Multiple Data (SIMD). This approach promises the most performance because video works on blocks of data and the operations on these blocks of data are the same.

Let’s look at motion compensation for a moment. The implementation of motion compensation loads 8 elements from the reference frame and writes them into the current frame. SIMD TIE instructions could achieve this task in 2 instructions by using an 8-element wide load/store path. All other parts except bitstream processing can be optimized in a similar way.

A general SIMD engine will therefore be used as the base TIE engine for the optimizations. The SIMD engine can work on eight 16-bit data elements or four 32-bit data elements. The engine contains a 16-entry register set of 128 bits. Multiple datatypes are defined for this register set:

- vec8x16 //representing 8 elements of 16 bits. 128 bit aligned
- vec8x8 //representing 8 elements of 8 bits. 64 bit aligned
- vec4x32 //representing 4 elements of 32 bits. 128 bit aligned

These datatypes are loaded by load instructions that load 128 or 64 bit from memory into a 128-bit register. The 64 bits are zero extended.

The following SIMD instructions are available:

- Arithmetic/Logic instructions, i.e. Addition, subtraction, arithmetic/logic shift and logic operations
- Load/Store instructions. Capable of loading 128 bits. From an address formed by a register and an immediate or two registers. Versions using auto-update addressing are also available

³ The Monsters Inc trailer stream is 18 frames per second instead of 15
Accelerating (MPEG-4) Video Decoding with an Xtensa Processor

- Multiply/accumulate instructions. Four 16x16 multipliers/addition units are available. Two multiply instructions are needed to multiply 8 elements. Each instruction produces 4 32-bit elements.
- Compare instructions and conditional move instructions.
- Select instruction. Explained below.

The **select** instruction mentioned above requires some explanation. The **select** instruction is capable of generating a new vector out of the elements of the two source vectors. Any 16 bit element of the two source vectors can be assigned to each of the 8 elements of the result vector.

![Figure 1: Element numbering for the TIE SEL instruction](image)

The select instruction has three inputs: two input vectors (of type vec8x16) and a selector (of type integer). The 32-bit selector specifier contains 8 fields of 4 bits, each field indicates which input element will be put there as shown in Figure 1. Below is an example of using the select instruction to extract the odd elements from the two input vectors. Please note that the Xtensa processor configuration used is a big endian processor.

```c
int sel0=0x13579BDF
vec8x16 v1, v2, result_vector;
short tmp1[8]= {0, 1, 2, 3, 4, 5, 6, 7}
short tmp2[8]= {8, 9, 10, 11, 12, 13, 14, 15}
Result_vector = SEL(v1, v2, sel0)
=> Result_vector equals {1, 3, 5, 7, 9, 11, 13, 15}
```

5 Optimizing an MPEG-4 Simple Profile Decoder

Now that we have a TIE engine and a compiled and runnable program, we can start optimizing the decoder. We will use the execution profile obtained from running the Suzie test stream. In this profile, shown below, only the top functions are shown for simplicity. The top function performs motion compensation, followed by the function that decodes a whole INTER frame. The third function performs the integer iDCT computation.
In the next sections, we describe how the different functions are optimized, starting with motion compensation, followed by IDCT, Bitstream processing and finally Variable-length decoding.

**Motion Compensation**

**General Motion Compensation**

Motion Compensation is the process that retrieves the content of the block from the location in the reference frame that is formed by the current block position and the motion vector. Pseudo code for the simple motion compensation case is shown below. We will use this pseudo code to show the principle of the optimizations used in the MPEG-4 decoder.

```c
Function Motion_Compensation(
    char *Ref, // Pointer to previous frame
    int x,   // x position of current block
    int y,   // y position of current block
    int MV_x, // x component of motion vector
    int MV_y, // y component of motion vector
    int width, // Width of the frame
    char *block// pointer to location of current block )
{
    Int a, b
    Char *Inptr
    Char *outptr

    Inptr = *(Ref + (y + MV_y)* width + x + MV_x)
    Outptr = block
    For (a=0;a<8;a++)
    For (b=0;b<8;b++)
        *(Outptr+b) = *(Inptr+b)
    Inptr +=width
}
```

---

**Flat profile:**

<table>
<thead>
<tr>
<th>Flat profile:</th>
<th>cumulative cycles</th>
<th>self cycles</th>
<th>calls</th>
<th>self cycles</th>
<th>cumulative cycles</th>
<th>total cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>(M)</td>
<td>(M)</td>
<td>(M)</td>
<td>(M)</td>
<td>(M)</td>
<td>(M)</td>
</tr>
<tr>
<td>17.10</td>
<td>580.63</td>
<td>580.63</td>
<td>58600</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>6.73</td>
<td>809.31</td>
<td>228.68</td>
<td>148</td>
<td>1.55</td>
<td>18.29</td>
<td>DecodeVopCombinedMotionShapeTextureInter</td>
</tr>
<tr>
<td>6.48</td>
<td>1029.33</td>
<td>162.66</td>
<td>592</td>
<td>0.27</td>
<td>0.27</td>
<td>BlockIDCT</td>
</tr>
<tr>
<td>5.98</td>
<td>1232.32</td>
<td>142.99</td>
<td>148</td>
<td>1.37</td>
<td>4.67</td>
<td>InterpolateImage</td>
</tr>
<tr>
<td>4.79</td>
<td>1394.98</td>
<td>142.99</td>
<td>592</td>
<td>0.27</td>
<td>0.27</td>
<td>ClipImageI</td>
</tr>
<tr>
<td>4.49</td>
<td>1547.55</td>
<td>142.99</td>
<td>447</td>
<td>0.34</td>
<td>0.34</td>
<td>Blend</td>
</tr>
<tr>
<td>4.29</td>
<td>1693.15</td>
<td>142.99</td>
<td>11158</td>
<td>0.01</td>
<td>0.01</td>
<td>GetMBblockdata</td>
</tr>
<tr>
<td>4.16</td>
<td>1834.52</td>
<td>142.99</td>
<td>1636</td>
<td>0.09</td>
<td>0.09</td>
<td>unrestricted_MC</td>
</tr>
<tr>
<td>3.52</td>
<td>1953.98</td>
<td>142.99</td>
<td>1350</td>
<td>0.09</td>
<td>0.09</td>
<td>SetConstantImageI</td>
</tr>
<tr>
<td>3.24</td>
<td>2063.96</td>
<td>142.99</td>
<td>592</td>
<td>0.19</td>
<td>0.19</td>
<td>CopyImageI</td>
</tr>
<tr>
<td>3.20</td>
<td>2172.56</td>
<td>142.99</td>
<td>13263839</td>
<td>0.00</td>
<td>0.00</td>
<td>__mulsi3</td>
</tr>
<tr>
<td>3.19</td>
<td>2280.85</td>
<td>142.99</td>
<td>10859</td>
<td>0.00</td>
<td>0.00</td>
<td>BlockDequantH263</td>
</tr>
<tr>
<td>3.13</td>
<td>2387.01</td>
<td>142.99</td>
<td>3750400</td>
<td>0.00</td>
<td>0.00</td>
<td>unrestricted_MC</td>
</tr>
<tr>
<td>2.90</td>
<td>2485.48</td>
<td>98.47</td>
<td>745</td>
<td>0.13</td>
<td>0.13</td>
<td>MakeImageEdge</td>
</tr>
</tbody>
</table>
```
This pseudo code can be optimized, in principle, by using wide TIE loads. These wide loads could load all 8 pels at once, although this optimization creates an alignment problem. An Xtensa processor can only load aligned data, which means that a 32-bit data element must have an address that is a multiple of 4, as an Xtensa processor addresses bytes. The problem caused by using SIMD loads for motion compensation is that motion compensation accesses memory on a one byte (pel) boundary. Loading 64-bit (vectors of 8 pels) imposes an alignment requirement of 64 bits, which is higher than the 8-bit alignment needed by the scalar implementation.

Performing two aligned SIMD loads will solve the alignment problem because this loads 16 aligned pels from which any eight sequential pels can be extracted. Figure 2 shows how to extract the correct pels from two loads where (Address mod 8) = 5. Using this technique, we can now optimize the original pseudo code shown above. The resulting pseudo code is shown below.

```c
Function Motion_Compensation(
    char *Ref, // Pointer to previous frame
    int x,  // x position of current block
    int y,  // y position of current block
    int MV_x, // x component of motion vector
    int MV_y, // y component of motion vector
    int width, // Width of the frame
    char *block// pointer to location of current block
) {
    vec8x8 TmpVec,
    vec8x8 *Inptr
    Int a, b
    Int sel_val
    Int p0 = 0x01234567 // We are using a big endian processor
    Int p1 = 0x89abcdef // configuration
    Addr    = (Ref + (y + MV_y)* width + x + MV_x)
    Inptr   = Addr&0xffffffff // Align address by clearing lower 3 bits
    tmp    = Addr&0x7        // save lower 3 bits
    tmp     = 32 – (tmp<<2)
    sel_val = Funnel_Shift(p0, p1, tmp) // Create correct selector value
    // for sel instruction based on
    // lower 3 address bits
    For (a=0;a<8;a++) {
        TmpVec1 = LV64_I(Inptr, 0) // read first aligned vector
        TmpVec2 = LV64_X(Inptr, 8) // read second aligned vector
        Inptr+= width    // Set read pointer to next row
        SEL(TmpVec1, TmpVec2, sel_val) // Select correct 8 pels out 2 vec
        SV64_IU(TmpVec1, block, 8) // Store MC pels into block and
        // auto update address
    }
}
```
The original code for the motion compensation required about 272 instructions to execute the two for loops. In the optimized case 40 instructions are executed for the one loop. This is faster by almost a factor of 7.

Now that we have explained the basic optimization method used for motion compensation, we should discuss the complications caused by:

- Half Pel motion compensation
- Unrestricted Motion Compensation

### Half Pel Motion Compensation

Half pel motion compensation creates problems because the pel value is interpolated from two or even 4 pel values. This is shown in Figure 3. \( \text{fac} \) in this figure refers to a configurable rounding factor. MPEG-4 Simple Profile does not support quarter pel motion compensation. The solution for half pel motion compensation can, however, also be used for quarter pel motion compensation.

Let’s first look at the simpler case of half pixel motion compensation in the vertical direction. First, we need to load all the data. This task will require 9 sets of TIE SIMD loads. The
interpolation can now be performed on a row base by adding two rows together and doing the rounding and divide using general SIMD add and shift instructions. The pseudo code for performing half pixel in the vertical direction for one row is shown below.

```c
// Load top line
TmpVec1 = LV64_I(Inptr, 0)  // read first aligned vector
TmpVec2 = LV64_I(Inptr, 8)  // read second aligned vector
Inptr  += width           // Set read pointer to next row
SEL(TmpVec1, TmpVec2, sel_val)  // Select correct 8 pels out 2 vec
TopVec  = TmpVec1

// Load bottom line
TmpVec1 = LV64_I(Inptr, 0)  // read first aligned vector
TmpVec2 = LV64_I(Inptr, 8)  // read second aligned vector
Inptr  += width           // Set read pointer to next row
SEL(TmpVec1, TmpVec2, sel_val)  // Select correct 8 pels out 2 vec

// Calculate and store vertically interpolated vector
BotVec  = Add16(TopVec, TmpVec1)// Add bottom and top vectors
TmpVec1 = Add16(BotVec, fac) // Add 1+rounding vector to total
TmpVec1 = SLL16_I(TmpVec1, 1) // Divide by 2
SV64_IU(TmpVec1, block, 8) // Store MC pels into block and
                           // auto update address
```

Half pixel motion compensation in the horizontal direction is slightly more complex because 9 pels are needed to create the 8 result pels. Here, we can also use general SIMD instructions if we can generate two vectors containing pels 0 to 7 and 1 to 8. These vectors can be generated from the two aligned vectors by two SEL instructions. Below is the pseudo code for half pixel motion compensation in the horizontal direction with (Address mod 8)=5.

```c
sel_val1= 0x56789abc
sel_val2= 0x6789abcd
TmpVec1 = LV64_I(Inptr, 0)  // read first aligned vector
TmpVec2 = LV64_I(Inptr, 8)  // read second aligned vector
Inptr  += width           // Set read pointer to next row
Tmp    = TmpVec1
SEL(TmpVec1, TmpVec2, sel_val)  // Select pels 0 to 7
SEL(Tmp, TmpVec2, sel_val)  // Select pels 1 to 8
TmpVec1 = Add16(TmpVec1, Tmp) // Create horizontal interpolated
                            // vector
TmpVec1 = Add16(TmpVec1, fac) // Add 1+rounding vector to total
TmpVec1 = SLL16_I(TmpVec1, 1) // Divide by 2
SV64_IU(TmpVec1, block, 8) // Store MC pels into block and
                           // auto update address
```
Unrestricted Motion Compensation

The second problem for performing motion compensation in a SIMD way is unrestricted motion compensation. Unrestricted motion compensation allows that some of the reference pels lie outside the frame. The value of a pel that lies outside the frame is the value of the nearest pel inside the frame. An example of unrestricted motion compensation where the first pixel has a horizontal index of -3 is shown in Figure 4. This figure shows that the resulting vector contains four copies of element zero.

![Figure 4: Example of Unrestricted Motion Compensation (Full Pel)](image)

Two methods used in video decoders to handle unrestricted motion compensation are:

- Creating a border around the reference frame
- Testing the address of the pels during motion compensation to see if it falls outside the frame and to adjust the address if necessary.

Advantage of creating a border is that no extra instructions have to be executed during motion compensation, which is a critical loop. The downside is that memory transfers are needed to generate this border image, which are not required. The second approach does not need these memory transfers but requires extra instructions and even branches inside the inner loop of motion compensation. The trade-off is therefore to use cycles inside or outside the inner loop.

The MoMuSys decoder implements both methods for handling unrestricted motion compensation. The first method involves copying the reference frame to a larger buffer and then creating the border around the frame. Many of the cycles used by the CopyImageI function of the execution profile shown in the beginning of section 5 come from this task. This is in principle enough but the MoMuSys code also uses many cycles in the unrestricted_MC function that implements the second method.

The advantage of TIE is that it gives us the option of having the best of both unrestricted motion compensation methods without any of the drawbacks. We can extend the optimization for motion compensation shown above to handle unrestricted motion compensation. The difference between the motion compensation shown above and unrestricted motion compensation is in the two reference vectors. We have to differentiate between unrestricted motion compensation above and below the frame and to the left or right of the frame.

For unrestricted motion compensation to the left and right of the frame we need to load a vector containing all elements just inside the frame and generate a vector containing only the pel nearest the frame border. Let's use Figure 4 as an example. This would mean that the first vector inside the frame has to be loaded, which contains pels 0 to 7. From this vector we can extract a vector containing only pel 0. These two vectors can be used to generate the motion compensated vector. A fragment of the pseudo code to perform unrestricted motion compensation is shown in the following code sample.
Accelerating (MPEG-4) Video Decoding with an Xtensa Processor

// Description of Unrestricted Motion Compensation using pseudo C
int p0 = 0x01234567
int pl = 0x00000000
if (x<0) { // Pel address is to the left of the frame
    Addr = (Ref + y*width) // Address of first pel inside frame
    tmp = (abs(x)<<2
    sel_val = Funnel_Shift(pl, p0, tmp) // Create correct selector value // for sel instruction based on // lower 3 address bits
    Vec1 = LV64_I(Addr, 0) // Load 64 bits (8 pels)
    Vec0 = Vec1
    SEL(Vec0, Vec1, pl) // create left vector
    SEL(Vec0, Vec1, sel_val) // Produces the vector shown in // Figure 4 if x=-3
}

Unrestricted motion compensation above and below the frame requires that the two reference vectors for which the Y coordinate is inside the frame is loaded. Performing motion compensation using an array of pointers can support this. Assume that motion compensation has to be performed for an 8x8 block from location X=4 and Y=-3. The pointer array is shown below.

// Pointer values for unrestricted motion compensation when x=4 and Y=-3
int RowPtr[8]
RowPtr[0] = Ref + 0 // Y=-3
RowPtr[1] = Ref + 0 // Y=-2
RowPtr[2] = Ref + 0 // Y=-1
RowPtr[3] = Ref + 0 // Y=0
RowPtr[4] = Ref + width // Y=1
RowPtr[5] = Ref + 2 * width // Y=2
RowPtr[6] = Ref + 3 * width // Y=3
RowPtr[7] = Ref + 4 * width // Y=4

Performance of Optimized Motion Compensation

So far we have described how the process of motion compensation can be accelerated. Before we discuss the performance gains from these optimizations, we should include another optimization. This optimization is not really a motion compensation optimization but rather a global structure optimization. In the original code, 3.24% of the cycles are used for the AddMagel function. This function adds the frame containing all iDCT results with the frame containing all motion compensation results. Doing motion compensation on the block level and adding the iDCT result in the motion compensation function removes the need for this whole function. This optimization saves a large number of memory transfers. It also saves on the total required memory size since only a memory the size of 6 iDCT blocks is required instead of a full frame memory for all iDCT results. Therefore requiring only one frame memory plus 6 iDCT block instead of two frame memories. This enables another optimization since 6 blocks of memory can be put into a local memory providing a wide and single cycle access to it.

Another optimization that is relevant to motion compensation is that all motion vectors are represented as shorts instead of floating point numbers. This replaces floating point operations by scalar operations without loss of precision and it also removes the conversions between scalar and floating point.
The original code requires 1.66 Gigacycles cycles for performing all motion related tasks while decoding the Suzie test stream, this consist of:

- 1.269Gcycles to perform the motion compensation
- 273 Mcycles for adding the iDCT result with the motion compensation result.
- 118 Mcycles for creating a new reference frame with an edge for unrestricted motion compensation

The optimized code requires only 27.18 Million cycles to perform motion compensation.

| TABLE 4: OPTIMIZED MOTION COMPENSATION PERFORMANCE VERSUS ORIGINAL |
|---------------------------------|---------------------|--------------------------|
| **Original cycle count (x10^6)** | **SIMD TIE cycle count (x10^6)** | **TIE speedup**          |
| Miss America                    | 1620                | 21.43                    | 75.6x               |
| Suzie                           | 1660                | 27.18                    | 61.1x               |
| Foreman                         | 4571                | 90.04                    | 50.8x               |
| Car Phone                       | 4296                | 79.35                    | 54.1x               |
| Monsters Inc                    | 15578               | 194.96                   | 79.9x               |

Further optimizations are possible by automatic generation of the select word and combining the select with the load or to combine the select with the half pel calculation. But these optimizations have not been implemented since none of the motion compensation routines are in the top 5 of the final profile.

**Inverse Discrete Cosine Transform (IDCT)**

The previous section described how motion compensation could be optimized using TIE instructions. In this section, we will look at the main algorithm used in spatial compression: the inverse Discrete Cosine Transform (IDCT).

A large variety of algorithm implementations are available for computing a two-dimensional 8x8 iDCT. All these implementations can be divided into two groups: using a 2-D iDCT or multiple 1-D iDCTs. The decoder uses the implementation that applies two steps of one-dimensional iDCTs onto an 8x8 block. Written below in pseudo code is the basic structure of this iDCT implementation.

```plaintext
IDCT(short in_block[64], short out_block[64]){
  Short Tmp[64];
  Short *Inptr, *Outptr, *Tmpptr;
  For (I=1;I<8;I++){
    Inptr =&in_block[I];  // Inptr points to input column I
    Tmpptr=&Tmp[I];       // Tmp points to output column I
    Column_iDCT(Inptr, Tmpptr); //Perform a 1D-iDCT on one column
  }
  For (I=1;I<8;I++){
    Inptr =&Tmp[I*8];     // Inptr points to input row I
    Outptr=&out_block[I*8]; // Tmp points to output row I
    Row_iDCT(Inptr, Outptr); //Perform a 1D-iDCT on one row
  }
}
```
The 1-D iDCT is executed in the Column_iDCT and Row_iDCT functions, which requires 12 multiplies and 32 additions. Therefore, to optimize the 8x8 iDCT, we must optimize the 1-D iDCT. There are two basic optimization strategies that can be used for optimizing the 8x8 iDCT function:

- Optimize execution of one 1-D iDCT
- Optimize the whole 8x8 iDCT by executing all 1-D iDCTs in parallel using SIMD

Supporting butterfly calculations and rotation in TIE, which are the basis of the iDCT, can accelerate the 1-D iDCT. The problem with this strategy is that to achieve a high optimization factor requires very specialized TIE instructions.

We chose the second method since it could achieve high performance using general SIMD instructions. In principle a factor of 8 can be achieved by using our 8 way SIMD TIE engine. This would allow all column 1-D iDCTs to be calculated in parallel. Transforming the code of the Column iDCTs to SIMD execution is relatively straightforward. In principle every operator has to be replaced by a SIMD operator.

Doing the same for the row iDCTs is not possible since the data layout is incorrect. This is because we would load all elements of the row iDCT instead of the same element from all rows. Our SIMD instructions perform operations on the same element of multiple vectors and do not perform operations on different elements of the same vector.

To solve this problem we need to add two extra operations to the iDCT function, which transpose the input and output block of the row_iDCT. This will fix the data layout problem.

Figure 5 shows the transpose of a 3x3 matrix as an example. Doing a full transpose of an 8x8 block using the SEL instruction, as described in section 4, requires 24 instructions. The overhead of the two transpose operations is therefore 48 cycles. This overhead can be reduced by 32 cycles by performing the transpose operation in TIE state.

![Figure 5: Transpose of a 3x3 matrix.](image)

The pseudo code for the SIMD function is given below. It clearly shows that the ‘for’ loops have now been removed since all eight 1D-iDCTs are now performed in parallel.

```c
IDCT(short in_block[64], short out_block[64])
{
  Short Tmp[64];
  SIMD_iDCT(in_block, Tmp);  // Perform eight 1D-iDCT in parallel
  Transpose_Block(Tmp);     // Transpose block for row iDCTs
  SIMD_iDCT(Tmp, out_block); // Perform eight 1D-iDCT in parallel
  Transpose_Block(out_block); // Transpose block back
}
```

The resulting performance of the optimized 8x8 iDCT is shown in Table 5.
### TABLE 5: OPTIMIZED IDCT PERFORMANCE VERSUS ORIGINAL

<table>
<thead>
<tr>
<th></th>
<th>Original iDCT cycle count</th>
<th>SIMD TIE iDCT cycle count</th>
<th>TIE speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss America</td>
<td>3220.86</td>
<td>345.24</td>
<td>9.4x</td>
</tr>
<tr>
<td>Suzie</td>
<td>3283.00</td>
<td>343.77</td>
<td>9.5x</td>
</tr>
<tr>
<td>Foreman</td>
<td>3414.70</td>
<td>342.22</td>
<td>10.0x</td>
</tr>
<tr>
<td>Car Phone</td>
<td>3356.83</td>
<td>342.62</td>
<td>9.8x</td>
</tr>
<tr>
<td>Monsters Inc</td>
<td>3349.21</td>
<td>343.26</td>
<td>9.8x</td>
</tr>
</tbody>
</table>

### Bitstream Processing

Most video decoders use a set of functions to process the video bitstream. The main functions in the MoMuSys decoder are:

- **BitstreamShowBits(N)** returns N number of bits from the top of the bitstream, where N <= 32. Bitstream is not modified.
- **BitstreamFlushBits(N)** moves the top of the bitstream by N bits (N<=32)
- **BitstreamReadBits(N)** equals to executing BitstreamShowBits(N) followed by BitstreamFlushBits(N).

All data is extracted from the video stream using one of these functions.

The bitstream processing functions can be optimized with TIE by having the top bits of the video bitstream inside a TIE state register. Using this structure, all three bitstream access functions can be easily written in TIE. The only extra functionality that needs to be provided is a mechanism to prevent this buffer from running empty by loading the buffer, when possible with new bits. The basic structure of the TIE required to handle bitstream processing is shown in Figure 6.
A prerequisite of the TIE hardware shown above is that there are always enough bits in the TIE bitstream buffer. To support this requirement a TIE instruction is created that loads the next 32 bits from the bitstream in memory into the LoadBuffer register shown above. When 32 or more bits are available in the bitstream buffer then the address will be updated together with the bitstream register. The Bitsleft value will also be incremented by 32.

Executing a BitstreamReadBits instruction results in taking the top 32 bits from the bitstream buffer and shifting this value to the right by 32-N. This produces a 32-bit value containing the N top bits of the bitstream buffer. Then the Bitstream buffer is shifted to the left by N, removing the top N bits from it. Finally, the Bitsleft value is reduced by N to correctly represent the number of valid bits inside the bitstream buffer. The TIE description for the Bitstream processing is shown in Appendix B.

Table 6 shows the number of cycles required by the original and TIE optimized bitstream processing functions. The cycle counts for the TIE version are larger than one due to the load instruction that is present to guarantee that the Bitstream buffer contains bits and due to the call overhead. The call overhead can be reduced but this optimization is not performed since these bitstream access functions are low on the execution profile.

Table 6 shows the number of cycles required by the original and TIE optimized bitstream processing functions. The cycle counts for the TIE version are larger than one due to the load instruction that is present to guarantee that the Bitstream buffer contains bits and due to the call overhead. The call overhead can be reduced but this optimization is not performed since these bitstream access functions are low on the execution profile.

Variable Length Decoding

Performing the bitstream processing functions in TIE enables the optimization of variable length decoding. Variable length decoding is used to reduce the average number of bits to encode a
codeword by assigning a short bitstring to the most frequently occurring codeword and long bitstrings to codewords that hardly ever occur.

The encoder uses variable length coding to reduce the number of bits used for encoding an 8x8 iDCT block. Variable length decoding fills this block by extracting variable length codewords from the bitstream. This extraction requires the hardware that we have added for the bitstream access functions in the previous section. The extracted codeword represent the run, level and sign of the IDCT coefficient (also called Run length coding). The run values indicated the number of zero's that occur between two values.

MPEG-4 contains two different methods for Variable length decoding: normal and reversible. We will not discuss reversible variable length decoding here. The normal variable length decoding contains two different sets of codes: one for Intra blocks and one for Inter blocks. The decoding process as implemented in the MoMuSys code is shown below. It implements the VLD table by dividing it up into three separate tables depending on the number of leading zero's.

code = BitstreamShowBits (stream, 12); // Extract 12 bits from Bitstream

if (short_video_header)
intra = 0;

if (code >= 512)
    if (intra == 0)
        tab = &DCT3Dtab0[(code >> 5) - 16];
    else
        tab = &DCT3Dtab3[(code >> 5) - 16];
else if (code >= 128)
    if (intra == 0)
        tab = &DCT3Dtab1[(code >> 2) - 32];
    else
        tab = &DCT3Dtab4[(code >> 2) - 32];
else if (code >= 8)
    if (intra == 0)
        tab = &DCT3Dtab2[(code >> 0) - 8];
    else
        tab = &DCT3Dtab5[(code >> 0) - 8];
else
{
    /* ERROR */
}

BitstreamFlushBits (stream, tab->len);
BitstreamTraceVLC(*tab, "VLC 3D", trace);

if (intra == 0) {
    tcoef.run = (tab->val >> 4) & 255;
    tcoef.level = tab->val & 15;
    tcoef.last = (tab->val >> 12) & 1;
}
else {
    tcoef.run = (tab->val >> 8) & 255;
The variable length decoding code shown above reads the maximum code word size (12 bits) from the bitstream. Then one of the three tables is addressed depending on the value of the read bits. A VLD table returns the run, level, length and last bit. The returned length is then used to flush the bitstream by the correct number of bits. One sign bit is then read from the bitstream which combined with the level creates the iDCT value. It is written into the iDCT block location indicated by the block index incremented by the run value. The last bit indicates that this decode was the last of the current iDCT block.

VLD can be accelerated with TIE by adding a VLCdecode TIE instruction, which reads 12 bits from the bitstream TIE register, described in the previous section. It then loads the VLD run, level, last bit and length. The VLCdecode instruction returns the length of the actual code word. The run, level and last bit are written into TIE state. We also need to add 3 TIE registers which hold the base address of the three VLD tables adjusted by 32/16 and 8 as shown in the C code above.

Another TIE instruction is created which reads the sign bit and depending on the sign bit negates the level and returns the iDCT value. The TIE reference description for these instructions is shown in Appendix A.

The optimized C code for decoding the VLD entry and placing it in the iDCT block is shown below.

```c
/* Code for decoding a whole iDCT block */
do {
  tmplen=VLCdecode(intra); // TIE instruction that does the whole
  // bitstream look up and VLD load
  BitstreamFlushBits (texture_bits, tmplen); // Flush the used VLD bits
  if (RVLCescape() != ESCAPE) /* ESCAPE */
  {
    i += RVLCrun(); // Read the run value from State
    j = jpeg_natural_order[i]; // Perform inverse ZigZag
    coeff_block[j]=VLCsign(); // Read sign bit from bitstream and return
    // level or negated level.
    last = RVCLClast(); // Read last bit from state. If this bit is
    // set then we are finished decoding this
    // block
    i++;
  }
  else
  {
    /* Code for handling ESCAPE code omitted for brevity */
  }
} while (!last && (i<64));
```
For simplicity we have not discussed the ESCAPE code handling. The reason for this is that ESCAPE codes are occurring infrequently and are already sufficiently optimized by using the bitstream access function of the previous section. The performance of loading all iDCT blocks with VLD values is shown below in Table 7.

<table>
<thead>
<tr>
<th></th>
<th>Original iDCT cycle count (*10^6)</th>
<th>SIMD TIE iDCT cycle count (*10^6)</th>
<th>TIE Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss America</td>
<td>75.83</td>
<td>3.34</td>
<td>22.7x</td>
</tr>
<tr>
<td>Suzie</td>
<td>137.15</td>
<td>6.88</td>
<td>19.9x</td>
</tr>
<tr>
<td>Foreman</td>
<td>710.14</td>
<td>36.08</td>
<td>19.7x</td>
</tr>
<tr>
<td>Car Phone</td>
<td>601.80</td>
<td>29.82</td>
<td>20.2x</td>
</tr>
<tr>
<td>Monsters Inc</td>
<td>1477.47</td>
<td>70.44</td>
<td>21.0x</td>
</tr>
</tbody>
</table>

6 Putting It All Together: An Optimized MPEG-4 Decoder

In the previous section, we discussed how four parts of a video decoder could be optimized by using TIE. Two different TIE accelerators were used. The first is the SIMD engine described in section 4 and a TIE engine for handling the bitstream processing and variable length decoder.

First, we will give a brief overview of other optimizations that have been performed on the MoMuSys MPEG-4 decoder followed by the synthesis results of the TIE block. This will be followed by the performance results of the optimized MoMuSys MPEG-4 video decoder and finally we will discuss unimplemented optimizations.

Other Optimizations

We have discussed the optimizations on the large parts of a video coder in detail in section 5. In this section a brief overview is given of other implemented optimizations, note that not all these optimizations involve the use of TIE.

Dequantization

One of the methods of compression is transforming the values of an 8x8 block to the frequency domain and then dividing these numbers by a quantization value. This requires that the reverse happen at the decoder side. The quantization value for a macroblock is transmitted in the bitstream and Dequantization multiplies the decoded iDCT value with the quantization value to retrieve the approximation of the original DCT value.

Two optimization approaches are possible for dequantization. The first approach optimizes quantization of a single iDCT value. Please note that quantization only needs to be performed for non-zero iDCT values. The second approach performs quantization on multiple iDCT values at the same time using our SIMD TIE engine.

Using the SIMD approach reduces the cycle count for dequantization from 1647 cycles to 259 cycles per iDCT block for the Suzie test stream.
AC/DC Prediction

AC/DC prediction is only used for Intra frames, which occur significantly less than Inter frames. It is very important to optimize this function since Intra frames require significantly more cycles than Inter frames. The clock frequency of a real decoder will have to be high enough to decode any frame in real time, even an Intra frame. We must therefore optimize Intra frames to lower the clock frequency of a real decoder.

AC/DC prediction reduces the number of bits required for encoding an Intra frame. It estimates the DC value and/or AC values from one of multiple previous iDCT blocks. Especially the AC prediction requires a large number of cycles. These cycles are mainly used by divisions and logarithmic function inside a loop. These calculations can be moved outside the loop by rewriting the code.

This optimization, which did not require TIE, reduced the cycle count for AC/DC prediction from 2080 to 520 cycles.

Color Conversion

The MoMuSys MPEG-4 decoder generates a YCbCr image. In most cases this has to be transformed to another colorspace. The MPEG-4 demo that is based on this work has to transform from the YCbCr to RGB colorspace.

Transforming the YCbCr to RGB requires the matrix multiplication shown in Figure 7. The resulting values for R, G and B should be shifted to the right by 16 to obtain the real RGB value. Color conversion is a critical operation because this matrix multiplication happens for every pixel. The TIE SIMD multipliers were modified to be able to handle 18-bit signed constants and 16x16 bit multiplications. This higher precision is needed to prevent extra instructions since two constants are larger than the maximum-signed value allowed in 16-bit two’s complement. The multipliers still produce a 32-bit result since the Y, Cb and Cr values are only 8 bits.

Optimizing the Color Conversion with the SIMD engine results in performing the matrix multiplication for 8 pixels in parallel resulting in an optimization factor of 8.

![Figure 7: Matrix multiplication used for color conversion](image)

Post-Filtering

Post-filtering is an optional processing step after the video decoding that can easily require as many cycles as the video decoding itself. We will not discuss the post-filtering algorithms and their optimization here. They can, however, be optimized by the TIE engine used for the video decoding to achieve an optimization factor of around 8.

Memory Hierarchy

The main focus so far has been on accelerating calculations by applying SIMD instructions for accessing and processing the data. This focus neglects an area of optimization that can further improve performance, namely memory hierarchy. All performance numbers so far have assumed an ideal external memory to the core. This is a memory system that replies to the processor within one cycle of the processor clock.
A more realistic memory system would be to use a 32-bit SDRAM as external memory. This memory will take 60ns + 3*10ns = 90ns to return 128 bits over a 100Mhz interface. Now we are suddenly looking at 9 or 10 cycles for an external memory access.

It is important to make sure that the video decoder performance is not hurt too badly by the speed of the external memory. This can be partially achieved by aiming for a low miss rate for the Instruction and Data Cache. Another method is to use local memories.

Performance is improved by using an instruction RAM that contains the most critical functions. This avoids any cache misses for these functions and the corresponding dependence on memory latency. For data transfers, even more improvement can be achieved by using a local data memory. The biggest gain would be to have the frame stores in local data memory but this would require quite a large local memory, which would only be feasible for low-resolution video. These frame stores are therefore assumed to be in an external memory.

One part of the video coding that can be accelerated by using local data memory is the path of extracting iDCT blocks from the bitstream to the point where we add these blocks to the motion compensation results. These 6 iDCT blocks can then be kept in local data memory, which provides single cycle access to these blocks.

**Synthesis Results for the Core and TIE**

In the introduction, we mentioned that we would not focus on area and power but the result of the optimizations should of course still be feasible in physical and economical terms. This section will therefore discuss the area requirements for the optimized solution.

The processor configuration used in this document has the following main configuration features:

- Big Endian
- 16x16 multiplier enabled
- 2 co-processors
- density option and loop instruction enabled
- 1 miscellaneous special register
- Boolean registers enabled
- 128 bit PIF
- 32 write buffers
- 2-way set associative Instruction Cache 8KB 64 byte line size
- 3-way set associative Data Cache 6KB 64 byte line size
- 16KB data Ram
- 16KB instruction Ram

This processor configuration requires around 53K gates.

After compiling the TIE code it can be synthesized using the generated synthesize scripts. The total TIE gate count is 67K gates.

The memory and cache sizes for this configuration are not a hard requirement. The memory and cache sizes can be traded off against performance. It is even possible to remove the local instruction and data memories completely. But these trade-offs of memory area versus performance are not the focus of this application note.
Performance Results for Optimized MoMuSys Video Decoder

After spending 7 man-months on the optimizations described above, we could run the test streams through an optimized MPEG-4 video decoder. This is not an optimal decoder since more optimizations are still possible as will be described in the next section. But before we discuss the performance for the test streams, let’s look again at the execution profile of the optimized decoder for the Suzie test stream shown below.

Flat profile:

<table>
<thead>
<tr>
<th>flat profile:</th>
<th>cumulative cycles</th>
<th>self cycles</th>
<th>calls</th>
<th>self total cycles</th>
<th>total cycles</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(K)</td>
<td>(K)</td>
<td></td>
<td>(K)</td>
<td>(K)</td>
<td></td>
</tr>
<tr>
<td>11.53</td>
<td>11783.38</td>
<td>11783.38</td>
<td>34277</td>
<td>0.34</td>
<td>0.34</td>
<td>IDCT</td>
</tr>
<tr>
<td>8.66</td>
<td>20637.21</td>
<td>8853.83</td>
<td>34277</td>
<td>0.26</td>
<td>0.26</td>
<td>BlockDequantH263</td>
</tr>
<tr>
<td>6.50</td>
<td>27279.71</td>
<td>6642.50</td>
<td>11059</td>
<td>0.60</td>
<td>3.62</td>
<td>GetMBblockdata</td>
</tr>
<tr>
<td>6.38</td>
<td>33796.59</td>
<td>6516.88</td>
<td>35451</td>
<td>0.18</td>
<td>0.20</td>
<td>Unrestr_MC</td>
</tr>
<tr>
<td>4.99</td>
<td>38897.11</td>
<td>5100.53</td>
<td>33671</td>
<td>0.15</td>
<td>0.19</td>
<td>GetBlockNoRVLC</td>
</tr>
<tr>
<td>4.05</td>
<td>43032.01</td>
<td>4134.90</td>
<td>148</td>
<td>27.94</td>
<td>630.26</td>
<td>UnrestrHpXHpY_MC</td>
</tr>
<tr>
<td>3.81</td>
<td>46924.44</td>
<td>3892.44</td>
<td>44228</td>
<td>0.09</td>
<td>0.33</td>
<td>GetPred_Advanced</td>
</tr>
<tr>
<td>3.64</td>
<td>50641.74</td>
<td>3717.30</td>
<td>14540</td>
<td>0.26</td>
<td>0.37</td>
<td>find_pmvs</td>
</tr>
<tr>
<td>3.37</td>
<td>54081.17</td>
<td>3439.43</td>
<td>12546</td>
<td>0.27</td>
<td>0.29</td>
<td>UnrestrHpXHpY_MC</td>
</tr>
</tbody>
</table>

The top function is now the iDCT followed by the function that performs the dequantization. This profile assumes an ideal memory system. If a more realistic memory system is used then two of the motion compensation routines: Unrestr_MC and UnrestrHpXHpY_MC functions will be higher in the list due to the memory latency. Notice that all functions related to copying/adding or filling images have disappeared due to our optimizations, leaving the real operations. All numbers in this profile are in Kcycles (10^3 cycles) instead of Mcycles (10^6 cycles for the original profile.

Table 8 shows the processor cycles required for decoding the different test streams for the original decoder and the optimized decoder. Both of these decoders run on the same processor configuration.

| TABLE 8: OPTIMIZED MPEG-4 VIDEO DECODER PERFORMANCE VERSUS ORIGINAL |
|-------------------|-------------------|------------------|-------------------|-------------------|
|                  | Original video decoder performance | Optimized video decoder performance | Clock freq 15fps⁴ | TIE Speedup |
| Miss America     | 3.126 x 10⁹       | 76.81 x 10⁶      | 7.7 Mhz           | 40.1x            |
| Suzie            | 3.389 x 10⁹       | 102.19 x 10⁶     | 10.3 Mhz          | 33.2x            |
| Foreman          | 10.045 x 10⁹      | 359.5 x 10⁶      | 13.5 Mhz          | 27.9x            |
| Car Phone        | 9.222 x 10⁹       | 308.7 x 10⁶      | 12.2 Mhz          | 29.9x            |
| Monsters Inc     | 29.327 x 10⁹      | 822.8 x 10⁶      | 8.6 Mhz           | 35.6x            |

Table 8 shows that the optimization factor is between 27 and 40 for the different test streams.

The reason for the large difference in optimization factor is due to differences in the bitstreams. Motion compensation is optimized more than performing dequantization and iDCT for instance. Therefore, streams that use more iDCT blocks require more cycles. The difference between the

⁴ Clock frequency refers to the average clock frequency required. Clock frequency for a real system has to be higher. The measure is used to give an indication of the required clock frequency.
streams can be reduced by performing the same operation per macroblock independent of its content or by further optimizing the different algorithms.

As is shown in Table 8, different test streams have different requirements. This leads to the problem of what is the worst-case stream that a decoder has to handle. But it also shows how important it is to know what test stream is decoded when looking at performance numbers for a video decoder solution.

**Unimplemented Optimizations**

In 7 months, we went from a downloaded MPEG-4 video decoder that we didn’t understand to an optimized version that runs more than 25 times faster. Moreover, there is still further room for optimizing the performance. In this section, we want to give you as the reader of this application note an impression of what can be further optimized. We will, of course, not dive into detail on the different optimization or claim that this is a complete list of all optimizations that are still left.

First of all the code still contains a lot of conditional code and function calls that find their origin in the generality of the code. All test streams used here are QCIF test streams that can be decoded by a simple profile MPEG-4 video decoder. Focusing on a smaller subset of the visual standard such as simple profile will simplify the code and reduce the amount of cycles needed.

Dequantization can still be further optimized by adding SIMD instructions that performs a different computation depending on the sign of the iDCT value. This requires at least 5 instructions at the moment, whereas this can easily be done within one specialized TIE SIMD instruction.

The iDCT algorithm can be further accelerated. Within Tensilica, we have achieved less than 200 cycles for performing an iDCT on a dedicated TIE engine. This is still a serious optimization over the current 350 cycles.

Motion compensation can still be further optimized. Firstly the number of cycles for selecting the right 8 pels can be optimized by a factor of 3 at least. But the main optimization that should be considered for a real system is to decouple the issue of the load with the return of the data. The motivation for this is that memory latency can easily run into tens of cycles. The Xtensa processor will be stalled during this time. This while the processor could do other work while the block is loaded from memory. Therefore, Motion Compensation could initiate a block read from memory through an external device that writes the data into the local memory by using the T1050 inbound PIF feature, for instance. The processor can now calculate iDCTs while the load of an 8x8 region is performed.

7 Conclusions

More and more products seem to require support for some type of video decoding or even encoding. In this application note we describe how Xtensa’s design methodology of configurability and especially extensibility can optimize video decoding. Using a 120K-gate core we can run the MoMuSys MPEG-4 video decoder between 25 and 40 times faster than the original version of the code running on the same core.

It is very important to note that all these results have been achieved in 7 man months. This fast time to demo was possible because of the design methodology that Tensilica offers. The design effort could be focused on optimizing the MoMuSys decoder by adding a SIMD TIE engine and a Bitstream processing TIE engine without having to worry about correctness of the resulting processor. All the optimization work was done using the Instruction set simulator and TIE compiler, which provide fast optimization cycles. After optimizing the code I had the code running on an FPGA implementation of the processor core in days.

The resulting video decoder can be seen running on our XT2000 evaluation board decoding the Monsters Inc. trailer in real time.
Appendix A – TIE Code for Bitstream Processing and Variable Length Decoding

Here is the complete TIE description used for accelerating the bitstream processing. The GetBits, ShowBits and FlushBits implement the Bitstream Processing functions BitstreamGetBits, BitstreamShowBits and BitstreamFlushBits respectively.

VLCdecode and VLCsign are used for accelerating the variable length decoding.

```plaintext
interface VAaddr       32      core    out
interface MemDataIn32  32      core    in

opcode BITSTR  op2 = 4'b1111 CUST1
opcode LdInBuf  r=4'b0000 BITSTR
opcode GetBits r=4'b0001 BITSTR
opcode ShowBits r=4'b0010 BITSTR
opcode FlushBits r=4'b0011 BITSTR
opcode VLCdecode r=4'b0100 BITSTR
opcode VLCsign  r=4'b0101 BITSTR

state NextPtr  32
state NrBytes  24
state InBuf  64
state BitsLeft  7
state LdBuffer  32
state ValidLdBuf  1
state nb  5

state VLCtable0  32
state VLCtable1  32
state VLCtable2  32
state VLCrun  16
state VLClevel  16
state VLClast  1
state VLCescape  16

user_register NextPtr   0 NextPtr
user_register NrBytes   1 NrBytes
user_register InBuf0  2 InBuf[63:32]
user_register InBuf1  3 InBuf[31:0]
user_register BitsLeft  4 BitsLeft
user_register LdBuffer  5 LdBuffer
user_register ValidLdBuf  6 ValidLdBuf
user_register nb  7 nb
```
Accelerating (MPEG-4) Video Decoding with an Xtensa Processor

user_register VLCrun  8  VLCrun
user_register VLClevel  9  VLClevel
user_register VLClast  10  VLClast
user_register VLCtable0  11  VLCtable0
user_register VLCtable1  12  VLCtable1
user_register VLCtable2  13  VLCtable2
user_register VLCescape  14  VLCescape

iclass LdInBuf  {LdInBuf}
  {out bt}
  {inout NextPtr, inout NrBytes,}
  {inout InBuf, inout BitsLeft,}
  {inout LdBuffer, inout ValidLdBuf}
  {out VAddr, in MemDataIn32}
iclass GetBits  {GetBits}
  {out art, in ars}
  {inout InBuf, inout BitsLeft}
  {}
iclass ShowBits  {ShowBits}
  {out art, in ars}
  {in InBuf}
  {}
iclass FlushBits {FlushBits}
  {in ars}
  {inout InBuf, inout BitsLeft}
  {}
iclass VLCdecode {VLCdecode}
  {out art, in ars}
  {in InBuf, in VLCtable0, in VLCtable1, in VLCtable2, out VLCrun,}
  {out VLClevel, out VLClast, out VLCescape}
  {out VAddr, in MemDataIn32}
iclass VLCsign  {VLCsign}
  {out art}
  {inout InBuf, inout BitsLeft, inout VLClevel}
  {}

reference LdInBuf
{
  wire [31:0] Tmp;
  wire TmpValid = ValidLdBuf;
  assign VAddr = NextPtr;
  assign Tmp = MemDataIn32;
  wire cntl = (BitsLeft>32)|NrBytes==0;
  wire [5:0] shifamount = 32-(BitsLeft[5:0]);
  wire [63:0] TmpLdBuffer=(shifamount==32)?{LdBuffer,32'b0}:
                      ((32'b0,LdBuffer)) << shifamount[4:0];
  wire [63:0] TmpInBuf = ((cntl)!(TmpValid==1'b0))? InBuf:
                      InBuf|TmpLdBuffer;
  assign NrBytes = (cntl)? NrBytes: NrBytes - 4;
}
assign NextPtr = (cntl)? NextPtr : NextPtr + 4;
assign InBuf = (cntl)? InBuf:TmpInBuf;
assign BitsLeft = ((cntl)|(TmpValid==1'b0)) ? BitsLeft : (BitsLeft + 32);
assign LdBuffer = (cntl)? LdBuffer : Tmp;
assign ValidLdBuf = (NrBytes==0)?ValidLdBuf:1'b1;
assign bt = NrBytes==0;
}

reference GetBits
{
  wire [4:0] shiftval = (ars[5]==1'b1)? 5'b0:(32-ars[4:0]);
  assign art = (InBuf[63:32])>>(shiftval);
  assign InBuf = (ars[5]==1'b1)? {InBuf[31:0], 32'b0}:
    InBuf<<ars[4:0];
  assign BitsLeft = BitsLeft-ars[5:0];
}

reference ShowBits
{
  wire [4:0] shiftval = (ars[5])?0:32-ars[4:0];
  assign art = (InBuf[63:32])>>(shiftval);
}

reference VLCdecode
{
  wire [11:0] TmpVal = (InBuf[63:52]);
  wire Intra = ars;
  assign VAddr = (TmpVal>=512)?VLCtable0+{TmpVal[11:5],2'd0}:
    (TmpVal>=128)?VLCtable1+{TmpVal[8:2],2'd0}:
      VLCtable2+{TmpVal[6:0],2'd0};
  wire [15:0] vlcval= MemDataIn32[31:16];
  assign VLCrun = (Intra)?vlcval[14:8]:vlcval[11:4];
  assign VLClevel = (Intra)?vlcval[7:0]:vlcval[3:0];
  assign VLClast = (Intra)?vlcval[15]:vlcval[12];
  assign VLCescape = vlcval;
  assign art = {16'd0,MemDataIn32[15:0]};
}
schedule VLCdecode {VLCdecode}
{
  def art 2;
  def VLCrun 2;
  def VLClevel 2;
  def VLClast 2;
  def VLCescape 2;
}

reference VLCsign
Accelerating (MPEG-4) Video Decoding with an Xtensa Processor

```vhdl
{  
  wire sign = InBuf[63];
  assign InBuf = {InBuf[62:0], 1'b0};
  assign BitsLeft = BitsLeft-1;
  wire [15:0] level = (sign)?0-VLClevel:VLClevel;
  assign VLClevel = level;
  assign art = {16{level[15]}, level};
}

reference FlushBits
{
  assign InBuf = (ars[5]==1'b1)? {InBuf[31:0], 32'b0}:
                    InBuf<<ars[4:0];
  assign BitsLeft = BitsLeft-ars[5:0];
}
```