Benchmark of MPEG-2 Video Decoding on ePUMA Multi-core DSP Processor

Examensarbete utfört i Datorteknik
vid Tekniska högskolan vid Linköpings universitet
av

Xiaoyi Peng

LiTH-ISY-EX--11/4459--SE

Linköping 2011
Benchmark of MPEG-2 Video Decoding on ePUMA Multi-core DSP Processor

Examensarbete utfört i Datorteknik vid Tekniska högskolan i Linköping av

Xiaoyi Peng

LiTH-ISY-EX--11/4459--SE

Handledare: Jian Wang
            ISY, Linköpings universitet

Examinator: Di Wu
            ISY, Linköpings universitet

Linköping, 6 December, 2011
Video decoding technologies have been widely used in our daily life. Higher resolutions and more advanced coding technologies may promote the capabilities of video decoding. A new multi-core digital signal processing processor, ePUMA, which stands for embedded Parallel DSP platform with Unique Memory Access, is chosen to investigate how it supports video decoding.

This thesis aims to benchmark the algorithms of video decoding and evaluate the performance using ePUMA in MPEG-2 standard, which is a common standard with the purpose of compressing video signals. Based on the slice-parallelism methodology on eight co-processors of ePUMA, the implementation of the algorithms consists of variable length decoding, inverse scan, inverse quantization, two-dimensional inverse discrete cosine transform, motion vector decoding, form prediction and motion compensation. The performance of the kernels is benchmarked by ePUMA system simulator. The result shows that to decode real-time Full HD (1920*1080 pixels, 30 frames per second) video, it will require ePUMA to run at 280 MHz for I frames and at 320 MHz for P frames.
Abstract

Video decoding technologies have been widely used in our daily life. Higher resolutions and more advanced coding technologies may promote the capabilities of video decoding. A new multi-core digital signal processing processor, ePUMA, which stands for embedded Parallel DSP platform with Unique Memory Access, is chosen to investigate how it supports video decoding.

This thesis aims to benchmark the algorithms of video decoding and evaluate the performance using ePUMA in MPEG-2 standard, which is a common standard with the purpose of compressing video signals. Based on the slice-parallelism methodology on eight co-processors of ePUMA, the implementation of the algorithms consists of variable length decoding, inverse scan, inverse quantization, two-dimensional inverse discrete cosine transform, motion vector decoding, form prediction and motion compensation. The performance of the kernels is benchmarked by ePUMA system simulator. The result shows that to decode real-time Full HD (1920*1080 pixels, 30 frames per second) video, it will require ePUMA to run at 280 MHz for I frames and at 320MHz for P frames.
Acknowledgments

First of all, I would like to thank Professor Dake Liu to provide this opportunity to do this master thesis at Division of Computer Engineering in Linköping University. Then I would like to express my most sincere appreciation to my examiner Di Wu for his support and comments, and to my supervisor Jian Wang for lots of discussion and supervision of my work. Also, I want to thank Andréas Karlsson for his help with understanding some key knowledge. Finally, I would like to thank my families and friends for their love and support.
Contents

1 Introduction .......................... 1
   1.1 Background ...................... 1
   1.2 Purpose .......................... 2
   1.3 Scope ............................ 2
   1.4 Outline ......................... 3

2 Video Codec Fundamental ............. 5
   2.1 Introduction to Video Codec .... 5
   2.2 Video Stream .................... 6
      2.2.1 Video Stream Structure .... 6
      2.2.2 Progressive and Interlaced Sequences .... 7
      2.2.3 Picture Type and Reordering ... 8
      2.2.4 Color Component Representation ... 9

3 Overview of MPEG-2 Video Decoding .... 11
   3.1 MPEG-2 Video Decoding Software Structure .... 11
   3.2 MPEG-2 Video Codec Process ............ 13
      3.2.1 DCT and IDCT .................. 13
      3.2.2 Quantization and Inverse Quantization ... 15
      3.2.3 ZIGZAG Scan and Inverse Scan .... 16
      3.2.4 VLC and VLD .................. 17
      3.2.5 Motion Estimation, Motion Prediction and Motion Compensation ... 19

4 Overview of ePUMA Architecture ...... 23
   4.1 Master Senior processor ............ 23
      4.1.1 Master Memory .................. 23
      4.1.2 Datapath ........................ 24
      4.1.3 Instruction Set ................. 24
   4.2 SIMD Sleipnir Co-processor ........... 25
4.2.1 Memory Architecture ........................................ 25
4.2.2 Pipline ......................................................... 27
4.2.3 Instruction Set .................................................. 28
4.3 Memory hierarchy .................................................. 29
4.4 Network Architecture ............................................. 29
4.5 DMA ............................................................. 30
4.6 Simulator .......................................................... 31

5 Design and Implementation ........................................ 33
5.1 Slice-layer Parallelsim ............................................. 33
5.2 Master Control Flow ............................................... 34
5.3 Kernel Implementation .......................................... 38
  5.3.1 Implementation on Master processor ....................... 38
  5.3.2 Implementation on Co-processor ............................ 38

6 Results and Discussion ........................................... 51
6.1 Execution Time .................................................... 51
6.2 Kernel Cost ....................................................... 52
6.3 Comparison ......................................................... 53
6.4 Analysis ............................................................ 54
6.5 Discussion .......................................................... 55

7 Conclusion and Future Work ..................................... 57
7.1 Conclusion .......................................................... 57
7.2 Suggestion .......................................................... 58
7.3 Future Work ........................................................ 58

Bibliography .......................................................... 59

A Appendix of ePUMA ................................................. 61

B Figures of Some Implementation .................................. 65
  B.1 Intra and non_intra Block VLD on ePUMA ................. 65
  B.2 Motion Vector Decoding on ePUMA ......................... 65

C Lists of Some Implementation .................................... 68
Chapter 1

Introduction

1.1 Background

Now that multimedia technology has been widely implemented in many modern applications, such as mobile devices, DVD or Blue-Ray players, High-Definition Television (HDTV). To achieve high computing power at a low consumption, video decoding algorithms play an important role on the performance. Since higher resolutions are required and more complicated coding technologies have been introduced, the need for more computing capability promotes the development of video decoding.

The Division of Computer Engineering at Linköping University has developed a platform called ePUMA, which emphasizes on the Embedded Parallel Digital Signal Processing Processor based on master-multi-SIMD architecture, that consists of one master core and eight slave cores working in parallel. It satisfies the high efficiency demands as result of the utilization of advanced technologies, such as P3RMA (Programmable, Parallel, and Predictable Random Memory Access), combined network connections, and enhanced Direct Memory Access (DMA). The essential one is to use separated data access kernels and algorithm kernels to minimize the communication overhead of parallel processing by running kernels in parallel based on its architecture.[2] The brief architecture of ePUMA has shown in figure 1.1.

MPEG-2 standard is an extension of the MPEG-1 international standard in 1994 with the aim of compress digital video and audio signals, which is a major mechanism due to its popularity in the real world, for instance, the transmission on digital television and widely-use DVD players. It stan-
dardizes both encoder and decoder parts of multimedia signal, particularly providing the progressive and interlaced video resolutions. Also later compression standards are similar with the MPEG-2 in structure, making it a good choice to implement real-time video decoding on ePUMA platform in MPEG-2 firstly.

1.2 Purpose

The tasks of this thesis include:

- Identify the critical computing kernels in video decoding algorithm of MPEG-2.
- Implement the kernels in ePUMA parallel DSP.
- Benchmarking the kernel performance.
- Predict the system performance.

1.3 Scope

MPEG-2 decoder software provided by MPEG Software Simulation Group and the MPEG-2 standard are the main references. The most time dissipation part is understanding the structure and principle of MPEG-2 video
decoding. The implementation consists of variable length decoding, inverse scan, inverse quantization, two-dimensional inverse discrete cosine transform, motion vector decoding, form prediction and motion compensation. ePUMA simulator developed by Linköping University is used to simulate the algorithms and evaluate the performance. What is more, some analysis and conclusion are made according to result and execution time.

1.4 Outline

This thesis is targeted to audiences with education background of electrical engineering, computer engineering or similar. The contents of the chapters are described as below.

- **Chapter 1 - Introduction**: Describes the background, purpose and scope of this thesis.

- **Chapter 2 - Video Codec Fundamentals**: Illustration about the video codec knowledge.

- **Chapter 3 - Overview of MPEG-2 video decoding**: The whole flow of MPEG-2 video stream decoding principle is explained in this chapter.

- **Chapter 4 - Overview of ePUMA architecture**: A brief introduction of ePUMA platform hardware and major technologies.

- **Chapter 5 - Design and Implementation**: This chapter presents the design procedure of the implementation and detailed description of function.

- **Chapter 6 - Result and Discussion**: According to the final result, the discussion of the kernel performance mainly focuses on execution time.

- **Chapter 7 - Conclusion and Future Work**: Conclusion of this thesis and some future work would be written in this chapter.
Chapter 2

Video Codec Fundamental

Video is made up of pictures or frames technically and pixel is a smallest unit or element in video. In the video frames, pixel values are not independent but are correlated with their neighbors both within the same frame and across frames, that are defined as spatial and temporal redundancy.[8] Because of the very large redundancy information in the moving pictures, the files are required to be compressed into acceptable storage size so that it increases the utilization of transmission bandwidth.

2.1 Introduction to Video Codec

As the figure 2.1 shown, a pair of encoder and decoder is called codec. The purpose of encoder is to get rid of the huge redundancy data and decoder obviously is a sort of reconstruction or inverse process to coded data. For the encoder, first of all, whether motion estimation works or not is judged by prework modules to intra or non_intra frame. If it doesn’t work, the data is compressed by Discrete Cosine Transform (DCT), quantization, scan and Variable Length Coding(VLC) step by step. If it does work, the motion vectors are produced with the current and reference frame. Then the current picture minus the prediction value to produce the residual value. The last the residual value goes through the normal compress processes one by one.

For the decoder, it is generally separated into two kinds of decoding processes based on intra or non_intra frame. For intra picture, it goes to Variable Length Decoding(VLD), Inverse Quantization(IQ) and Inverse DCT(IDCT). For non_intra picture, it goes to motion vector decoding and
motion compensation that produces motion residual value. And then add those motion residual with data after VLD, IQ and IDCT to reconstruct the frame. The details of each functional module would be further written.

![Diagram of MPEG-2 Codec Structure](image)

Figure 2.1: MPEG-2 Codec Structure

### 2.2 Video Stream

#### 2.2.1 Video Stream Structure

Coded video data consists of ordered sets of layers. In general, video bit-stream can be thought as a syntactic hierarchy in which syntactic structures contain one or more subordinate structures.[9] As shown in figure 2.2, the hierarchy from high to low contains six layers: sequence, group of picture(GOP), picture, slice, macroblock and block. Picture and slice layers have particular header or start code, including the notification of the beginning of the layer and some extra information.
2.2 Video Stream

2.2.2 Progressive and Interlaced Sequences

MPEG-1 was targeted for video with up to about $15$ Mbit/s but restricted with non-interlaced (i.e. progressive) video. In order to fulfill the requirement of interlaced resolution of TV scanning, the optimization was raised up later, which is extended in MPEG-2.

The highest layer in the bit stream is video sequence. Progressive sequence is defined that all the properties of a frame are sampled at the same instances in time. Interlaced sequences frames with alternating lines of the frame represent different instances in time.\[^1\] In figure 2.3, an interlaced example shows one frame sampling in odd-numbered lines combines with one frame sampling in even-numbered lines to construct a entire picture. Such every other line of samples constructs a sample named field, that a frame consists of a top field and a bottom field. Top field is the field that contains the top-most line and the bottom filed is the other.

![MPEG-2 Video Bitstream Structure](image)
2.2.3 Picture Type and Reordering

Picture Type

Each GOP is a group of I, P, B pictures, which is defined that I is intra picture and B and P are non_intra picture.

- **I** - an Intra-coded(I) picture is coded using its own information. As mentioned before, the data coded by DCT, quantization and VLD, without going through motion estimation and motion prediction, is I-picture. I-picture is the basic frame of GOP, which would be the reference to P and B picture. Therefore, the first frame in a GOP of coded stream must be I.

- **P** - a Predictive-coded(P) picture is coded using motion prediction from a past reference.[9]

- **B** - a Bidirectionally predictive-coded(B) picture is coded using motion compensated prediction from a past and/or future reference. [9]

Picture Reordering

The order of the coded frames in the coded bitstream is the order in which the decoder processes them, but not necessarily in the correct order of display.[9] To enable backward prediction from a future frame, the encoder orders the pictures from natural display order to bitstream order so that the B-picture is transmitted after the previous and next pictures it references.[8] This results in reordering at decoder part to rearrange the frames, which can be comprehended in example 2.1.

--- Example 2.1: Example of Picture Reordering

- At the encoder input:
  
- After ordering, at the encoder output is the coded bitstream, and goes to decoder input:

- After reordering, at the decoder output:

### 2.2.4 Color Component Representation

A Slice is a series of an arbitrary number of consecutive macroblocks. A macroblock contains a section of rectangular matrices representing luminance (Y) and spatially corresponding chrominance (Cb and Cr) values, which is called YUV as well. The term 4:2:0 are often used to describe the sampling structure of the picture, which means the chrominance is horizontally and vertically subsampled by a factor of two relative to the luminance.[8] Therefore, A macroblock in 4:2:0 sample rate includes four Y blocks, one Cb block and one Cr block. The block, 8*8 pixels data, either refers to the reconstructed data, DCT objects or to ZIGZAG scan.

There are several other sampling format, for example 4:2:2. The comparative figure 2.4 between 4:2:0 and 4:2:2 sampling formats describes the difference in a macroblock and in sequential coded stream. The reason for half sampling in the chrominance space than in luminance of a image is that human eyes are more sensitive to brightness than to color.

<table>
<thead>
<tr>
<th>Sampling Format</th>
<th>Macroblock Sampling Structure</th>
<th>Block Sampling Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>4:2:0</td>
<td><img src="image1.png" alt="4:2:0 Macroblock" /></td>
<td><img src="image2.png" alt="4:2:0 Block" /></td>
</tr>
<tr>
<td>4:2:2</td>
<td><img src="image3.png" alt="4:2:2 Macroblock" /></td>
<td><img src="image4.png" alt="4:2:2 Block" /></td>
</tr>
</tbody>
</table>

Figure 2.4: YCbCr Sampling Principle
Chapter 3

Overview of MPEG-2 Video Decoding

MPEG-2 standard of International Telecommunication Union and MPEG-2 software provided by www.mpeg.org are the main references to the implementation. Understanding the principles and the software structure play an important role in design and implementation of this thesis. Moreover, the result that produced from the MPEG-2 software would be compared with that on ePUMA platform, to evaluate the performance of the implementation. Hence, in this chapter video decoding structure of MPEG-2 software and some principle of basic video codec is written.

3.1 MPEG-2 Video Decoding Software Structure

The figure 3.1 shows the software structure in MPEG-2 decoder. The Picture Decode could be divided into three loops. In the first loop to each block, it goes through Variable Length Decode, Inverse Scan, Inverse Quantization, Form Prediction, IDCT and Motion Compensation. Secondly, Macroblock Modes, Motion Vector, Coded Block Pattern are decoded to each macroblock. The last it decodes Start of Slice part that includes the start information like start notification, quantizer scale code as well as the macroblock address increment. Once all the slices in a picture are finished the software exists and jumps to next.
Figure 3.1: Software Structure in MPEG2 Video Decoding
3.2 MPEG-2 Video Codec Process

Figure 3.2 illustrates simplified encoding process for a macroblock of 4:2:0 sampling structure video. A simplified decoding part would be reversed. Brief knowledge of encoding technologies are introduced as well for better comprehending the decoding technologies. And I would like to write them in pairs.

![Figure 3.2: Simplified Video Encoding Processes](image)

3.2.1 DCT and IDCT

**DCT**

DCT (Discrete cosine transform), that has been mentioned several times before, is a mathematical transform related to the Fourier transform. It is a widely-used algorithm to compress the image or video, referring to one block data. A two-dimensional DCT performs on 8 pixel by 8 lines, which is defined as:

\[
F(u, v) = \frac{2}{N} C(u) C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} f(x, y) \cos \left( \frac{(2x + 1)u\pi}{2N} \right) \cos \left( \frac{(2y + 1)v\pi}{2N} \right)
\]

with \( u, v, x, y = 0, 1, 2, \ldots N-1 \), where \( x, y \) are spatial coordinates in the sample domain; \( u, v \) are coordinates in the transform domain.

\( C(u), C(v) \) are the coefficients corresponding with \( u, v \).

\[ C(u), C(v) = \begin{cases} 
\frac{1}{\sqrt{2}} & \text{for } u, v = 0 \\
1 & \text{otherwise}
\end{cases} \]

The scene with less movement is recommended to be encoded by frame-based DCT because of very few change in adjacent two lines. On the other
Overview of MPEG-2 Video Decoding

hand, the frame with the strenuous movement is encoded by field-based DCT. The structure in MB is different in luminance on these two encoding methods, which has been shown in figure 5.5. In the case of frame-based DCT, each block shall be composed of lines from the two fields alternately. In the case of field-based DCT, each block shall be composed of lines from only one of the two fields.[9] But for chrominance in 4:2:0 format, it always constructs based on frame.

![Figure 3.3: MB Structure by Frame-based and Field-based DCT](image)

**Figure 3.3: MB Structure by Frame-based and Field-based DCT**

**IDCT**

The Inverse DCT (IDCT) is defined as:

\[
    f(x, y) = \frac{2}{N} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} C(u)C(v)F(u, v) \cos \left( \frac{(2x + 1)u\pi}{2N} \right) \cos \left( \frac{(2y + 1)v\pi}{2N} \right)
\]

In the implementation, one-dimensional horizontal fast IDCT is applied firstly, another one-dimensional vertical fast IDCT works again. In each dimensional IDCT part, the kernel is composed of four stages. For more information, please read the reference document[9].
3.2.2 Quantization and Inverse Quantization

Quantization

Quantization is the procedure of constraining from a large set of value to a relatively small set, which is modified into two mechanisms in MPEG-2. The first one is quantization matrix, that is defined in following two matrices depending on intra blocks or non_intra blocks for 4:2:0 structure frame, is used to produce the intermediate quantized result. The second step is the intermediate value would be divided again in order to decrease the bits by a scale factor, that is named \textit{quantization\_scale} with the 5 bits.

For intra blocks (both luminance and chrominance) matrix, $qmat=$:

\[
\begin{array}{cccccccc}
8 & 16 & 19 & 22 & 26 & 27 & 29 & 34 \\
16 & 16 & 22 & 24 & 27 & 29 & 34 & 37 \\
19 & 22 & 26 & 27 & 29 & 34 & 38 & 40 \\
22 & 22 & 26 & 27 & 29 & 34 & 37 & 40 \\
22 & 26 & 27 & 29 & 32 & 35 & 40 & 48 \\
26 & 27 & 29 & 32 & 35 & 40 & 48 & 58 \\
26 & 27 & 29 & 34 & 38 & 46 & 56 & 69 \\
27 & 29 & 34 & 38 & 46 & 56 & 69 & 86 \\
\end{array}
\]

For non\_intra blocks (both luminance and chrominance) matrix, $qmat=$:

\[
\begin{array}{cccccccc}
16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\
16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\
16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\
16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\
16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\
16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\
16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\
16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\
\end{array}
\]

Inverse Quantization

In the Inverse Quantization (IQ) the multiplication not only works on a matrix value, but also on the \textit{quantiser\_scale} which is explicitly indicated in the quantization arithmetic. A mapping table between \textit{quantization\_scale\_code} and \textit{quantiser\_scale} is written in the MPEG-2 white paper for appropriate quantiser\_scale\_factor to multiply.
The detailed arithmetic of IQ in intra block is:

\[ val = (val \times quantizer\_scale \times qmat[j]) >> 4 \]

The detailed arithmetic of IQ in non-intra block is:

\[ val = (((val << 1) + 1) \times quantizer\_scale \times qmat[j]) >> 5 \]

### 3.2.3 ZIGZAG Scan and Inverse Scan

#### ZIGZAG Scan

Since in a block the most valid or non-zero data is centralized at left top area, the ZIGZAG scan is introduced for transforming the data from array format to sequential for further encoding. There are two scanning definition in the standard. In the figure 3.4 it is an example based on pattern 1, which describes how to read the value one by one according to the address pattern.

**Scan pattern 0:**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>14</td>
<td>15</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>7</td>
<td>13</td>
<td>16</td>
<td>26</td>
<td>29</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>12</td>
<td>17</td>
<td>25</td>
<td>30</td>
<td>41</td>
<td>43</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>18</td>
<td>24</td>
<td>31</td>
<td>40</td>
<td>44</td>
<td>53</td>
</tr>
<tr>
<td>10</td>
<td>19</td>
<td>23</td>
<td>32</td>
<td>39</td>
<td>45</td>
<td>52</td>
<td>54</td>
</tr>
<tr>
<td>20</td>
<td>22</td>
<td>33</td>
<td>38</td>
<td>46</td>
<td>51</td>
<td>55</td>
<td>60</td>
</tr>
<tr>
<td>21</td>
<td>34</td>
<td>37</td>
<td>47</td>
<td>50</td>
<td>56</td>
<td>59</td>
<td>61</td>
</tr>
<tr>
<td>35</td>
<td>36</td>
<td>48</td>
<td>49</td>
<td>57</td>
<td>58</td>
<td>62</td>
<td>63</td>
</tr>
</tbody>
</table>

**Scan pattern 1:**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>6</td>
<td>20</td>
<td>22</td>
<td>36</td>
<td>38</td>
<td>52</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>7</td>
<td>21</td>
<td>23</td>
<td>37</td>
<td>39</td>
<td>53</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>19</td>
<td>24</td>
<td>34</td>
<td>40</td>
<td>50</td>
<td>54</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>18</td>
<td>25</td>
<td>35</td>
<td>41</td>
<td>51</td>
<td>55</td>
</tr>
<tr>
<td>10</td>
<td>17</td>
<td>26</td>
<td>30</td>
<td>42</td>
<td>46</td>
<td>56</td>
<td>60</td>
</tr>
<tr>
<td>11</td>
<td>16</td>
<td>27</td>
<td>31</td>
<td>43</td>
<td>47</td>
<td>57</td>
<td>61</td>
</tr>
<tr>
<td>12</td>
<td>15</td>
<td>28</td>
<td>32</td>
<td>44</td>
<td>48</td>
<td>58</td>
<td>62</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>29</td>
<td>33</td>
<td>45</td>
<td>49</td>
<td>59</td>
<td>63</td>
</tr>
</tbody>
</table>
3.2 MPEG-2 Video Codec Process

3.2.4 VLC and VLD

VLC

Variable Length Coding (VLC) is a compression technique by reducing the zero numbers in the group. Some special variables are introduced in the VLC of MPEG-2 at first:

- **val, level**: a value of the pixel, which must be a non-zero data.
- **run**: the number of zeros before the non-zero value.
- **len**: a abbreviation of length, means how many bits refers to particular variable length codeword

Example 3.1 illustrates the process. The first step is to group the values into runs of zeros (run) followed by a non-zero value (val or level). Additionally, the final run of zeros is replaced with an end of block (EOB) marker. Finally, based on the VLC table 3.1, grouped data is converted
into binary coded (len).

--- Example 3.1: Example of VLC ---

1. Original: 34 16 0 7 0 0 15 0 0 0 0 0 0 ... 0

2. Group the data: {34} {16} {0 7} {0 0 15} EOB

3. VLC result: 0000 0000 1110 1100 0000 0000 0111 110 0000 0000 1010 10 0000 0000 1010 00 10

<table>
<thead>
<tr>
<th>Run</th>
<th>Level</th>
<th>Len</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>34</td>
<td>0000 0000 1110 1100</td>
</tr>
<tr>
<td>0</td>
<td>16</td>
<td>0000 0000 0111 110</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0000 0000 1010 10</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>0000 0000 1010 00</td>
</tr>
<tr>
<td>EOB</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3.1: An example of VLC table

VLD

Variable Length Decoding (VLD) in MPEG-2 includes macroblock address increment, macroblock type, motion type, motion vector, DC coefficients and AC coefficients. They are working by the similar principle so that VLD to chrominance DC coefficient is taken for an example to illustrate, as shown in the list 3.1.

A structure named VLCtab is defined with val and len two variables. The code value of 5 bits lies in 0 to 31, that could be assumed a address of a table. A pair of data like {0, 2} means {val, len}. These two tables for chrominance dc size refer to Appendix B in [9]. The detailed explanation please read the comments in the list.
3.2 MPEG-2 Video Codec Process

### Listing 3.1: Example of VLD to chrominance DC coefficient

```c
int Get_Chroma_DC_dct_diff()
{
    int code, size;
    code = Show_Bits(5);  // Read 5 bits in the bitstream to buffer.
    if (code<31)  // whether larger than 31 or not
    {
        size = DCchromtab0[code].val;  // Read the value in the table DCchromtab0 according to code.
        Flush_Buffer(DCchromtab0[code].len);  // Read the length, and shift out such many bits.
    }
    else
    {
        code = Show_Bits(10) - 0x3e0;  // read 10 bits in the bitstream and then minus 0x3e0.
        size = DCchromtab1[code].val;  // read the value in the table DCchromtab1 according to code.
        Flush_Buffer(DCchromtab1[code].len);  // read the length, and shift out such many bits.
    }
    return;

    static VLCtab DCchromtab0[32] =
    { {0, 2}, {0, 2}, {0, 2}, {0, 2}, {0, 2}, {0, 2},
      ........
    };
    static VLCtab DCchromtab1[32] =
    { {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6}, {6, 6},
      ........
    };
```

3.2.5 Motion Estimation, Motion Prediction and Motion Compensation

**Motion Estimation**

The temporal prediction technique used in MPEG-2 video is based on motion estimation, by producing the motion vector from current and reference frames. [10] The accuracy of motion estimation decides the performance of compression so that a smaller residual value would be transmitted if it is an effective motion estimation. Based on macroblock, a offset or difference is calculated between two related frames which represents motion vector (MV).

As shown in figure 3.5, one of motion vectors of the moving five-pointed star is (mv_x, mv_y). A positive value of the horizontal component of a
motion vector indicates that the prediction is made from samples (in the reference frame) that lie to the right of the samples being predicted. A positive value of the vertical component of a motion vector indicates that the prediction is made from samples (in the reference frame) that lie below the samples being predicted. All MPEG-2 motion vectors are specified to a half-pel sample grid.

![Motion Vector Diagram](image)

**Figure 3.5: Motion Vector**

**Motion Prediction**

Because of close spatial relationship in motion vectors of a picture, the change is not only found out between two pictures, but also in one frame. At the beginning, the *Prediction MV (PMV)* is produced by existed *MV*, and then goes through *delta* algorithm to calculate *motion_residual*. Such process is called motion prediction as shown in figure 3.6, by encoding *MV* and *motion_residual* in the end as well as the prediction type.

![Motion Prediction Diagram](image)

**Figure 3.6: Motion Prediction**

**Motion Compensation**

Motion compensation is an inverse process to motion estimation, by working out the prediction data from motion vectors firstly and then add it with
motion residual to reconstruct. In another word, the motion compensation process forms predictions from previously decoded pictures, and then add with the data from IDCT to recover the final picture.

Motion vectors decoding is based on following equation principle that comes from MPEG-2 standard. After VLD of motion_code and motion_residual in each MB, it goes to basic algorithms that a delta calculation works for the offset value firstly. Then it adds the delta value with previous PMV for next saturation and scalar steps as figure 3.7. In field two motion vectors are assigned to each MB in P-picture and four motion vectors are assigned to each target MB in B-picture. [11]

\[
|\text{delta}| = \begin{cases} 
|\text{motion}_\text{code}| & \text{for } f_\text{code} = 1 \text{ or } \text{motion}_\text{code} = 0 \\
(|\text{motion}_\text{code}| - 1) \times f_\text{code} & \text{otherwise} \\
+ \text{motion}_\text{residual} + 1
\end{cases}
\]

![Figure 3.7: Motion Vector Decoding](image)

Since frame and field two types of picture structures in MPEG-2, five motion compensation modes are defined as following. Generally, the first three are most widely used methods. Frame picture could be predicted based on frame or field. In the field prediction for frame pictures, the target MB in a frame is split into top field pixels and bottom field pixels and field prediction is carried out independently for each 16x8 field. For more information, please read [9].

- Frame prediction for frame pictures
- Field prediction for frame pictures
- Field prediction for field pictures
- Dual-prime prediction for P-pictures
- 16x8 MC for field picture.
Chapter 4

Overview of ePUMA Architecture

ePUMA is an abbreviate name of embedded Parallel DSP platform with Unique Memory Access, which is developed by ePUMA research team at the division of computer engineering of Linköping University. It consists of one RISC master core and eight SIMD slave cores, that master processor is responsible for tasks partition and Direct Memory Access (DMA), while slave processors aim to execute kernels in parallel.

4.1 Master Senior processor

Master processor on ePUMA is called Senior, which is a single issue DSP processor for applications. It represents general purpose and achieves high computational efficiency by issuing different tasks to SIMD co-processors. Such a RISC DSP processor controls the execution of kernels by separating the tasks into general tasks for main processor and several parallel programming arithmetics for slave processors, which transferring kernels and data from main memory to SIMIDs local memory and accomplishing them.

4.1.1 Master Memory

Three memories are directly connected to the core that one program memory (PM) which is 32 bits wide and at most $2^{16}$ words, and two data memories (DM0 and DM1) which are 16 bits wide and at most $2^{15}$ words. There are 32 general registers (r0 to r31) which is 16 bits wide for computing buffer and also 32 special registers (sr0 to sr31) with 16 bits wide, except bitrev with 3 bits wide, for special purpose like addressing, loop
and pointer.[3]

4.1.2 Datapath

The master core consists of data path, control path and address path that the data path involves a general register file, ALU and a MAC. A five-stage pipeline is a typical execution flow in Senior, that IF -> ID -> OF -> EX -> WB, except the conv (convolution) takes seven-stage pipeline. The pipeline figure is shown in Appendix A. For more reading please look at reference [4].

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Stage</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>IF</td>
<td>Instruction Fetch</td>
</tr>
<tr>
<td>P2</td>
<td>ID</td>
<td>Instruction Decode</td>
</tr>
<tr>
<td>P3</td>
<td>OF</td>
<td>Operand Fetch</td>
</tr>
<tr>
<td>P4</td>
<td>EX</td>
<td>Execution</td>
</tr>
<tr>
<td>P5</td>
<td>WB</td>
<td>Write back</td>
</tr>
</tbody>
</table>

Table 4.1: Pipeline Specification

4.1.3 Instruction Set

Senior Instruction Set is provided in reference. Though it haven’t been published because of the development of ePUMA, it could be specified into following varieties:

- **Move-load-store** instructions concern operations of 16 bits data that the move instructions (move and set) can only work with registers and accumulators (move only) for data access, which many addressing methods available for the load-store instructions.

- **Short arithmetic, logic and shift** instructions affect on 16 bits arithmetic operations by concerning conditional flags.

- **Long arithmetic** instructions like conv and mul concern 32 bits arithmetic operations, with 8 guard bits carried out.

- **Iterative** instruction (repeat) is introduced for loop purpose.

- **Flow control** instructions decide the program flow, which jump, call and ret make use of 0 to 3 delay slots, that delay slots means instructions following will be executed without the effects of a preceding
instruction, otherwise \texttt{nop} have to be inserted for not executing instruction.

- \textbf{Alias} instructions make use of special cases for their own implementation, for instance, \textit{inc} means increase the register by one and \textit{push} means push register to stack.

### 4.2 SIMD Sleipnir Co-processor

Co-processor is named \textbf{Sleipnir} with eight-way Single Instruction with Multiple Data (SIMD) datapath, that there are eight Sleipnirs in total on ePUMA platform. By exploiting the same operation on multiply processors, such a data level parallelism performs on multiple data simultaneously. In particular, Sleipnir works on vector format with 128 bits long which supports byte, word, double word, complex word and complex double word. The benefit is if an addition operation works on word type, eight words in a vector could be executed in parallel at the same time. Any type of data could possible be accessed when you addressing the vector.

#### 4.2.1 Memory Architecture

A table 4.2 explains the registers and memory overview on Sleipnir including smallest address and size. The three local vector memories (LVMs) are the main data memories while Programme Memory (PM) contains 1024 program instructions and Constant Memory (CM) takes constance data in only vector format which can be used to address LVM. Vector Register File (VRF) concerns on intermediate data which provides operation on vector, half-vector, doubld word and word. Special Purpose Register (SPR) is set up for specific use.

<table>
<thead>
<tr>
<th>Name</th>
<th>Smallest addr.</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector register file</td>
<td>Word</td>
<td>8x128bits</td>
</tr>
<tr>
<td>Special purpose registers</td>
<td>Word</td>
<td>Table 4.4</td>
</tr>
<tr>
<td>Vector accumulator register</td>
<td>Word</td>
<td>320(8x40)bits</td>
</tr>
<tr>
<td>Local vector memory</td>
<td>Word</td>
<td>4096x128 bits</td>
</tr>
<tr>
<td>Constant memory</td>
<td>Vector</td>
<td>128x128 bits</td>
</tr>
<tr>
<td>Program memory</td>
<td>_</td>
<td>1024 instructions</td>
</tr>
</tbody>
</table>

Table 4.2: Memory Overview on Sleipnir
Local Vector Memory

Each LVM is a 4096 128-bit vectors memory and there are three LVMs (LVM0, LVM1, LVM2) connected to Sleipnir. With one read port and one write port on LVM, eight banks of 16-bit are applied to provide 128 bit in parallel. The key organization referring to [4] shows how to fetch a vector from a LVM in figure 4.1, which provides eight individual data elements to access by interleaved addressing mode. LVM access modes are described in table 4.3, for instance copy vr0.0 m0[48].w means the word at address 48 of LVM0 is copied into vr0.0 register. A variety of addressing methods on LVM are described in table A.1.

![Figure 4.1: LVM Multi-bank Memory Address](image)

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Width</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>8 bits</td>
<td>Word</td>
</tr>
<tr>
<td>d</td>
<td>16 bits</td>
<td>Double</td>
</tr>
<tr>
<td>h</td>
<td>64 bits</td>
<td>Half-vector</td>
</tr>
<tr>
<td>v</td>
<td>128 bits</td>
<td>Vector</td>
</tr>
</tbody>
</table>

Table 4.3: LVM Access Mode

Special Purpose Register

Table A.2 lists the special purpose register. Take an example to illustrate the addressing mode in LVM. The following instructions in list 4.1 define the DATA_CONF vector in CM as well as DATA and DATA_END label in LVM1. After DATA_CONF been copied to LVM address register zero
(ar0c), a vector in address 0 of LVM0 is copied to VR0 at the beginning. Then it copies a word in address ar0 of LVM1 to VR1.0, and increment ar0. The last, it copies the second word in LVM1 to VR1.1. Now the VR0 contains "0 1 0 1 0 1 0 1" and VR1 contains "1 2 0 0 0 0 0 0" while in special register file ar0 = 1, bot0 = 0, top0 = 15.

### Listing 4.1: LVM Addressing Example

```plaintext
.main
dcopy ar0c cm[DATA_CONF]
    //copy the address pattern DATA_CONF to ar0c.

copy vr0 m0[0].v
    //copy a vector in address 0 of LVM0 to VR0.

copy vr1.0 m1[ar0+=1].w
    //copy a word in address ar0 of LVM1 to VR1.0, and increment ar0.

copy vr1.1 m1[ar0].w
    //copy a word in address ar0 of LVM1 to VR1.1, and increment ar0.

8*nop
stop

.m0
0 1 0 1 0 1 0 1

.m1
DATA:
    1 2 3 4 5 6 7 8
    9 10 11 12 13 14 15
DATA_END:

.cm
DATA_CONF:
DATA DATA DATA_END 0 0 0 0 0 0
```

### 4.2.2 Pipeline

The pipeline of the SIMD processor is a 2-13 stage shown in Appendix A, that data path latency differs on forwarding path or not and different output latency due to destination type. It doesn’t matter what the source operands are, only the destination operand effects the latency cycles. These are 3/6 cycles for VRF/SPR, and 4/7 cycles for LVM, in quicker fetch or not seperately, that list 4.2 gives an example.

### Listing 4.2: Operand Latency Examples

```plaintext
lslwq vr1.0 cm[0] m1[0].w
    3 * nop
lslwq m1[ar0+=8].v vr2.0 vr1.0
    4 * nop
```
Table 4.4 about read\write ports of memories on Sleipnir would result in pipeline hazards, especially the single port on LVM doesn’t support read and write to LVM at the same time. The solution of this problem is by changing the order of instructions or change to another quick fetch instructions. Also, \texttt{nop} could be inserted to solve the conflict but consume some more clock cycles.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Read/write ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector register file</td>
<td>2/1</td>
</tr>
<tr>
<td>Special purpose registers</td>
<td>1/1</td>
</tr>
<tr>
<td>Local vector memroy</td>
<td>1 combined</td>
</tr>
<tr>
<td>Constant</td>
<td>2/ _</td>
</tr>
</tbody>
</table>

Table 4.4: Read/write Ports of Memories

4.2.3 Instruction Set

Similar as instruction set in Senior, it includes following groups on Sleipnir. Different delay slots of instructions and some available condition flags have been specified in instruction set manual.

- **Copy** Instead of move-load-store instruction in Senior, \texttt{copy} works for data transmission which \texttt{copyq} performs quicker copy from source to destination and \texttt{dcopy} shows a direct copy from source to either VRF or one of the SPRs. \texttt{dcopy} costs two less cycles compared to \texttt{copyq}.

- **Word/double instruction** Some word/double logic, arithmetic and shift instructions are introduced. Since the eight-way processing in Sleipnir, operations to eight words in a vector and operations to four doubles in a vector would be respectively finished in only one clock cycle.

- **Complex instruction** Complex Multiply and Accumulate Word (\texttt{CMACW}) and Complex Multiplication Word (\texttt{CMULWWW}) are created for more work would be done in less time.

- **Compare** \texttt{cmpw/cmpwq} and \texttt{cmpd} refer to compare two words or two doubles, by changing the computation flags. \texttt{max}, \texttt{med} and \texttt{min} decide the minimum, media and maximum value in two or three elements.
• **Control** `jmp` performs a jump to specific location, `call` and `return` are work for subroutines which support up to 4 levels sub-programing, `repeat` and `repeatr` are created for repeat loop while `rreg0` makes use of loop count.

• **Other** Butterfly function for half-vector and vector raises up the algorithms’ efficiency.

### 4.3 Memory hierarchy

ePUMA has a 3-level memory hierarchy which the level-1 is processor local register file, level-2 is the processor local memory and level-3 is the external memory. Figure 4.2 depicts the memory hierarchy that level-3 main memory contains most programs and computing data and processors can not access to level-3 external memory directly, otherwise the data movement from level-3 to level-2 controlling by the DMA function.

![Figure 4.2: ePUMA Memory Hierarchy](image)

### 4.4 Network Architecture

A three-layer interconnection architecture consists of a star network, a ring network and a serial bus which is combined with the memory hierarchy. As shown in figure 4.3, a flexible interconnection architecture performs a effective communication between the on-chip memories and off-chip main memory.
• **Star Network** The datapath of Star network communication is shown in figure 4.3 which works for interconnection between main memory and SIMD local memories (PM, CM, LVM).

• **Ring Network** The purpose of Ring Network is to transfer data from one SIMD local LVM to another SIMD local LVM. The communication is reconnected by requesting from Master when interruption happens.

• **Serial Bus** Short messages in 16-bit size pass on Serial Network in registers.

![Figure 4.3: ePUMA Network Architecture](image)

4.5 **DMA**

Direct Memory Access (DMA) is an external device independent of master core and running in parallel with DSP. It controls the data transaction between memories so that allows more tasks could be executed in DSP simultaneously. Both DMA and DSP can access the data memory but priority depends on system design and methodology. In ePUMA system, DMA is controlled by master core to transmit data between main memory and level-2 memories or broadcast in SIMD slaves. Each DMA tasks works in a fixed channel including source address and destination address, which by multiply destination ports to achieve broadcast. For more reading, please take a look at [6].
4.6 Simulator

ePUMA simulator has been provided by ePUMA research team. The core of ePUMA simulator is wrapped by a Python class, ePUMASimulator, which has an instance of ePUMASystem class what is the real hardware model. A couple of scripts could be used to start simulation that several examples from reference [2] illustrate how to simulate on a single Sleipnir processor and in a full system.

A file with suffix ".sasm" will be interpreted as a Sleipnir assembly file and a suffix ".asm" refers to a master program. Run it with simdexec.py script by command `simdexec.py -t *.sasm` would start the single SIMD simulation quickly. For a project, `esim.py -m *.asm -k *.sasm -k *.sasm` is running for reading these input files and assembly files firstly, then links those assembly programs together. It supports single-step debugging, specified cycles running, run to the end and memories data display. Errors would be published if there are any faults in programs. Furthermore, a command-line interface script called ssim also can be used to simulate on single Sleipnir by `ssim.py -k *.sasm`. In a word, a basic simulation flow includes four steps:

1. Assemble source files to a simulation.
2. Initialize simulator with simulation.
3. Run simulation.
4. Print simulation results.
Chapter 5

Design and Implementation

This chapter illustrates how to design and implements algorithms of MPEG-2 video decoding on ePUMA. A test video stream composed of three frames with I, P, B types respectively is chosen, which comes from the software MPEG-2 player testbench. Although it is not a high definition video stream that only displays in 128*128 pixels, we emphasize on execute video decoder flow on ePUMA and evaluate the performance by the critical cases in the test video.

5.1 Slice-layer Parallelism

To improve the performance of video decoding by taking advantage of the multi-core processor, layer parallelism is proposed as the main method in this thesis. Depending on the hierarchy structure of MPEG-2 video stream, possible layer parallelisms includes GOP-layer, frame-layer, slice-layer, macroblock-layer and block-layer parallelism.

The slice-layer parallelism of MPEG-2 is chosen in this thesis for following reasons. Firstly, if GOP-layer or frame-layer parallelism is applied, a large mount of data is processed in one co-processor which results in a burden to the system. Secondly, in macroblock-layer parallelism data dependence problem occurs due to differential encode on motion vector, where previous motion vector is needed to reconstruct the current one. Thirdly, block-layer parallelism leads to a waste of time for their amount of communication between master and co-processor because of the small scale. Therefore, the slice-layer parallelism is the best choice for both reasonable workload in a co-processor and independent processing to each slice. [7]
There are already some experimental implementation by slice-layer parallelism in MPEG-2 on other multi-core platforms, for instance Sony PlayStation 3 and IBM QS20 in reference [1], that both of them are heterogeneous master-multi-SIMD processor. Sony PlayStation 3 has one master processor and 6 available co-processors while IBM QS20 has two master processors and 16 slaves. To compare the result, a single-core processor namely Intel Xeon has been included in the research. By 6 co-processors, Sony PlayStation 3 achieves a speedup of 1.587 over the Xeon and IBM QS20 obtains 1.557 speedup than Xeon. The IBM QS20 gets a speedup of 3.088 over Xeon with maximum 16 slaves, that this fact proves the feasibility of slice-layer parallelism.

Figure 5.1 provides the overall execution flow of slice-layer parallelism. In slice preparation, master takes the responsibility of data partition and issues a DMA task to transfer one slice data to a co-processor. All co-processors execute the same program for slice decoding in parallel and write back the result when they finish the programe.

5.2 Master Control Flow

Master programme not only controls the slice data partition, but also controls the execution of SIMDs and data transaction between LVMs and MMs. Figure 5.2 shows the execution flow controlled by master. After decoding the information above slice layer, the master sends one slice data from MM to DMA. Followed by DMA and LVM data transaction, the SIMDs start to decode a slice. Once one slice is done, SIMDs send the result to MM and clean themself. Then they receive new slices and continue.

Prolog and Epilog

Prolog aims to set up the environment before algorithms begin, by copying tables for VLD into LVM in advance and loading slice data into LVM. Epilog stage has the purpose of terminating the algorithms in the end, including copying the final result from LVM to MM and cleaning the LVM into zero for further work. The number of cycles in these two parts would vary for different size input data and output result, but are irrelevant with the computation performance.
5.2 Master Control Flow

Video Stream

Header

Slice Preparation

Start of slice

Macroblock Header

Motion Vector

Coded Block Pattern

Intra/non_intra block VLD

Inverse Scan

Inverse Quantization

Form Prediction

IDCT

Motion Compensation

All blocks in one MB end?

Yes

All MBs end?

Yes

Slice Organization

Master Processor

Decoded Video

SIMD Processors

Master Processor

Figure 5.1: Slice-layer Parallelism
Figure 5.2: Execution Flow Controlled by Senior
DMA

DMA plays an significant role in data transmission. The three bits of signal 0xab0 define that a is m0, b is m1 and the last one is DMA, as shown in table 5.1. When two of three LVMs are defined as m0 and m1 which means these two are working for kernel execution, while the other LVM is defined as DMA which means it could read external data and broadcast to others.

<table>
<thead>
<tr>
<th>Signal 0xab0</th>
<th>m0</th>
<th>m1</th>
<th>DMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x120</td>
<td>LVM0</td>
<td>LVM1</td>
<td>LVM 2</td>
</tr>
<tr>
<td>0x130</td>
<td>LVM0</td>
<td>LVM2</td>
<td>LVM 1</td>
</tr>
<tr>
<td>0x210</td>
<td>LVM1</td>
<td>LVM0</td>
<td>LVM 2</td>
</tr>
<tr>
<td>0x230</td>
<td>LVM1</td>
<td>LVM2</td>
<td>LVM 0</td>
</tr>
<tr>
<td>0x310</td>
<td>LVM2</td>
<td>LVM0</td>
<td>LVM 1</td>
</tr>
<tr>
<td>0x320</td>
<td>LVM2</td>
<td>LVM1</td>
<td>LVM 0</td>
</tr>
</tbody>
</table>

Table 5.1: Definition of DMA Control Signal

An example in list 5.1 illustrates how the three LVMs work for m0, m1 and DMA. By receiving 0x210, three LVMs are configured that LVM1 is m0, LVM0 is m1, and LVM2 is DMA now. Then the function to load the data from MM to DMA(LVM2) is called. After that, three LVMs are switched that LVM2 is m0 and then some kernels would be executed. According to the example, the result is saved in m1(LVM0), so that m1(LVM0) is switched as DMA for transmitting to MM by star network.

Listing 5.1: DMA Execution Example

```
//in SIMDa:
out 0x10 0x210 0 //LVM switch, m0->LVM1, m1->LVM0, dma->LVM2
call LD_STAR //load data from MM to DMA
5*nop
wait
10*nop

out 0x10 0x310 0 //LVM switch, m0->LVM2, m1->LVM0, dma->LVM1
call MPEG2DEC //Kernel execution, source in m0, result in m1

out 0x10 0x320 0 //LVM switch, m0->LVM2, m1->LVM1, dma->LVM0
call ST_STAR //send data from LVM0 to MM
5*nop
```
5.3 Kernel Implementation

5.3.1 Implementation on Master processor

As explained in chapter 4, master processor is responsible for data distribution. Due to the length of a slice is uncertain, a slice data is found out by searching current slice_start_code until next slice_start_code. Slice start_code lies in range from 0x101 to 0x1AF, which is defined as 32 bits. Then a slice data is sent to Sleipnir LVM for further decoding.

5.3.2 Implementation on Co-processor

Kernel implementation on co-processor is based on the principle of figure 5.1. Two loops of algorithms are identified that one is used to decode single block and the other works for decoding macroblocks in a slice. In this section, some detailed explanation to each kernel implementation is written by the sequential order in the figure. However, the bit-read and bit-shift functions have to be introduced first because of their significance in the implementation. Because the video decoding process could be seen as "reading" bits and "translating" bits, these two functions are utilized for a great number of times.

Bit Read and Shift

Bit-read function refers to read the bits in a word but not shift them, while bit-shift function works for reading the bits and shifting them as well. The method of reading fix-length bits in a word is introduced in an example, as shown in list 5.2. One global register (gb0.0) is used for counting how many bits are left in a word and another (gb0.1) for how many bits have been used in a word. Two registers (like vr7.2 and vr7.3) are used to save the pseudo bits number.

In the example, an input data pointer has been copied into a LVM address register ar0c in advance. At first, algorithm compares the gb0.0 with target number or target register to judge whether move to next word data or not, as shown in figure 5.3. For 5 bits example, if gb0.0>=5, the algorithm left shifts the bits number that have used firstly, and then right shifts 11 bits secondly, which comes from 16-5=11. The last it updates the pseudo count register vr7.2 and vr7.3.

If gb0.0<5, it adds address pointer to next, followed by right shifting (16-target) bits to m0[ar0].w, as part1 register shown. Besides, it right
5.3 Kernel Implementation

Figure 5.3: Bit-read and Bit-shift Function

shifts \((16-(\text{target bits}-\text{gb}0.0))\) to \(m0[\text{ar}0+1].w\) as \textbf{part2} register, which is \((16-(5-\text{gb}0.0)) = (11+\text{gb}0.0)\) bits in the example. In the end, it combines the two register \textbf{part1} and \textbf{part2} to construct the result by orw instruction. Not only \text{vr}7.2 and \text{vr}7.3 in the process have been updated in the end, but also the address pointer has to return by minus one, because it just reads the bits data but not shifts.

When the number of bits is uncertain but comes from a target register, figure 5.3 shows the algorithms for different situations. Bit-shift function works without two registers (\text{vr}7.2 and \text{vr}7.3) for saving pseudo bits number and doesn’t minus the address pointer by one at last.

**Listing 5.2: shift bits Implementation**

```assembly
shiftbit: cmpwq gb0.0 vr0.1  //vr0.1 contains the number of bits would shift
6*nop
jmp.sge bits_full
islwq vr1.3 m0[ar0+=1].w gb0.1
subwq gb0.1 vr0.1 gb0.0/\(6-5=1\)
3*nop
subwq gb0.0 16 gb0.1/\(16-1=15\)
subwq vr1.2 16 vr0.1/\(6\)
3*nop
lsrwq vr1.3 vr1.2
lsrwq vr1.4 m0[ar0-=1].w gb0.0
3*nop
orwq vr1.5 vr1.3 vr1.4  //dct_diff in vr1.5
3*nop
```
Slice Header

Start of Slice To decode a slice on each co-processor, the first work is decoding some slice start information, particularly quantization_scale_code. The 5-bit fix length code of quantization scale would used to inverse quantization afterward.

Parameter VLD The header of macroblocks contains several parameters in sequential, which go through VLD one by one. First of all, macroblock_address_increment indicates the difference between current and previous macroblock address. Followed by macroblock_modes what contains macroblock_type, motion_type and DCT_type. macroblock_modes indicates the method of coding and content of the macroblock, motion_type of two bits demonstrates the macroblock prediction type while DCT_type defines whether a frame DCT or not.

Motion Vector

Motion vectors could be separated into forward and backward MVs. As shown in figure B.2, firstly motion_vector_count decides whether the motion_type is field or frame, that for field-based prediction one MB would be separated into top and bottom two fields. Thus, field-based motion type specifies two MVs in one MB and frame-based motion type means one MV in a MB, which indicates the interlaced and progressive property of MPEG-2 standard. Furthermore, each MV is divided into horizontal component and vertical component. Therefore if it is field-based motion type with both forward and backward prediction, at most 8 PMVs will be produced in this process. The flag motion_vertical_field_select[s][t] indicates if the reference field should be top or bottom.
To each component, VLD of \textit{motion\_code} and \textit{motion\_residual} are taken at first. Because of differential coding process in encode as explained in section 3.2.5, the motion vector is reconstructed by previous one as shown in following detailed algorithm. That PMV refers to previous motion vector. In the end with the limitation $\lim=16 \times r\_size$, the result is saturated and scaled.

1. Temp\_PMV =
   \[
   \begin{cases}
   Previous\_PMV + ((motion\_code - 1) \ll r\_size) + motion\_residual + 1 & \text{motion\_code} > 0 \\
   Previous\_PMV - ((-motion\_code - 1) \ll r\_size) + motion\_residual + 1 & \text{motion\_code} < 0 \\
   Previous\_PMV & \text{motion\_code} = 0 
   \end{cases}
   \]

2. Temp2\_PMV =
   \[
   \begin{cases}
   Temp\_PMV - 2 \times lim & \text{motion\_code} > 0 \\
   Temp\_PMV + 2 \times lim & \text{motion\_code} < 0 \\
   Temp\_PMV & \text{when motion\_code} = 0 
   \end{cases}
   \]

3. Current\_PMV =
   \[
   \text{scalar?}(\text{Temp2\_PMV} \ll 1) : \text{Temp2\_PMV}
   \]

\textbf{Coded Block Pattern}

Macroblock Patterns will be produced by \textit{coded\_block\_pattern} VLD. Based on 4:2:0 sampling structure of video, some parameters are set in default on ePUMA. For example there are 6 blocks in one MB, counter block\_number is set from 0 to 5 that 0 to 3 referring to luminance blocks while 4,5 referring to chrominance blocks.

\textbf{Block VLD}

Regarding block VLD, the realization of VLD on single data is written at first. Implementation in the whole design are working in similar methods. Take VLD of DCT data in luminance block in the list 5.3 as an example. All the VLD tables have been written into LVMs in advance by addressing a word. Three variables mentioned before – run, level and length – could be set into one word of 16 bits where left most eight bits refer to run, following four bits mean level and last four bits are set to length. For instance the value 0xB18 in word format actually represents \text{run} = 11, \text{level} = 1 and \text{length} = 8. A plenty of configuration addressing modes are
written into CM as well.

List 5.3 shows how to realize VLD by accessing a table DCTtab0 in Sleipnir programing. After reading 16 bits to get code, address registers DCTtab0_CONF is copied from CM to LVM address registers firstly. By algorithm

\[ \text{tab} = \&DCTtab0[(\text{code} >> 8) - 4] \]

, the run, level, length value are accessed by shifting bits. Level with one more sign bit reconstructs the last non-zero result.

Listing 5.3: VLD Tables Configuration

```plaintext
.main
    // vr1.5 contains "code" value.
    dcopy ar0c cm[DCTtab0_CONF]
    3*nop

    lsrwq vr1.0 vr1.5 cm[8] // tab = &DCTtab0[(code>>8)-4];
    4*nop
    subwq vr1.1 vr1.0 cm[4]
    4*nop
    andwq<div=8> vr2.5 ml[ar1+vr1.1].w 0xFF00
    // high 8 bits is the run (vr2.5)
    andwq<div=4> vr2.6 ml[ar1+vr1.1].w 0xF0
    // next 4 bits is the level (vr2.6)
    andwq vr2.7 ml[ar1+vr1.1].w 0xF
    // low 4 bits is the length (vr2.7)
    3*nop

.ml
DCTtab0:
0x4106 0x4106 0x4106 0x4106 0x227 0x227 0x917 0x917
0x47 0x47 0x817 0x817 0x716 0x716 0x716 0x716
......
0x315 0x315 0x315 0x315 0 0 0 0
DCTtab0_END:

.cm
DCTtab0_CONF:
DCTtab0 DCTtab0 DCTtab0_END 0 0 0 0
```

As for block VLD, figure B.1 describes the execution flow. MPEG-2 intra and non_intra block VLD have been combined into one because of some common decoding parts. Depending on `picture_coding_type`, the intra or non_intra MPEG-2 decode function is decided in advance. In the MPEG-2 intra block decoding, at first DC value is decoded according to `block_number cc`, and `block_value_count` is reset to 0. Then following 16 bits is read to achieve `code` value for taking a branch to generate `tab` result in the next step. Thirdly, `Normal` process is performed by `tab`, until it faces 64 which notices End Of Block(EOB) and goes to next new
block VLD. In the end, whether it is the last block in one MB or not is determined by checking the block_number cc equals to 5 or not. Once finishing 6 blocks in a MB, the algorithm jumps to next procedure. MPEG-2 non_intra block VLD works in the similar way as intra block, just without DC decoding.

In the Normal process, a register counts from 0 to 63 by adding the value of run. Thus the value of this register indicates the address of non-zero pixel in 8*8 matrix, which is the reading pattern of inverse scan. Then the counter adds one as result of the non-zero value and algorithm shifts out the number of bits according to value of length and goes to next. Therefore, pixels are processed one by one in this part, which results in block VLD is one of the most time consuming kernels.

Inverse Scan

IS and IQ have been realized in the Normal process. According to different scan pattern in encoder, Inverse Scan (IS) works based on following read and write patterns. The Inverse Scan read pattern is:

\[
\begin{align*}
0 & \quad 1 & \quad 2 & \quad 3 & \quad 4 & \quad 5 & \quad 6 & \quad 7 \\
8 & \quad 9 & \quad 10 & \quad 11 & \quad 12 & \quad 13 & \quad 14 & \quad 15 \\
16 & \quad 17 & \quad 18 & \quad 19 & \quad 20 & \quad 21 & \quad 22 & \quad 23 \\
24 & \quad 25 & \quad 26 & \quad 27 & \quad 28 & \quad 29 & \quad 30 & \quad 31 \\
32 & \quad 33 & \quad 34 & \quad 35 & \quad 36 & \quad 37 & \quad 38 & \quad 39 \\
40 & \quad 41 & \quad 42 & \quad 43 & \quad 44 & \quad 45 & \quad 46 & \quad 47 \\
48 & \quad 49 & \quad 50 & \quad 51 & \quad 52 & \quad 53 & \quad 54 & \quad 55 \\
56 & \quad 57 & \quad 58 & \quad 59 & \quad 60 & \quad 61 & \quad 62 & \quad 63
\end{align*}
\]

According to scan pattern 0 in encoder, IS pattern 0:

\[
\begin{align*}
0 & \quad 1 & \quad 8 & \quad 16 & \quad 9 & \quad 2 & \quad 3 & \quad 10 \\
17 & \quad 24 & \quad 32 & \quad 25 & \quad 18 & \quad 11 & \quad 4 & \quad 5 \\
12 & \quad 19 & \quad 26 & \quad 33 & \quad 40 & \quad 48 & \quad 41 & \quad 34 \\
27 & \quad 20 & \quad 13 & \quad 6 & \quad 7 & \quad 14 & \quad 21 & \quad 28 \\
35 & \quad 42 & \quad 49 & \quad 56 & \quad 57 & \quad 50 & \quad 43 & \quad 36 \\
29 & \quad 22 & \quad 15 & \quad 23 & \quad 30 & \quad 37 & \quad 44 & \quad 51 \\
58 & \quad 59 & \quad 52 & \quad 45 & \quad 38 & \quad 31 & \quad 39 & \quad 46 \\
53 & \quad 60 & \quad 61 & \quad 54 & \quad 47 & \quad 55 & \quad 62 & \quad 63
\end{align*}
\]
According to scan pattern 1 in encoder, IS pattern 1:

\[
\begin{array}{cccccccccccc}
0 & 8 & 16 & 24 & 1 & 9 & 2 & 10 \\
17 & 25 & 32 & 40 & 48 & 56 & 57 & 49 \\
41 & 33 & 26 & 18 & 3 & 11 & 4 & 12 \\
19 & 27 & 34 & 42 & 50 & 58 & 57 & 49 \\
51 & 59 & 20 & 28 & 5 & 13 & 6 & 14 \\
21 & 29 & 36 & 44 & 52 & 60 & 57 & 49 \\
53 & 61 & 22 & 30 & 7 & 15 & 23 & 31 \\
38 & 46 & 54 & 62 & 39 & 47 & 55 & 63 \\
\end{array}
\]

### Inverse Quantization

IQ is divided into three steps which Inverse Quantization Arithmetic, Saturation and Mismatch Control as shown in figure 5.4. Inverse Quantization Arithmetic is a IQ matrix and a `quantiser_scale` are multiplied.

Arithmetic of intra block IQ is:

\[
val = (val \ast quantizer\_scale \ast qmat[j]) >> 4
\]

Arithmetic of non-intra block IQ is:

\[
val = (((val << 1) + 1) \ast quantizer\_scale \ast qmat[j]) >> 5
\]

`quantiser\_scale` is calculated according to the table between decoded `quantiser\_scale\_type` and `quantiser\_scale\_code`, that is defined in table 7-6 of [9]. When `quantiser\_scale\_type` is 1, the `quantiser\_scale` is non-linear and keeps mapping to following pattern. On the other hand, when `quantiser\_scale\_type` is 0, the linear value of `quantiser\_scale` is double of `quantiser\_scale\_code`.

Linear quantiser\_scale:

\[
quantizer\_scale = 2 \ast quantizer\_scale\_code
\]

Non-linear quantiser\_scale, `quantizer\_scale= pattern[quantiser\_scale\_code]`:

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
8 & 10 & 12 & 14 & 16 & 18 & 20 & 22 \\
24 & 28 & 32 & 36 & 40 & 44 & 48 & 52 \\
56 & 64 & 72 & 80 & 88 & 96 & 104 & 112 \\
\end{array}
\]

qmat matrix for intra or non\_intra blocks has been listed in chapter 3. qmat value is addressed by pixel address after inverse scan. For instance in
5.3 Kernel Implementation

5.3.1 Intra-Block Quantization

Intra block when \(i=3\) before inverse scan, it is \(j=24\) after inverse scan in IS pattern 1. Therefore the qmat value is \(q\text{mat}[24]= 22\). If \(\text{val}=1\) before IQ, result after IQ should be \((1\times20\times24)\gg4=27\). If block is non-intra, the qmat value is 16 by the principle in section 3.2.2.

Saturation aims to control the value saturated to a certain yield. Mismatch Control reconstructs a final DCT coefficients at \(i[63]\). If the sum of coefficients in a block is even, a correction to \(i[63]\) shall be made.

\[
i[63] = \begin{cases} 
\text{Previous}_i[63] & \text{for sum is odd} \\
\text{Previous}_i[63] - 1 & \text{if Previous}_i[63] \text{ is odd and sum is even} \\
\text{Previous}_i[63] + 1 & \text{if Previous}_i[63] \text{ is even and sum is even}
\end{cases}
\]

Figure 5.4: Inverse Quantization Execution Flow on Sleipnir

Example of intra_block in the list C.1 has explained how to inverse scan and inverse quantization for each pixel result after VLD. Here quantizer_scale value has been decoded and stored in vr2.3. At last, result with the sign bit is stored into the LVM at IS address. EOB process includes mismatch control, reset the count and decides whether current block is the end of one MB.

Form Prediction

Prediction is used to read reference pixels to current pixels by corresponding MVs, which all the MVs are specified to half-per sample grid. However, due to limited working time, they are all treated as non half-per prediction.

According to address of the slice in the frame, the 16 pixels around the current slice are loaded into LVM in advance for reference, which is named Reference Space here. As shown in figure 5.5, two-dimensional pixels in luminance block is ordered to the new linear pixel co-ordinates by following algorithm:

\[
s = src + lx \times (y + yint) + x + xint + field?lx2 >> 1 : 0
\]
src is the start memory address of reference space. MV is scaled by \( y_{\text{int}} = \text{vec}_y \gg 1 \), \( x_{\text{int}} = \text{vec}_x \gg 1 \), while \((x, y)\) is the co-ordinate of top-left sample in current MB. For example \((16, 32)\) is the co-ordinate of MB with \(\text{count} = 2\), which starts from 0 in current slice. \(lx\) is the residual value between previous line and current line, \(lx_2\) is twice of coded picture width. For frame-based prediction \(lx = \text{coded\_picture\_width}\) while for field-based prediction \(lx = \text{coded\_picture\_width} \times 2\). Therefore, prediction pixels to the non half-pixel calculation is read as list 5.4.

**Listing 5.4**: Pseudo Code of Form Prediction

```c
for (j=0; j<h; j++)
    {for (i=0; i<w; i++)
        { d[i] = s[i];}
    s+= lx_2; //s is the source address
    d+= lx_2;} //d is destination address
```

Normally \(h=8\), \(w=16\) are set as result of top and bottom field. Because of half-sampled chrominance data, the MVs would be halved to form the prediction of chrominance as well as \(lx\). Thus, \(x_{\text{int}} = (\text{vec}_x / 2) \gg 1\), \(y_{\text{int}} = (\text{vec}_y / 2) \gg 1\), the algorithm in chrominance is
\[ s = src + lx \cdot (y/2 + yint) + x/2 + xint + field?lx2 >> 1 : 0 \]

Even though in frame-based prediction there is just one MV in the MB, the one MV could be considered as two equal MVs in top and bottom fields of MB. Thus, the frame-based prediction could be reordered like field-based prediction in order to increase the flexibility by reading the pixels following field-based pattern.

The reason \( lx \) in field-based prediction is double of \( lx \) in frame-based prediction is interlaced property, which is double size of picture width. Because of reorder in frame-based prediction, the \( lx2 \) would be 256 in Y blocks all the time. Figure 5.6 explains the order in memory for both frame and field predictions.

Figure 5.6: Order in Memory to Frame or Field Based Prediction
Fast IDCT

Fast IDCT works on 8 * 8 pixels block. It contains two steps that **Horizontal IDCT** and **Vertical IDCT**, as shown in figure 5.7. Because of multibank organization of LVMs, the operations are based on a vector of 8 pixels. Furthermore, the efficient instruction `tmacw` performs a triangular MAC operation to word, which one vector of each row is multiplied with coefficients in CM and the result is accumulated.

![Diagram of IDCT Execution Flow](image)

(a) IDCT Execution Flow on ePUMA      (b) IDCT Execution Flow in Other Implementation

Figure 5.7: IDCT Execution Flow

Other hardware implementation usually takes four steps that Horizontal IDCT -> Transpose -> Vertical IDCT -> Transpose. But thanks to variety of addressing modes in Sleipnir, the reading patterns as `m0[ar0+=8%].v` and `vrf[rar1+=8].v` reduces the transpose stage and no latency is required by using registers, as shown in list 5.5.

<table>
<thead>
<tr>
<th>Listing 5.5: IDCT to Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>// First multiplication</td>
</tr>
<tr>
<td>repeat 2 8</td>
</tr>
<tr>
<td>7 * tmacw&lt;mul=1,ss,rnd,sat,clr&gt; vrf[rar0+=1].w cm[car0] m0[ar0+=8%].v</td>
</tr>
<tr>
<td>7 * tmacw&lt;mul=1,ss,rnd,sat,clr&gt; vrf[rar0+=1].w cm[car0+=1%] m0[ar0+=8%].v</td>
</tr>
<tr>
<td>// Second multiplication</td>
</tr>
<tr>
<td>repeat 2 8</td>
</tr>
<tr>
<td>7 * tmacw&lt;mul=1,ss,rnd,sat,clr&gt; m0[ar0+=1%].w cm[car0] vrf[rar1+=8].v</td>
</tr>
<tr>
<td>7 * tmacw&lt;mul=1,ss,rnd,sat,clr&gt; m0[ar0+=1%].w cm[car0+=1%] vrf[rar1+=8].v</td>
</tr>
</tbody>
</table>

According to field-based and frame-based DCT in coding process, the result is in the different order in memory, which means in field-based IDCT the block 0 and 1 contain top field IDCT result, while block 2 and 3 include bottom field IDCT result.
5.3 Kernel Implementation

Motion Compensation
To the intra picture, there is no prediction value so that motion compensation only adds value 128 to each pixels. As it is always frame-based IDCT, there is no change in the order.

To the non_intra prediction, the field and frame prediction methods and IDCT types result in different addition algorithms. Some reordering to those lines of pixels only happens in field-based IDCT. In the end, some rearrangement is acquired for reconstruction.

Transfer and compress
The two-dimensional result should be transferred into one-dimensional or linear in memory, according to display sequence that from top to bottom and from left to right. Figure 5.8 shows an example of sequential order in memory is the first rows of block 0 and block 1 in MB0 - MB2. Thus the transfer is required to store the data of blocks in selected address of memory.

As shown in list 5.6, first of all the source pointer ar0 takes the step of 8, and the pointer ar1 of linear data takes the step of 128, which is the coded picture width. After transfer of block 0, the destination pointer ar1 jumps to 8+bot1, which is the beginning of block 1. Then the ar0 moves to next block 0 of next MB by adding 256. repeat instruction operates to next 12 instruction for 8 times to 8 MBs.
Moreover, because final data lies from \(-2^{255}\) to \(2^{255}\), the 16 bits of each pixel could be compressed. That means one vector of 8 pixels is converted into half-vector, by taking the lower 8 bits of each word. Observe that in the end of decoding each slice, LVMs in each SIMD would be reset. Result of luminance to one slice is compressed into 128 vectors from 256 vectors, and chrominance result is compressed into 32 vectors from 64 vectors.
Chapter 6

Results and Discussion

6.1 Execution Time

The execution time of single kernel is illustrated in table 6.1. As explained in chapter 6, the VLD, IS and IQ of a block have been combined in one process and it costs 2115 cycles to one block in I-frame and 1945 cycles in P-frame. Execution cycles of this process is a little less in P-frame because it excludes DC value of VLD.

Decoding parts including MV, coded block pattern and form prediction only happens in P-frame, and they take 470 more clock cycles. The reason why MB header is much more time consuming in P-frame than in I-frame, is that some extra information decoding is needed for non_intra blocks.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>I-frame</th>
<th>P-frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB header</td>
<td>11 cycles</td>
<td>48 cycles</td>
</tr>
<tr>
<td>MV Decoding</td>
<td>___</td>
<td>287 cycles</td>
</tr>
<tr>
<td>Coded block pattern</td>
<td>___</td>
<td>19 cycles</td>
</tr>
<tr>
<td>Block VLD_IS_IQ</td>
<td>2115 cycles</td>
<td>1945 cycles</td>
</tr>
<tr>
<td>Form Prediction</td>
<td>___</td>
<td>141 cycles</td>
</tr>
<tr>
<td>IDCT</td>
<td>140 cycles</td>
<td>140 cycles</td>
</tr>
<tr>
<td>Motion Compensation</td>
<td>9</td>
<td>27 cycles</td>
</tr>
<tr>
<td>Transfer</td>
<td>20 cycles</td>
<td>20 cycles</td>
</tr>
<tr>
<td>Compress</td>
<td>10 cycles</td>
<td>10 cycles</td>
</tr>
<tr>
<td>Total</td>
<td>2305 cycles</td>
<td>2637 cycles</td>
</tr>
</tbody>
</table>

Table 6.1: Execution time of Each Kernel
Significantly, IDCT only costs 140 cycles for a block of 8*8 pixels. Transfer and Compress costs a few cycles as well. Multibank organization of LVMs and instruction operation on vectors are the main reasons of such effective kernels.

To sum up, it spends 2305 and 2637 cycles for a block in I- and P-type pictures respectively excluding the prolog, epilog and DMA transaction. The proportion of time consumption in the table 6.2 depicts that block VLD_IS_IQ accounts for 72% and 65%, and IDCT occupies 25.14% and 21.24%, in I- and P-frame separately.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>I-frame</th>
<th>P-frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB header</td>
<td>0.29%</td>
<td>1.33%</td>
</tr>
<tr>
<td>MV Decoding</td>
<td>_</td>
<td>10.88%</td>
</tr>
<tr>
<td>Coded block pattern</td>
<td>_</td>
<td>0.72%</td>
</tr>
<tr>
<td>Block VLD_IS_IQ</td>
<td>91.34%</td>
<td>73.75%</td>
</tr>
<tr>
<td>Form Prediction</td>
<td>_</td>
<td>5.34%</td>
</tr>
<tr>
<td>IDCT</td>
<td>6.07%</td>
<td>5.34%</td>
</tr>
<tr>
<td>Motion Compensation</td>
<td>0.39%</td>
<td>1.02%</td>
</tr>
<tr>
<td>Transfer</td>
<td>0.61%</td>
<td>0.51%</td>
</tr>
<tr>
<td>Compress</td>
<td>1.30%</td>
<td>1.11%</td>
</tr>
</tbody>
</table>

Table 6.2: Execution Proportion of Each Kernel

### 6.2 Kernel Cost

Different instruction costs to each single kernel are shown in table 6.3. The most cost is Block VLD_IS_IQ kernel, followed by MV decoding kernel. The reason why Block VLD_IS_IQ consumes the most is that different branches are required for intra and non_intra blocks, and for luminance and chrominance DC value. MV decoding includes not only VLD of \textit{motion_code} and \textit{motion_residual} but also the calculation of differential decoding. Therefore this one is much more consuming than others.

Kernels consuming the least cost are IDCT and compress, because of the effective addressing modes and operations based on vector.
### 6.3 Comparison

List 6.1 shows the results of a block after Block VLD_IS_IQ. Comparing with results in MPEG-2 Player, their consistency proves that this process works equally well in the functional aspect.

#### Listing 6.1: Result after Block VLD_IS_IQ

```plaintext
// frame 0, MB 0
// Result after Block VLD_IS_IQ (MPEG-2 Player):
32 20 23 27 32 -33 0 0
0 20 0 0 0 0 0 0
0 0 0 0 36 0 0 0
27 0 0 33 0 0 0 0
0 0 0 0 0 0 0 0
0 -33 36 0 87 0 -60 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1

// Result after Block VLD_IS_IQ (Sleipnir):
32 20 23 27 32 -33 0 0
0 20 0 0 0 0 0 0
0 0 0 0 36 0 0 0
27 0 0 33 0 0 0 0
0 0 0 0 0 0 0 0
0 -33 36 0 87 0 -60 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1
```

List 6.2 shows the results after IDCT. The results produced from MPEG-2 Player and Sleipnir are shown in list 6.2. The IDCT process performs a little different from the MPEG-2 Player because of the finite data precision on a DSP processor.
### Listing 6.2: Result after IDCT

<table>
<thead>
<tr>
<th>Input data:</th>
<th>[32 \ 20 \ 23 \ 27 \ 32 \ -33 \ 0 \ 0 | 32 \ 20 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 | 27 \ 0 \ 0 \ 33 \ 0 \ 0 \ 0 \ 0 | 0 \ 0 \ 0 \ 0 \ 36 \ 0 \ 0 \ 0 | 0 \ -33 \ 36 \ 0 \ 87 \ 0 \ -60 \ 0 | 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 | 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Result after IDCT (MPEG-2 Player with coefficients scale 2^{16})</strong></td>
<td>38 \ 4 \ -35 \ 7 \ 28 \ -15 \ -9 \ 18</td>
</tr>
<tr>
<td></td>
<td>14 \ 18 \ 35 \ -7 \ 0 \ 26 \ -19 \ -16</td>
</tr>
<tr>
<td></td>
<td>10 \ 13 \ -2 \ 1 \ 5 \ -12 \ -7 \ 13</td>
</tr>
<tr>
<td></td>
<td>9 \ 7 \ -32 \ -2 \ 8 \ -26 \ -1 \ 20</td>
</tr>
<tr>
<td></td>
<td>4 \ 17 \ 26 \ -21 \ -5 \ 36 \ -6 \ -24</td>
</tr>
<tr>
<td></td>
<td>24 \ 17 \ 3 \ -4 \ 19 \ 28 \ 6 \ 4</td>
</tr>
<tr>
<td></td>
<td>30 \ 2 \ -44 \ 13 \ 34 \ -20 \ 8 \ 43</td>
</tr>
<tr>
<td></td>
<td>-2 \ -4 \ 6 \ -7 \ -1 \ 1 \ -22 \ 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result after IDCT (Sleipnir with scale 2^{13})</th>
<th>38 \ 3 \ -35 \ 7 \ 28 \ -13 \ -8 \ 18</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14 \ 18 \ 36 \ -6 \ 0 \ 26 \ -18 \ -16</td>
</tr>
<tr>
<td></td>
<td>10 \ 14 \ -1 \ 1 \ 5 \ -11 \ -7 \ 13</td>
</tr>
<tr>
<td></td>
<td>9 \ 7 \ -31 \ -3 \ 8 \ -26 \ 0 \ 20</td>
</tr>
<tr>
<td></td>
<td>3 \ 17 \ 26 \ -21 \ -5 \ 36 \ -7 \ -24</td>
</tr>
<tr>
<td></td>
<td>24 \ 17 \ 3 \ -4 \ 19 \ 28 \ 7 \ 4</td>
</tr>
<tr>
<td></td>
<td>30 \ 3 \ -41 \ 14 \ 34 \ -19 \ 8 \ 43</td>
</tr>
<tr>
<td></td>
<td>-2 \ -4 \ 5 \ -7 \ -2 \ 1 \ -21 \ 2</td>
</tr>
</tbody>
</table>

### 6.4 Analysis

In ePUMA, slice-parallelism is performed by eight SIMDs so that SIMDs start to decode new slices when they have finished previous one, as shown in figure 6.1. Therefore for I-frame it achieves $2305 \times 4 \times 64 / 8 = 73 760$ cycles per frame, for P-frame it gives $2637 \times 4 \times 64 / 8 = 84 384$ cycles per frame, with eight co-processors by following equation.

$$\text{Execution\_Cycles} = \frac{\text{clk\_per\_block} \times \text{blocks\_per\_MB} \times \text{MB\_per\_frame}}{\text{Numbers\_of\_processors}}$$

From the results of execution time and kernel cost, the most time and kernel cost consuming one is Block VLD process, and the least time and kernel cost consuming one is IDCT process. The algorithm of VLD decides the structure of the implementation. It consists of numerous branches, which could not be implemented on vector format or operated on eight
words at the same time.

Consequently, compared with IDCT, the VLD with low efficiency is not suitable to be implemented in SIMD co-processors. In the reference [5], a CABAC hardware acceleration is proposed to ease the heavy algorithms. Context-Adaptive Binary Arithmetic Coding (CABAC) is a form of entropy encoding used in H.264 video encoding. In that implementation, different options of hardware and instruction level optimization are discussed. One of the optimizations is program flow through a specialized branching operation. Another one is a two-bit parallel accelerator with hardware sharing between JPEG 2000 and H.264 encoder. If such a hardware acceleration could be realized on ePUMA, which is independent of Sleipnir, the performance of execution time and kernel cost would be improved.

6.5 Discussion

Based on our implementation, the result of Full HD videos, that 1920*1080 pixels video with 8100 MB per frame, in real-time 30 frames per second were expected. To evaluate the performance of video decoding on master-multi-SIMD structure system, a calculation equation is made as following.

\[
\text{Required\_Clock\_Frequency} = \frac{\text{cycles\_per\_block} \times \text{blocks\_per\_MB} \times \text{MB\_per\_frame} \times 30 \text{FPS}}{}
\]

Therefore, without parallelism, required clock frequency on ePUMA of Block VLD is \(2115 \times 4 \times 8100 \times 30 = 2,055,780,000\) Hz, and required clock frequency of two-dimensional IDCT of a block is \(140 \times 4 \times 8100 \times 30 = 136,080,000\) Hz. If 8 co-processors are working in parallel, the equation becomes:

\[
\text{Required\_Clock\_Frequency} = \frac{\text{cycles\_per\_block} \times \text{blocks\_per\_MB} \times \text{MB\_per\_frame} \times 30 \text{FPS}}{\text{Numbers\_of\_processors}}
\]

Obviously, a gradually linear decrease in execution time would be achieved with increasing number of processors. Then the VLD and IDCT requires 256,972,500 and 17,010,000 cycles per second over eight co-processors. For I frame, required clock frequency is \(2305 \times 4 \times 8100 \times 30 / 8 = 280,057,500\), which approximately is 280 MHz. For P frame, required clock frequency is \(2637 \times 4 \times 8100 \times 30 = 320,395,500\), which is about 320 MHz. Figure 6.1 shows slice-parallelism of video decoding on ePUMA.
Figure 6.1: Slice-Parallelsim of Video Decoding on ePUMA
Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis not only illustrates video structure from sequence (top) to block (bottom), but also explains the algorithms and software structure of MPEG-2 video decoding. Some brief introduction to ePUMA platform make it more understanding for audience to master-multi-SIMDs structure.

Based on the slice-parallelism methodology over eight co-processors of ePUMA, the implementation of the algorithms consists of variable length decoding, inverse scan, inverse quantization, two-dimensional inverse discrete cosine transform, motion vector decoding, form prediction and motion compensation. These implementation of algorithms are benchmarked on ePUMA simulator. The result shows that to decode real-time Full HD (1920*1080 pixels, 30 frames per second) video, it will require ePUMA to run at 280 MHz for I frames and at 320MHz for P frames.

There is no denying that some parts could be optimized in the design. Comparing with IDCT execution, Block VLD takes much more time because amount of utilization of read bit and shift bit function and branch work. Considering a word as an unit when read input bits in my implementation proves that an improvement of execution time could focus on this problem. One possible solution is taking one vector 128 bits as an unit by two bits counter and a data pointer. That could reduce some execution time by decreasing the number of data pointer shifting to next vector. Another possible solution is branch optimization in instruction level.
7.2 Suggestion

Some suggestions for ePUMA improvement are written.

**Constant Memory**

Only vector-addressable in CM not only leads to waste of memory if just one word value is asked, but also can not provide single word to be addressed for reading and writing. The solution is changing the address modes of CM.

**Local Vector Memory**

One read port and one write port cause some copy movement from LVM to VRF, if operation works on two LVMs at the same time. The possible solution is adding one more port.

7.3 Future Work

There are many interesting work could be investigated in the future. Because of limited time, the algorithm of B frame decoding haven’t finished yet, which involves frame reorder and buffer control. In addition, for high definition video it proves a challenging for more advanced coding technologies like H.264.
Bibliography


Appendix A

Appendix of ePUMA

Figure A.1: Pipeline Templates of Sleipnir[6]
Figure A.2: Pipeline of Sleipnir[6]
<table>
<thead>
<tr>
<th>Mode#</th>
<th>Index</th>
<th>Offset</th>
<th>Pattern</th>
<th>Syntax example</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>arX</td>
<td>0</td>
<td>0,1,2,3,4,5,6,7</td>
<td>[ar0]</td>
</tr>
<tr>
<td>1</td>
<td>arX</td>
<td>0</td>
<td>cm[carX]</td>
<td>[ar0 + cm[car0]]</td>
</tr>
<tr>
<td>2</td>
<td>arX</td>
<td>0</td>
<td>cm[imm8]</td>
<td>[ar0 + cm[10]]</td>
</tr>
<tr>
<td>3</td>
<td>arX</td>
<td>0</td>
<td>cm[carX + imm8]</td>
<td>[ar0 + cm[car0 + 10]]</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>vrX.Y</td>
<td>0,1,2,3,4,5,6,7</td>
<td>vr0.0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>vrX.Y</td>
<td>cm[carX]</td>
<td>vr0.0 + cm[car0]</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>vrX.Y</td>
<td>cm[imm8]</td>
<td>vr0.0 + cm[10]</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>vrX.Y</td>
<td>cm[carX + imm8]</td>
<td>vr0.0 + cm[car0 + 10]</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>vrX</td>
<td>vr0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>cm[carX]</td>
<td>cm[car0]</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>cm[imm8]</td>
<td>cm[10]</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>cm[carX + imm8]</td>
<td>cm[car0 + 10]</td>
</tr>
<tr>
<td>12</td>
<td>arX</td>
<td>0</td>
<td>vrX</td>
<td>ar0 + vr0</td>
</tr>
<tr>
<td>13</td>
<td>arX</td>
<td>vrX.Y</td>
<td>0,1,2,3,4,5,6,7</td>
<td>ar0 + vr0.0</td>
</tr>
<tr>
<td>14</td>
<td>arX</td>
<td>imm16</td>
<td>0,1,2,3,4,5,6,7</td>
<td>ar0 + 1024</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>imm16</td>
<td>0,1,2,3,4,5,6,7</td>
<td>1024</td>
</tr>
</tbody>
</table>

Table A.1: LVM Addressing Modes of Sleipnir
<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVM address</td>
<td>ar0</td>
<td>register 0</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>bot0</td>
<td>bottom register 0</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>top0</td>
<td>top register 0</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>step0</td>
<td>step register 0</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>ar0c</td>
<td>a half-vector composed of ar0, bot0, top0 and step0</td>
<td>4x15/16 bits</td>
</tr>
<tr>
<td>LVM address</td>
<td>ar1</td>
<td>register 1</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>bot1</td>
<td>bottom register 1</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>top1</td>
<td>top register 1</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>step1</td>
<td>step register 1</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>ar1c</td>
<td>a half-vector composed of ar1, bot1, top1 and step1</td>
<td>4x15/16 bits</td>
</tr>
<tr>
<td>LVM address</td>
<td>ar2</td>
<td>register 2</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>bot2</td>
<td>bottom register 2</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>top2</td>
<td>top register 2</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>step2</td>
<td>step register 2</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>ar2c</td>
<td>a half-vector composed of ar2, bot2, top2 and step2</td>
<td>4x15/16 bits</td>
</tr>
<tr>
<td>LVM address</td>
<td>ar3</td>
<td>register 3</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>bot3</td>
<td>bottom register 3</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>top3</td>
<td>top register 3</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>step3</td>
<td>step register 3</td>
<td>15/16 bits</td>
</tr>
<tr>
<td></td>
<td>ar3c</td>
<td>a half-vector composed of ar3, bot3, top3 and step3</td>
<td>4x15/16 bits</td>
</tr>
<tr>
<td>CM address</td>
<td>car0</td>
<td>register 0</td>
<td>7/8 bits</td>
</tr>
<tr>
<td></td>
<td>cbot0</td>
<td>bottom register 0</td>
<td>7/8 bits</td>
</tr>
<tr>
<td></td>
<td>ctop0</td>
<td>top register 0</td>
<td>7/8 bits</td>
</tr>
<tr>
<td></td>
<td>car0c</td>
<td>a half-vector composed of car0, cbot0, ctop0 and <em>empty</em></td>
<td>4x7/8 bits</td>
</tr>
<tr>
<td>CM address</td>
<td>car1</td>
<td>register 1</td>
<td>7/8 bits</td>
</tr>
<tr>
<td></td>
<td>cbot1</td>
<td>bottom register 1</td>
<td>7/8 bits</td>
</tr>
<tr>
<td></td>
<td>ctop1</td>
<td>top register 1</td>
<td>7/8 bits</td>
</tr>
<tr>
<td></td>
<td>car1c</td>
<td>a half-vector composed of car1, cbot1, ctop1 and <em>empty</em></td>
<td>4x7/8 bits</td>
</tr>
<tr>
<td>Repeat register</td>
<td>rreg0</td>
<td>with <code>repeat</code> instruction</td>
<td>16 bits</td>
</tr>
<tr>
<td>Loop register</td>
<td>lc0</td>
<td>loop counter</td>
<td>16 bits</td>
</tr>
<tr>
<td>Master register</td>
<td>mask0</td>
<td>for conditional read and write</td>
<td>8 bits</td>
</tr>
<tr>
<td>Global register</td>
<td>gb0.0</td>
<td>for top-level communication</td>
<td>16 bits</td>
</tr>
<tr>
<td></td>
<td>gb0.1</td>
<td>not for kernels computations</td>
<td>16 bits</td>
</tr>
<tr>
<td></td>
<td>gb0.2</td>
<td></td>
<td>16 bits</td>
</tr>
<tr>
<td></td>
<td>gb0.3</td>
<td></td>
<td>16 bits</td>
</tr>
<tr>
<td>Coefficient register</td>
<td>coeff0</td>
<td>some coefficients</td>
<td>8x16 bits</td>
</tr>
</tbody>
</table>

Table A.2: Special Register Definition for LVM addressing
Appendix B

Figures of Some Implementation

B.1 Intra and non_intra Block VLD on ePUMA
B.2 Motion Vector Decoding on ePUMA
Figure B.1: Intra and Non_intra Block VLD on ePUMA
B.2 Motion Vector Decoding on ePUMA

Figure B.2: Motion Vector Decoding on ePUMA
Appendix C

Lists of Some Implementation

Listing C.1: List of VLD, Inverse scan and Inverse Quantization

cmpwq vr2.5 65
    cmpwq vr2.5 64
    5*nop
    jmp.eq escape
    jmp.eq eob

normal:addwq vr7.5 vr7.5 vr2.5 //vr2.5 is run of length of zeros
    cmpwq vr2.7 gb0.0
    6*nop
    jmp.sge bit_solu //bit solution
    subwq gb0.0 gb0.0 vr2.7
    addwq gb0.1 gb0.1 vr2.7 //update bit count
    subwq ar0 ar0 1
    3*nop
    jmp bit_solu_end

    bit_solu: subwq gb0.1 vr2.7 gb0.0
    3*nop
    subwq gb0.0 16 gb0.1
    3*nop

bit_solu_end://next1 bits

    cmpwq gb0.0 1 //next1bits
    6*nop
    jmp.sge next_sign
    addwq ar0 ar0 1
    dcopy gb0.0 15
    dcopy gb0.1 1
    nop
    lsrwq vr0.2 m0[ar0].w 15
    3*nop
    jmp next_sign_end
next_sign:  lslwq vr0.0 m0[ar0].w gb0.1
3*nop
lsrwq vr0.2 vr0.0 15
subwq gb0.0 gb0.0 1
addwq gb0.1 gb0.1 1
next_sign_end:
dcopy ar1c cm[WRITE1_CONF]
dcopy ar3c cm[QMAT_CONF]
2*nop
dcopy vr0.7 m1[ar1+vr7.5].w //vr0.7=j, vr7.5=i, inverse scan
mulwww<uu,mul=16> vr0.5 vr2.6 vr2.3 //val*quantiser_scale
6*nop
mulwww<uu,mul=12> vr2.6 vr0.5 m1[ar3+vr0.7].w //(val*quantiser_scale* 
    qmat)>>4
4*nop
cmpwq vr0.2 1
6*nop
jmp.eq sign
copyq m1[ar2+vr0.7].w vr2.6
6*nop
addwq vr7.7 vr7.7 vr2.6 //sum
addwq vr7.5 vr7.5 1 //add one, non-zero value
3*nop
jmp judge1

sign:  subwq vr0.6 0 vr2.6
3*nop
copyq m1[ar2+vr0.7].w vr0.6
6*nop
addwq vr7.7 vr7.7 vr0.6 //sum
3*nop
addwq vr7.5 vr7.5 1 //add one, non-zero value
3*nop
judge1:
cmpwq m0[picture_coding_type].w 1
6*nop
jmp.eq mpeg2_intra_ac
jmp dct_block //get next_16_bits
// ================

escape: ...
// ================
eob:  addwq gb0.3 gb0.3 1
      cmpwq vr2.7 gb0.0
6*nop
jmp.sge bit_solu_e //bit solution
subwq gb0.0 gb0.0 vr2.7
addwq gb0.1 gb0.1 vr2.7 //update bit count
subwq ar0 ar0 1
3*nop
jmp bit_solu_end_e
bit_solu_e:
subwq gb0.1 vr2.7 gb0.0
2*nop
andwq vr7.0 vr7.7 1
subwq gb0.0 16 gb0.1
2*nop
bit_solu_end_e:
   //mismatch control
   cmpwq vr7.0 0
   6*nop
   jmp.ne sum_odd
   //sum_even
   xorwq vr7.0 m1[ar2+63].w 1
   3*nop
   copyq m1[ar2+63].w vr7.0
   6*nop
sum_odd:
   addwq ar2 ar2 64 //fill up 64 numbers
   dcopy vr7.5 0 //clean value count
   dcopy vr2.5 0
   dcopy vr7.7 0 //clean sum
   //3*nop
   cmpwq gb0.3 6 //block counter
   6*nop
   jmp.eq end_mb_vld

   cmpwq m0[picture_coding_type].w 1
   6*nop
   jmp.eq mpeg2_intra_ac
   //jmp mpeg2_intra //if it is intra block( picture_coding_type=1 ),
   jump to mpeg2_intra
   jmp dct_block //get next_16_bits