Implementation and Evaluation of a Video Decoder on the Coreworks Platform

Edgar Fernandes da Costa Marques

Thesis to obtain the Master of Science Degree in
Electrotechnical and Computers Engineering

Supervisor(s): Prof. Nuno Filipe Valentim Roma
Prof. Paulo Ferreira Godinho Flores

Examination Committee
Chairperson: Prof. Gonçalo Nuno Gomes Tavares
Supervisor: Prof. Paulo Ferreira Godinho Flores
Member of the Committee: Prof. Fernando Manuel Duarte Gonçalves

May 2016
Resumo

A codificação de vídeo é uma área de investigação em constante desenvolvimento, aumentando de forma consistente a eficiência dos codecs de vídeo e a qualidade de imagem conseguida pelos mesmos. Implementações em hardware dedicado ou embregido são um alvo importante destes codecs, obtendo-se em geral melhores resultados a nível de tempo de processamento e de consumo de energia em contraste com implementações em software. Thor é um codec de vídeo recentemente lançado e que servirá como base para um projecto da Internet Engineering Task Force (IETF) com o objectivo de desenvolver uma nova norma de compressão de vídeo, livre de licenças.

A plataforma desenvolvida pela Coreworks é formada por um processador embregido que funciona em sintonia com aceleradores implementados em hardware. Esta plataforma tem sido usada fundamentalmente para soluções de áudio, no entanto, pretende-se explorar a sua utilização para outras aplicações complexas como a descodificação de vídeo.

O trabalho desenvolvido nesta tese envolve a implementação do descodificador do Thor e a sua subsequente aceleração na plataforma da Coreworks. Para tal, o descodificador de vídeo do Thor foi portado para a plataforma. Para a sua aceleração, foram implementadas instâncias de Sideworks, que consiste em aceleradores implementados em hardware. Usando as ferramentas disponibilizadas pela Coreworks, foram acelerados os blocos da transformada inversa e da quantização inversa. Após uma implementação desta plataforma numa FPGA, conseguiu-se uma aceleração até 1,43 para o bloco da transformada inversa quando comparado com a sua implementação no processador embregido.

**Palavras-chave:** descodificador de vídeo do Thor, plataforma embregida, aceleradores implementados em hardware, transformada inversa
Abstract

Video encoding is a research area under continuous development, with each newly introduced video codec consistently outperforming the previous ones. Dedicated or embedded hardware implementations are a big target of these codecs, generally allowing improvements on processing time and power consumption, in comparison with software implementations. Thor video codec was recently introduced as part of the project organized by the Internet Engineering Task Force (IETF) to develop a next-generation royalty free video codec.

The platform developed by Coreworks has been mainly used for audio solutions and consists of an embedded processor working together with instances of hardware accelerators. However, we intend to explore its usage for other complex applications such as video decoding.

This thesis presents an implementation and subsequent acceleration of Thor’s video decoder on Corework’s embedded platform. This requires porting the targeted video decoder to the platform, followed by the design of reconfigurable hardware accelerators, referred as Sideworks instances. Using Corework’s tools, the Dequantize and Inverse Transform stages of the decoder are accelerated through the design and implementation of hardware accelerators. By implementing this platform in a FPGA, we were able to accelerate up to 1.43 the Inverse Transform in comparison with its implementation in the embedded processor.

Keywords: Thor video decoder, embedded platform, hardware accelerators, Inverse Transform
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resumo</td>
<td>iii</td>
</tr>
<tr>
<td>Abstract</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>Nomenclature</td>
<td>xiii</td>
</tr>
<tr>
<td>Glossary</td>
<td>xiii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>2 Video Compression</strong></td>
<td>3</td>
</tr>
<tr>
<td>2.1 Video Coding</td>
<td>3</td>
</tr>
<tr>
<td>2.1.1 Colour Compression</td>
<td>4</td>
</tr>
<tr>
<td>2.1.2 Partitioning</td>
<td>5</td>
</tr>
<tr>
<td>2.1.3 Frame Types</td>
<td>6</td>
</tr>
<tr>
<td>2.1.4 Intra Prediction</td>
<td>7</td>
</tr>
<tr>
<td>2.1.5 Inter Prediction</td>
<td>8</td>
</tr>
<tr>
<td>2.1.6 Transform</td>
<td>10</td>
</tr>
<tr>
<td>2.1.7 Quantization</td>
<td>11</td>
</tr>
<tr>
<td>2.1.8 Entropy Coding</td>
<td>12</td>
</tr>
<tr>
<td>2.1.9 Loop Filter</td>
<td>13</td>
</tr>
<tr>
<td>2.1.10 Decoder</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Thor video coding algorithm</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1 Partitioning</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2 Prediction</td>
<td>15</td>
</tr>
<tr>
<td>2.2.3 Transform and Quantization</td>
<td>16</td>
</tr>
<tr>
<td>2.2.4 Entropy Coding and In-Loop Filters</td>
<td>16</td>
</tr>
<tr>
<td>2.2.5 Comparison between Thor and current video coding technology</td>
<td>16</td>
</tr>
<tr>
<td><strong>3 Target Implementation Platform</strong></td>
<td>21</td>
</tr>
<tr>
<td>3.1 Platform Hardware</td>
<td>21</td>
</tr>
<tr>
<td>3.2 Design Tasks</td>
<td>23</td>
</tr>
<tr>
<td>3.3 Datapath Elements</td>
<td>25</td>
</tr>
</tbody>
</table>
### 4 Implementation

- 4.1 Thor Video Decoder Implementation ........................................... 29
- 4.2 Hardware Implementation ......................................................... 30
  - 4.2.1 Dequantize ................................................................. 30
  - 4.2.2 Inverse Transform ......................................................... 32
  - 4.2.3 Implementation of the Hardware-specific Code in the Algorithm .... 37

### 5 Results

- 6 Conclusions

Bibliography
List of Tables

4.1 Delays resultant of the design elements. ................................................. 32

5.1 Results of Thor’s video decoder implementation on Coreworks’ platform, without hardware acceleration. ......................................................... 42

5.2 Speedup results obtained for the inverse transform stage, for each hardware implementation performed. ................................................................. 42

5.3 Speedup results obtained for the decoder, for each hardware implementation performed in the inverse transform stage. .................................................. 43

5.4 Speedup results obtained for the hardware implementation performed in the dequantize stage. ................................................................. 43
# List of Figures

2.1 Codec evolution trade-off. ................................................. 3  
2.2 Typical encoder block diagram. ....................................... 4  
2.3 Original image. ........................................................... 5  
2.4 RGB components of the original image. .............................. 5  
2.5 YCbCr components of the original image. ............................ 5  
2.6 CB partitioning used by HEVC inter prediction modes. ............... 6  
2.7 Example of a sequence of frames. ...................................... 7  
2.8 Intra prediction modes of H.264/AVC standard ......................... 8  
2.9 Intra prediction modes of HEVC standard. ........................... 8  
2.10 Three-Step Algorithm example. ...................................... 9  
2.11 Transform process on a prototype Macroblock. ........................ 10  
2.12 Quantization process on a prototype block. .......................... 11  
2.13 ZigZag coefficient scanning method ................................. 11  
2.14 Coefficient scanning methods for HEVC. ........................... 12  
2.15 Typical decoder block diagram. ................................. 14  
2.16 Intra prediction modes of Thor standard. .......................... 15  
2.17 Video codec comparison with PSNR metric. .......................... 18  
2.18 Video codec comparison with PSNR-HVC metric. ..................... 18  
2.19 Video codec comparison with SSIM metric. ........................ 19  
2.20 Video codec comparison with FASTSSIM metric. .................... 19  
3.1 FireWorks architecture. ............................................... 22  
3.2 Sideworks architecture. ............................................... 22  
3.3 Main Design Tasks for the development of an embedded solution. 23  
3.4 Sideworks control element. ........................................... 25  
3.5 Time unit element. ..................................................... 26  
3.6 Basic arithmetic unit element. ....................................... 26  
3.7 Memory element. ....................................................... 27  
3.8 Delay element. ......................................................... 27  
3.9 Adder and Multiplier elements. ................................. 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Pseudo-code relative to the dequantize process implemented in the platform.</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>First part of the dequantize architecture (memory address generation).</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>Second part of the dequantize architecture (output computation).</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Thor's one dimensional inverse transform for an exemplary 32x32 TB</td>
<td>34</td>
</tr>
<tr>
<td>4.5</td>
<td>Pseudo-code relative to the inverse transform process implemented in the platform.</td>
<td>35</td>
</tr>
<tr>
<td>4.6</td>
<td>First part of the inverse transform architecture (memory address and input generation).</td>
<td>37</td>
</tr>
<tr>
<td>4.7</td>
<td>Second part of the inverse transform architecture (output computation).</td>
<td>38</td>
</tr>
</tbody>
</table>
# Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMVP</td>
<td>Advanced Motion Vector Prediction</td>
</tr>
<tr>
<td>AVC</td>
<td>Advanced Video Coding</td>
</tr>
<tr>
<td>BAU</td>
<td>Basic Arithmetic Unit</td>
</tr>
<tr>
<td>CABAC</td>
<td>Context-Adaptive Binary Arithmetic Coding</td>
</tr>
<tr>
<td>CAVLC</td>
<td>Context-Adaptive Variable-Length Coding</td>
</tr>
<tr>
<td>CB</td>
<td>Coding Block</td>
</tr>
<tr>
<td>CLPF</td>
<td>Constrained Low Pass Filter</td>
</tr>
<tr>
<td>CTB</td>
<td>Coding Tree Block</td>
</tr>
<tr>
<td>CTU</td>
<td>Coding Tree Unit</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete Cosine Transform</td>
</tr>
<tr>
<td>DLL</td>
<td>Down-Left-Left</td>
</tr>
<tr>
<td>DST</td>
<td>Discrete Sine Transform</td>
</tr>
<tr>
<td>FASTSSIM</td>
<td>Multi-scale Structural similarity</td>
</tr>
<tr>
<td>FU</td>
<td>Functional Unit</td>
</tr>
<tr>
<td>HEVC</td>
<td>High Efficiency Video Coding</td>
</tr>
<tr>
<td>JCT-VC</td>
<td>Joint Collaborative Team on Video Coding</td>
</tr>
<tr>
<td>LDSP</td>
<td>Large Diamond Search Pattern</td>
</tr>
<tr>
<td>MPEG</td>
<td>Moving Picture Experts Group</td>
</tr>
<tr>
<td>MPM</td>
<td>Most Probable Modes</td>
</tr>
<tr>
<td>MU</td>
<td>Memory Unit</td>
</tr>
<tr>
<td>NAL</td>
<td>Network Abstraction Layer</td>
</tr>
<tr>
<td>PB</td>
<td>Prediction Block</td>
</tr>
<tr>
<td>PSNR-HVS</td>
<td>Peak signal-to-noise ratio modified by Human Visual System properties</td>
</tr>
<tr>
<td>PSNR</td>
<td>Peak signal-to-noise ratio</td>
</tr>
<tr>
<td>SAO</td>
<td>Sample Adaptive Offset</td>
</tr>
<tr>
<td>SB</td>
<td>Super Block</td>
</tr>
<tr>
<td>SC</td>
<td>Sideworks Control</td>
</tr>
<tr>
<td>SDSP</td>
<td>Small Diamond Search Pattern</td>
</tr>
<tr>
<td>SSIM</td>
<td>Structural similarity</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>TB</td>
<td>Transform Block</td>
</tr>
<tr>
<td>ULL</td>
<td>Up-Left-Left</td>
</tr>
<tr>
<td>UL</td>
<td>Up-Left</td>
</tr>
<tr>
<td>UUL</td>
<td>Up-Up-Left</td>
</tr>
<tr>
<td>UUR</td>
<td>Up-Up-Right</td>
</tr>
<tr>
<td>VCEG</td>
<td>Video Coding Experts Group</td>
</tr>
<tr>
<td>VLC</td>
<td>Variable-Length Codeword</td>
</tr>
<tr>
<td>WPP</td>
<td>Wavefront Parallel Processing</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Digital video files require an excessive amount of bandwidth and storage space, even the ones with lower quality, making their transmission very difficult. For that reason, digital video is compressed before storage and transmission and decompressed before its final use. With that purpose, the continued development of video codecs with better performance has been of the utmost importance since the early digital video applications.

The most popular codecs have been mainly created by either MPEG or VCEG [1]. Recently, in 2013, a new codec was presented, brought by a collaboration of these two main companies, the High Efficiency Video Coding (HEVC) [2], establishing the new video compression standard. This video codec standard can double the data compression ratio compared to its predecessor, H.264/MPEG-4 Advanced Video Coding (AVC) [3], at the same level of video quality. Despite the promising features, issues involving the amount of patents present in the codec and how it affects licensing costs, have motivated several organizations to start working on a royalty free video codec. Thor video standard [4] is a project of Cisco with that purpose in mind. Although in its early stages, it already shows interesting features, and it is highly optimized for hardware, targeting embedded platforms.

Coreworks is a provider of Semiconductor Intellectual Property (SIP) for multi-standard multimedia equipments, such as smartphones, tablets, set-top boxes, HD/UHD TV, etc. Their field of work prioritizes the development of interfaces for some standard audio formats, and digital audio encoders and decoders for different audio coding standards1. Corework’s platform relies on an embedded processor, working side by side with a set of reconfigurable hardware accelerators. With the challenge of expanding the platform to video encoding/decoding solutions, this thesis has the purpose of implementing and accelerating a video decoder in the Coreworks’ platform.

The main objective of this work is to implement a video decoder on Corework’s platform, and successfully accelerate parts of the decoder, using hardware implementations. A careful study of the selected video codec needs to be performed, not only for the adaptation to the targeted platform, but also to identify which parts of the algorithm should be accelerated. Alongside this study, is a research of the platform, to understand all the tools at disposal, and the limits of what can be implemented.

1taken from http://www.coreworks-sa.com/
The structure of this thesis starts with an approach to video compression, describing the basic concepts of video codecs for each stage, while also showcasing the latest developments and how they contrast with the targeted video codec. The next chapter gives an outline on Coreworks’ platform components and its capacities, it is also presented the sequence of tasks needed to be executed to implement and accelerate an algorithm in an embedded platform such as Coreworks’. Special focus is given to the design elements used to define the hardware implementation.

In chapter 4 is described the work done to implement the selected decoder on the Coreworks’ platform with the implementation of hardware accelerators for some computation stages. The approach taken was to follow the sequence of tasks described in the previous chapter, while describing and explaining the work and the decisions made at every task. A thorough look is given to the blocks implemented in hardware. Lastly, in chapter 5 the results are presented and discussed, showing which of the objectives were met and by what measure. The last chapter concludes about the work developed and gives some directions for future work.
Chapter 2

Video Compression

2.1 Video Coding

Video compression provides the reduction of video file sizes with minimal loss in image quality, serving many purposes, such as storage or broadcasting. In accordance, a video codec is a combination of algorithms that allows the compression and decompression of a video data file, by using an encoder and a decoder, respectively. Profiles and levels can be defined for codecs. Profiles make restrictions on the used techniques, while levels make restrictions relative to the values of the encoding parameters.

Since the first codec has been introduced, about 30 years ago, there has been a constant improvement, which represented a significant increase of the compression rate, and of the quality of the decompressed video. However, there is trade-off [5] that must be respected, which includes not only the compression rate and the final video quality, but also the involved computational complexity, as it is shown in figure 2.1.

![Codec evolution trade-off.](image)

High Efficiency Video Coding (HEVC) [2] is the latest video coding standard, a joint project of the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG) standardization organizations, which have been working together in a partnership known as the Joint Collaborative Team on Video Coding (JCT-VC).

Its predecessor was H.264/MPEG-4 Advanced Video Coding (AVC) [3], with which HEVC shares
a lot of techniques and functionalities. Therefore, exemplifying with these two codecs, should be the most fitting way to showcase the current state of the depicted blocks, and the recent evolution they have been through. At the same level of video quality, HEVC can double the data compression ratio of its predecessor, a significant improvement, that results from the sum of various small enhancements on multiple blocks.

Compression techniques can be classified as lossy or lossless. What separates the two is the amount of data that can be lost (possibly resulting in worse quality of the decoded video). In a lossless process, all original data can be recovered when the video is uncompressed. In contrast, although certain information is permanently eliminated in a lossy procedure, such loss usually represents redundant data that has little or no influence in the quality of the video to the human eye. Generally, a codec is made of lossy and lossless processes, so there is a balance between the compression rate and the achievable video quality.

Figure 2.2 shows the block diagram of a typical encoder. In the following subsections, it will be presented a brief description of the main encoder modules, in order to facilitate the presentation of the proposed implementation.

Figure 2.2: Typical encoder block diagram.

2.1.1 Colour Compression

A video signal is fundamentally a combination of images (also referred as frames). Naturally, a greater number of frames per second originates greater video quality as well. Images are typically acquired using the RGB colour space (separated by components in figure 2.4, relative to the original image in figure 2.3). However, in most video compression schemes, the colour space must be changed to YCbCr (separated by components in figure 2.5). This scheme is composed by the luma component “Y” (responsible for brightness) and the blue and red chroma components, “Cb” and “Cr” (responsible for colour) respectively. This segmentation allows a reduction of the colour resolution, while maintaining a full resolution of the luma component. This compression technique takes advantage of perceptual features of human vision, whereupon the human eye has more difficulty detecting changes in colour, in
comparison with changes in brightness.

Figure 2.3: Original image.

Figure 2.4: RGB components of the original image.

Figure 2.5: YCbCr components of the original image.

An example is the 4:2:0 colour sub-sampling, which consists of a full resolution luma component, and two chroma components subsampled in both vertical and horizontal dimensions. This particular type of subsampling provides a compression of 50\% on the overall data size [1].

2.1.2 Partitioning

Along the compression procedure, frames are commonly divided into blocks of pixels. The size of these blocks of data and the number of times they are divided in, largely depends on the codec.

In HEVC, the upper level block partition is denoted as Coding Tree Unit (CTU) [6]. Each CTU consists of a luma Coding Tree Block (CTB) and two chroma CTBs, along with the associated syntax. The size of these CTBs is decided by the encoder, in conformity with its needs in terms of memory and computational requirements. Each CTB can have the dimensions of 16x16, 32x32 or 64x64. As a consequence of supporting larger block sizes than previous standards, increased compression is achieved, particularly in videos with high resolutions.
CTBs can encompass one or several Coding Blocks (CB), depending on the extent of the considered partitioning. Naturally, this splitting process is always accomplished by following the quadtree syntax elements, which define the minimum size allowed for CBs. There are two more partitioning processes at the Prediction and Transform levels, into Prediction Blocks (PB) and Transform Block (TB), respectively. Although the possible partitioning is standard in the Transform and Intra Prediction cases, in Inter Prediction there are special modes, illustrated in figure 2.6.

![CB partitioning used by HEVC inter prediction modes.](image)

At an upper partitioning level, HEVC and H.264/AVC both divide the encoded frames in data structures called slices. Slices may encompass regions or the entirety of a frame. The goal is to be able to independently decode different slices, that may contain information of the same frame. This provides an insurance in case of data loss.

Some innovations were introduced by HEVC, mainly to improve parallel processing capability. One is Wavefront Parallel Processing (WPP), a specification that splits a slice into rows of CTUs which are subsequently processed, row by row, with an intermission of two CTUs between rows. The other innovation is the definition of Tiles. Having the same purpose of WPP, this element consists of rectangular regions of a picture.

### 2.1.3 Frame Types

The most straightforward way to compress a sequence of frames is by eliminating or minimizing the redundancies, either spatial or temporal, and by only storing the differences (residuals) instead of every single piece of information. This concept clearly defines two possible approaches: minimizing the redundancies that can be found between different frames (inter-prediction), and those that are present in a single frame (intra-prediction).

Frames are usually divided in three different types, relative to the kind of encoding they are subject to.

- **An I-frame** is a frame that mostly exploits intra-prediction techniques and, as such, has no dependencies outside the frame. Its compression results solely from data present in the frame.

- **A P-frame** is an inter-predicted frame that is encoded using motion compensated prediction based on a reference frame (inter prediction), which can either be an I-frame or a P-frame. Additionally, it also uses the coding techniques that are used on an I-frame.
A bidirectional predictive frame, also known as B-frame, is not only exposed to the same compression techniques the P-frame is, but also to the motion compensated prediction from an additional reference picture. All things considered, it is the type of frame that requires less bits to encode, but also, larger computational complexity.

Naturally, when the video is restored in the decoder side (frame by frame), each frame that was previously encoded with one or more reference pictures, must be decoded with the same reference pictures. An example of a possible sequence of frames and its prediction dependencies, is illustrated by figure 2.7.

![Figure 2.7: Example of a sequence of frames.](image)

### 2.1.4 Intra Prediction

Regardless of the frame types, Intra Prediction may be used in the encoding of any frame. As its name suggests, it only uses the single, specific frame it is working with. The concept relies on predicting each block, with data from the neighbouring blocks, that have been formerly encoded. Hence, this mechanism exploits the spatial redundancy within frames, by only encoding the difference between each block and the predicted block (known as residuals). Since frames have a lot of redundant data, this is a valuable way to increase the compression rate.

There is a comprehensive set of modes for Intra predicting the blocks, which exploit the redundancies in different directions of the frame (see Figure 2.8 and 2.9). Among them, DC mode is a distinct mode that averages the whole neighbouring data, while planar mode is an approximate fit for the horizontal and vertical variation along the block edges.

Each individual codec supports a different set of modes. In particular, H.264/AVC supports eight intra prediction modes, illustrated in figure 2.8 [7]. HEVC [8] has a total of 33 directional modes, complemented with the DC and planar prediction, illustrated in figure 2.9.

As it was referred before, the most recent standards adopt a fine-grained partitioning of the frame in different sized blocks, so that smoother areas can be built by larger blocks, while detail-heavy areas are represented by smaller ones. This approach has a big influence in intra prediction, since it grants the opportunity to use less intra prediction modes in larger macroblocks (smoother areas), while using a greater number of modes for smaller blocks (detail-heavy areas).
To face the larger amount of modes, HEVC has a system in place to avoid testing every possible prediction mode. This system establishes the three Most Probable Modes (MPM) for luma, by using a sequence of rules and restrictions. In the event one of the MPMs is the current luma prediction mode, only the MPM index is sent to the decoder. Alternatively, a 5-bit code is sent. On the other hand, the chroma is given five choices: horizontal, vertical, planar, DC or the same mode as the corresponding luma.

2.1.5 Inter Prediction

Based on the intent to minimize temporal redundancies, inter-prediction uses data from other frames, in order to predict the blocks that compose the current frame. Generally speaking, a set of adjacent frames has little to no motion between them, facilitating the prediction mechanism. However, eventually existing motion can be estimated and accounted for, designated by Motion Estimation and Motion Compensation [9], respectively. Motion estimation may be categorized in different sorts of techniques:
gradient techniques, pel-recursive techniques, block matching techniques, and frequency-domain techniques. Special highlight is given to block matching techniques, since they are the ones mostly used in recent codecs.

Block matching takes advantage of the previously referred block partitioning. The concept relies on a comparison between a block of the frame to be predicted, and blocks from the reference frame, in pursuance of the best match possible. Subsequently, by examining the position of the best match, a motion vector is obtained, describing the displacement between it and the block to be predicted. Compression is accomplished by only transmitting the data concerning the motion vectors associated with the blocks of the reference picture, together with the prediction error signal, formed by the difference between the reference frame (already motion compensated) and the current frame. In simpler terms, besides the reference picture (in full), only the data needed to transform the reference picture into the frame to be predicted is stored in the decoder side.

There has been a constant development regarding the considered block matching algorithms, where the size of blocks and search areas have a strong impact on the performance of the motion estimation result. The simplest approach is the Full-Search Algorithm [10], a brute-force method that ensures the best outcome possible by matching every block with all possible candidates, only confined by the search window. Naturally, this originates a higher computational complexity, not practical for modern codecs.

In alternative, sub-optimal algorithms have been extensively used, in order to reduce the computational cost. The Three-Step Search Algorithm [11] selects one of nine candidate macroblocks in the first step, as the closest match possible. Then, it proceeds to approximate to a more optimized solution with another eight candidates in finer tuned spacing, which is repeated for the final step. The algorithm is illustrated in figure 2.10

![Figure 2.10: Three-Step Algorithm example.](image)

Another more recent sub-optimal search algorithm is the Diamond Search Algorithm [12], which considers two search patterns: a Large Diamond Search Pattern (LDSP), composed of nine checking points, of which, eight surround the center in a diamond shape; and a Small Diamond Search Pattern (SDSP) formed by five checking points. In sum, the LSDP pattern is repeated until the step in which the
minimum block distortion occurs at the center point. From this point on, the SDSP is used until the final search stage.

The algorithms presented in this section are just examples of the many alternatives developed through time. Furthermore, each method frequently has a variety of variations, in a constant effort for improvement.

For HEVC’s motion estimation, two modes can be executed: Advanced Motion Vector Prediction (AMVP) or Merge Mode. Merge mode organizes a motion vector competition scheme, formed by spatial neighbour candidates, temporal candidates and generated candidates. The chosen candidate’s motion vector is inherited by the current block, resulting in regions with the same prediction information.

Similarly to merge mode, AMVP creates a list of candidates. By using data from the reference picture and adjacent prediction blocks, priority is given to spatial motion candidates. Solely in the case there are not two spatial motion candidates available, a temporal motion vector prediction is included.

As to the motion compensation [13], 8-tap filters and 7-tap filters are used for luma fractional sample interpolation. The former are used for half-sample positions, while the latter for quarter-sample positions. For chroma samples, 4-tap filters are assigned. Explicit Weighted Prediction is enabled for P-frames and B-frames, which consists in scaling and offsetting the prediction.

2.1.6 Transform

Transform operators serve the purpose of facilitating the exploitation of the spatial correlation within a block, by reorganizing the information of each block by frequency order, making the resulting coefficients easier to encode than the original samples. Figure 2.11 shows the complete procedure.

Despite the existence of several transform techniques, this stage of video compression is undoubtedly associated with the Discrete Cosine Transform (DCT) [14]. Rightfully so, as it has been a constant presence in recent codecs, particularly HEVC, being the only exception, the 4x4 transform of luma intra prediction residuals, for which, and integer transform derived from Discrete Sine Transform (DST) is specified.

DCT is a lossless and reversible operation. Its input is the prediction error, previously determined by intra-prediction and inter-prediction, and the output are the transform coefficients.

Figure 2.11: Transform process on a prototype Macroblock. (a) Original block. (b) Transform kernel. (c) Transform output.
2.1.7 Quantization

Unlike the transform stage, Quantization [15] is a lossy process. This routine works around the inability of the human eye to perceive changes in high frequency brightness. These components tend to be ignored, while granting a greater emphasis to low frequency ones.

Fundamentally, the transformed blocks are usually divided by a customized quantization matrix, trimming the high frequency components and rounding to the nearest integer (see figure 2.12(b)). Subsequently, a Quantization Parameter (QP) determines the smallest representable value increment, transforming some values to zero and allowing the reduction of the number of possible values (see figure 2.12(c)). The process is illustrated in figure 2.12. Naturally, this process shortens the amount of data needed to encode the block.

Figure 2.12: Quantization process on a prototype block. (a) Quantization input block. (b) High frequency coefficient trimming. (c) Quantization output after the QP value modifications.

In the HEVC's case, the QP value is limited between 0 and 51 and scaling matrices are applied in order to perform the larger transformations (such as 32x32 and 16x16 sizes), decreasing the memory needed to store data relative to the transformations.

The coefficients of a block are generally scanned in a ZigZag fashion, as shown in figure 2.13. This method takes advantage of the coefficient reorganization processed in the Transform stage. In what concerns HEVC, three possibilities are defined: diagonal up-right, horizontal and vertical scans, portrayed in figure 2.14. The method is chosen depending on the size of the block and the type of prediction.

Figure 2.13: ZigZag coefficient scanning method
2.1.8 Entropy Coding

Entropy Coding is the final phase of an encoder, a lossless process to compress data by representing, in binary, high occurrence patterns with few bits and low occurrence patterns with many bits. Huffman Coding [16] and Arithmetic Coding [17] are two examples that served as groundwork for recent entropy codes.

Huffman coding assigns a variable-length code for each coefficient. The higher the coefficient’s probability of occurrence, the shorter the codeword. To determine the codeword for each coefficients, a Huffman Tree is projected with ramifications based on the coefficient’s probability of occurrence.

In contrast, Arithmetic Coding determines one codeword for the entire spectrum of coefficients on the input stream. The codeword is a number, representing a segmental subsection based on the coefficient’s probability of occurrence. Altogether, Arithmetic is closer to optimal than Huffman, achieving higher compression as well. On the other hand, it results in higher computational complexity, making Huffman simpler and quicker.

Modern codecs kept up with the evolution of entropy coding algorithms, particularly, the emergence of Context-Adaptive Variable-Length Coding (CAVLC) [18] and Context-Adaptive Binary Arithmetic Coding (CABAC) [19].

In fact, posterior to the transform and quantization processes, there is a high number of zeros in the blocks coefficients, even blocks solely constituted by zeros. Previous methods did not take this in account. Therefore, what gives an edge comparing with other methods, is the implementation of context conditional Variable-Length Codeword (VLC) tables. CAVLC adopts a dedicated zero-block codeword table for fast all-zero block encoding and the nonzero index table for direct nonzero coefficient encoding.

Possessing higher complexity, CABAC can improve the efficiency even further. Besides context conditional VLC tables, information from nearby elements is used to optimize the probability estimate. In conjunction with arithmetic coding and low-complexity multiplication-free operations, it justifies the typical bit rate reduction of 10−15(%) in comparison with CAVLC.

A further evolution was made from H.264/AVC, which had both CABAC and CAVLC, to HEVC that solely uses CABAC [20]. Furthermore, several improvements were made comparing with H.264/AVC’s CABAC. One of the improvements focus was Context Modelling, which can significantly improve CABAC’s efficiency. Instead of just providing further contexts, HEVC’s approach is giving more information with less contexts, namely, by taking advantage of the partitioning operated in previous steps. Advances
were also laid out to handle the increased size of transformation blocks.

2.1.9 Loop Filter

As referred before, frames are partitioned in blocks in order to facilitate and accelerate the processing that follows. A side effect of this sectioning are the introduction of discontinuities at the block boundaries, that can occur in the reconstructed signal, known as blocking artefacts. They are mainly related with small prediction errors, introduced in the course of the intra prediction and inter prediction procedures.

The solution to this problem is the adoption of a Deblocking Filter [21] that smooths rough edges formed between blocks. This filter is usually incorporated in both encoding and decoding processes.

In HEVC, it is applied to samples adjacent to a block boundary, that is aligned on a 8x8 sample grid. Exceptions are made when the boundary is also a picture boundary, or when deblocking is deactivated in the current slice or tile boundary.

Another filter used and introduced by HEVC is the Sample Adaptive Offset (SAO) filter, whose purpose is to make a better reproduction of the original signal. In particular, after the quantization is performed, several coefficients’ amplitude are difficulty recovered. The SAO filter adds an offset value, with the aid of look-up tables provided by the encoder.

2.1.10 Decoder

A Decoder [22] is fundamentally the opposite of an encoder, and its responsibility is to revert the encoder’s work with the highest accuracy possible, in favour of the decoded video quality. Many aspects have to be in accordance in order to decode the information contained in the input bitstream. Naturally, the decoder needs to be in line with the encoder’s data syntax and the way to interpret it. Other aspects range from the speed at which the bitstream can be fed to the decoder, to the amount of data that needs to be stored in buffers, etc. With all these constraints, it is clear that a decoder has a high correlation with the encoder, and as such, needs to be in conformity with the encoder’s standard, only constrained by profile and level implications. The typical block diagram of a decoder is presented in figure 2.15.

Apart from the bitstream containing the encoded information, the encoder also sends data from the prediction blocks, so that the process can be reverted with the highest precision. Commonly, motion and mode decision data are required for the prediction signals to be mimicked by the decoder. Special attention is given to the frame decoding order, since most frames have dependencies with other frames.

The only process inherently non-invertible is quantization. In this case, the decoder does a best-effort approximation with inverse quantization. All other processes are invertible, so, by definition, there should be no loss of data.

In the syntax matter, H.264/AVC’s main syntax architecture features have perdued in HEVC. The bitstream is organized in Network Abstraction Layer (NAL) units, containing the syntax structure. Usually, the first NAL unit is the header, carrying information about the type of data of the payload in question.
2.2 Thor video coding algorithm

The promising features of HEVC, when compared with its predecessor standard, H.264/AVC, make this codec the current best option for video compression. Despite the unequivocal quality demonstrated by HEVC, several problems have arisen. New HEVC licensing pools emerged, constituting already two licensing pools, with the possible appearance of a third. These events made HEVC licensing costs increase rapidly, which immediately urged several companies and organizations to start working on their own video codec. In fact, the Internet Engineering Task Force (IETF) is already devising some efforts to develop a next-generation royalty free video codec in its NetVC workgroup.

Thor [4] video standard is a product of the events surrounding HEVC and, although it is still in the beginning of its development, already shows great potential. This project, together with Mozilla’s Daala video codec, are part of the contributors to the IETF’s endeavours. Because it is particularly optimized for hardware, Thor is ideal for an implementation in embedded systems. Even though, the overall performance is not up to par with the current standard, some stages can match the performance achieved by HEVC, as discussed in the following subsections.
2.2.1 Partitioning

Similarly to HEVC, Thor enables the partitioning of blocks at several levels of the encoder. The biggest block size is 64x64 and is referenced as Super Block (SB). Blocks can then be divided into coding blocks by using a quadtree fashion, with the minimal size being 8x8.

In the prediction level, there are different possibilities for intra and inter prediction. While in intra, only one PB is allowed per coding block, in inter prediction it is possible to split in one, two or four PBs.

For the transform stage, each CB can be split into one or four TBs.

2.2.2 Prediction

The five angular modes illustrated in figure 2.16(a), added with DC, vertical and horizontal modes, compose the full spectrum of Thor’s intra prediction modes. The five angular modes are as follow: Up-Up-Right (UUR), Up-Up-Left (UUL), Up-Left (UL), Up-Left-Left (ULL) and Down-Left-Left (DLL). By attributing one pixel for each direction in the name of the angular modes, their characteristics are made clear. For instance, in the UL mode, the direction is one pixel up and one pixel left, as demonstrated in figure 2.16(b). Naturally, the 35 mode spectrum present in HEVC assures better intra-prediction performance.

![Figure 2.16: (a) Intra prediction modes of Thor standard.(b) UL Intra prediction mode demonstration.](image)

Regarding inter prediction, Thor represents luma motion vectors in quarter-pixel resolution, meaning that the precision of motion vectors is a quarter of the distance between luma samples. For chroma motion vectors, the precision is one eighth of the distance between chroma samples.

The motion estimation process in Thor is fundamentally similar to HEVC. There are four motion vector prediction modes: Inter0, Inter1, Inter2 and Bipred. Inter0 and Inter1 modes resemble HEVC’s merge mode, by forming a list of candidates with information from neighbouring blocks. The motion vector is then chosen according to the availability of blocks, and finally, the chosen motion vector’s index is inherited by the current block. Inter0 is solely projected for SBs, taking only one candidate with the value of zero for any other block size.
In contrast with the previous modes, Bipred and Inter2 do not inherit another block's motion vector, but predict it instead. The prediction is based on the selection of three candidates from the neighbouring blocks (dependent on the availability of the blocks). In order to finish the motion vector prediction, a median of the candidates is done. Since the motion vector is predicted and not inherited, explicit motion vector information is required. In Bipred mode, two reference frames are used for prediction, and two motion vectors are predicted as well.

Regarding motion compensation, 6-tap filters and 4-tap filters are used for luma and chroma fractional sample interpolation, respectively. Although for the latter, the specifications are the same in HEVC, in the luma case, HEVC uses higher order filters, naturally resulting in better results.

2.2.3 Transform and Quantization

Transform sizes range from 4x4 to 64x64, being the latter an unique feature of Thor. Apart from this, this stage is very similar to HEVC. The process is performed by an integer-approximation to DCT and an embedded structure is also contemplated, that allows smaller transform matrix elements to be extracted from bigger transforms.

The quantization process is rather undeveloped, not even possessing any QP control, which means there is a fixed value used for every case. The ZigZag method is the only possibility for coefficient scanning.

2.2.4 Entropy Coding and In-Loop Filters

Thor applies a CAVLC scheme, a downgrade comparing to HEVC's exclusive use of CABAC. Yet, it contains improvements and adaptations that increase its performance.

Two filters are specified: a Deblocking Filter and a Constrained Low Pass Filter (CLPF). The luma deblocking is performed on a 8x8 grid on block edges, while chroma deblocking is performed on a 4x4 grid.

The CLPF is applied after the deblocking filter, and has a similar behaviour to HEVC's SAO, working towards additional refinement of the reconstructed signal. Unlike SAO, CLPF utilizes fixed coefficients.

2.2.5 Comparison between Thor and current video coding technology

Understanding the main concepts used by current video codecs and how each codec contrasts from one another, this subsection provides a rough idea of their performance comparison.

A web application\(^1\) that provides a direct comparison of different video codecs is used to conduct several tests. This application is a continuous integration system, with different customization possibilities.

First and foremost, a range of test video sets are available, allowing still images and all kinds of video sets. For each video set, there is the possibility to select codecs out of the reference codecs, or

\(^1\) Available at arewecompressedyet.com
specific experimental runs. These are not interchangeable, being some options only available for some test video sets.

Secondly, the picture quality metric may be adjusted to fit the characteristics focused by a specific test. Finally, there is the option to select a video resolution. In what concerns the presentations of the results, the bits per pixel metric is chosen for the x-axis. The y-axis is always used to represent the picture quality, measured in deciBel (dB).

The considered picture quality metrics are four:

- Peak signal-to-noise ratio (PSNR): The PSNR is basically the ratio between the maximum power of a signal and the corrupting noise, which, in video coding, corresponds to the ratio between the original data and the error induced by compression. This metric correlates poorly with visual quality.

- Structural similarity (SSIM): The SSIM is an improvement comparing with PSNR, and should give a more reliable codec comparison. It evaluates performance by comparing image quality with an uncompressed reference picture.

- PSNR modified by Human Visual System properties (PSNR-HVS): A more sensitive alternative to contrast changes and noise, providing better correlation with video visual quality than PSNR.

- Fast Multi-scale SSIM (FASTSSIM): A more advanced form of SSIM, possessing more sensitivity to details and textures.

The undergone tests focused on a comparison between Thor and several codecs. The used codecs were: x265 (software that uses the HEVC standard), x264 (associated with the H.264/AVC format), VP9 (video coding format developed by Google) and Daala (represented by master in the tests caption). Furthermore, the selected test video set was “ntt-short-1”, a set of 1-second video clips with varying characteristics. With a cleaner representation of the graphs in mind, the logarithmic scale is also selected.

By observing figures 2.17, 2.18, 2.19 and 2.20, the performance of a codec, for a given amount of bits per pixel, is measured by the obtained picture quality. As an example, in figure 2.17 it is noticeable that at the mark of 0.5 bits per pixel, Thor, x265 and VP9 can reach a value of, proximately, 43dB, whereas x264 and Daala are close to 42dB. By following this logic, it is observable through these figures with different picture quality metrics that x265 and VP9 achieve better results.

Although Thor can not reach the level of these two, it clearly outperforms Daala and x264 for the PSNR case. However, in both PSNR-HVC and FASTSSIM cases, Daala and x264 surpass Thor’s performance. One of the reasons behind this behaviour is the mediocre Quantization block used by Thor, completely exposed by metrics that further explore detail and contrast changes of videos.

There is a definite gap in performance between state-of-the-art video codecs, namely the x265 (the representative of the current video compression standard), and Thor. However, this gap is not substantial enough to undermine the benefits that Thor provides to the purpose of this thesis. Thor’s low complexity and being optimized for hardware implementations, facilitate the study and subsequent adaptation of
the codec. Furthermore, although some stages of Thor are a downgrade comparing with HEVC, there are cases where the performances of the two video codecs is similar. Being the acceleration of a video codec, the main goal of this thesis, by making one of these stages the target, it is possible to accelerate a state-of-the-art stage, while using a more suited codec for hardware implementations. All things considered, Thor is ideal for an implementation in embedded systems and, consequently, to the
Figure 2.19: Video codec comparison with SSIM metric.

Figure 2.20: Video codec comparison with FASTSSIM metric.

purpose of this thesis.
Chapter 3

Target Implementation Platform

The target hardware platform to implement Thor’s video decoder is a solution developed by Coreworks. This platform is composed by one or more embedded processors called FireWorks combined with one or more configurable hardware accelerators, called Sideworks. This platform was specially designed for audio solutions, and more recently, is being adapted to video decoding as well. The distinctive characteristic of Coreworks’ product is the ability to create a reconfigurable architecture with high efficiency in silicon area and energy consumption.

Regarding the interface with the user, FaceWorks is the contemplated software, a network interface that provides standardized functions to manage the use of the platform.

In order to implement a hardware application in the platform, a set of basic tools are available for Coreworks, that allow the generation and configuration of the Sideworks accelerator, based on a given datapath. This datapath is built by a design tool, using a set of existing elements on the Coreworks library, or developed by the designer itself.

3.1 Platform Hardware

Coreworks’ platform embedded processor is FireWorks, a 5-stage pipeline architecture processor, as illustrated in figure 3.1. This processor uses a 32-bit data, modified Harvard RISC architecture. The difference with a Harvard architecture is the option for instruction memory and data to be accessed with the same address space. FireWorks’ tool kit features a GNU compiler and tools for simulation and implementation.

Sideworks instances are used together with FireWorks, both fetching data and code from a common memory. These instances are reconfigurable accelerators generated by the internal SideGen software design tool. The general composition of a Sideworks instance is illustrated in figure 3.2.

The flow of information is controlled by the Configuration Register File, containing the specifications pre-defined for the datapath implemented. This information will select the options from several multiplexers, that contain every possibility for the specific bitfile, generating the intended path of operations, inputs and outputs.
Relative to the flow of a datapath itself, it commonly starts with the generation of the addresses to read or write from/to a Memory Unit (MU). The data read from these MUs, along with inputs previously instanced, feed the inputs of Functional Units (FU). After processing, the outputs of the FUs can be fed into other FUs or be written into memories, depending on the operations specified for the datapath. After all operations are completed the outputs are returned to the main program being run in Fireworks.
3.2 Design Tasks

In order to implement and accelerate an algorithm in Corework’s platform, a series of tasks have to be performed. The main design tasks are represented in figure 3.3, which naturally begins with choosing the algorithm to be implemented in the platform.

![Design Tasks Diagram](image)

Figure 3.3: Main Design Tasks for the development of an embedded solution.

The second step is implementing the algorithm (containing no acceleration yet), so it is able to run in the platform at full capacity. For this task, an adaptation is needed, since the algorithm is generally not compatible with embedded systems, or the targeted platform in particular.

Naturally, the complexity of this process depends on how optimized to hardware, the algorithm is. If the algorithm is built for desktop computing environments, this adaptation is commonly referred as Porting and specific details have to be taken into account.

When comparing desktop operating systems with embedded systems, the constraints to resources is a key-factor, specifically in terms of time, memory and power. Desktop systems are usually assumed to have no limits whatsoever. This difference explains the reason for an embedded system’s efficient use of resources.

Dynamic memory allocation is a particular example of a procedure that must be carefully planned and managed in order to be implemented in a embedded system. Systems such as the Corework’s
platform are expected to be highly dependent and have a predictable, real-time behaviour, while reliably running for long periods of time. The combination of being commonly seen as unpredictable and the system’s memory constraints, make dynamic memory management a frequently avoided procedure in embedded systems.

Another important issue is the availability of some library functions and the difference in the call interface. Different systems tend to use distinct functions for the same functionality or even have functionalities that other systems do not have. In particular, desktop operating systems use some libraries that are exclusive to the system. For example, libraries usually used in desktop systems relative to file handling and list development cannot be used on the Coreworks’ platform. Libraries containing specific data types may also not be available. These issues need to be resolved without impacting the algorithm’s results.

Lastly, it’s crucial to understand that debugging is processed differently in desktop systems and embedded systems. Generally, external tools are needed on embedded systems, and the procedure is not so straightforward. A common way to debug the code in embedded systems is through the use of strategically placed \texttt{print} functions, relaying information that can be used to track errors.

After the adaptation, an analysis of the algorithm is done, in order to find out which parts of the code should be accelerated. The strategy is to accelerate the largest bottlenecks to the algorithm in terms of time (so the acceleration has the biggest possible impact). An analysis to obtain these bottlenecks can be performed through the use of profilers. Besides the profiler analysis, a study is done to evaluate how difficult it is to accelerate that portion of the algorithm. A balance has to be done between the two conditions, in order to perform a significant acceleration using a datapath with reasonable complexity.

The following step is the design of the datapaths containing the procedure to be accelerated. In this step, the hardware must be an accurate representation of the code, therefore, a full understanding of the datapath elements available is crucial. Usually, the design is made using elements already developed by Coreworks, unless a new block has to be devised for a specific task.

A detailed simulation ensures that the previous task was correctly performed. In the simulation, the flow of data through all different operations is observed with a set of example values for the inputs. By certifying that the operations have the correct values at the entrance and exit, the datapath can be validated. Usual problems tend to relate with a de-synchronization between different operations. This issue can be solved with the implementation of delay values on the intended places.

Finally, the hardware-specific code has to be included in the algorithm and tested, to guarantee the program runs smoothly.

The implementation of the datapath in the algorithm, by using functions developed by Coreworks, starts with the selection of the datapath. Then, the values that will be used in the registers are stored, along with the data that will be stored in the memories to be read. Subsequently, the datapath is started and the algorithm is paused until the Sideworks process has ended. Finally, the output values, present in a memory, are stored in a variable destined to that purpose. With everything implemented, tests are conducted in order to ascertain the acceleration achieved.
3.3 Datapath Elements

The design of datapaths serves the purpose of accelerating pieces of code or even entire blocks in the algorithm, through the implementation of an accurate representation in hardware. The datapaths can be constructed using the elements already made available by the SideFU library. There’s a wide range of elements with different objectives and various possible customizations. Although some elements perform well-known operations not requiring any particular explanation, a select number of mandatory elements for the creation of a datapath, are illustrated and described:

- The Sideworks Control (SC), represented in 3.4 (represented as SIW_CTRL), is the element that controls the beginning and end of the whole datapath process. The done input receives the signal that relays the end of operations, and the enable (en) output relays the beginning of all operations. The enable signal is usually connected to the Timing Units. A delay (dly) is usually imposed, so there is time to prepare all necessary steps prior to the start of the datapath procedure.

- The Timing Unit, observable in figure 3.5, is the mechanism that controls the pacing at which operations are made, and the number of times they are made. There are four possible loops and each can have their own number of iterations, defined by the input limit. This allows, for example, operation A to be performed five times while operation B is performed seven times, which can be very useful, specially when the code being accelerated contains “for” cycles. It is important to notice that the limits value corresponds to the number of iterations desired minus one. This is because the block also considers zero as an iteration, so for a limit of zero, the block would execute one iteration.

  Each loop has a start, end, and enable outputs. While end only gets activated when the loop ends, start does so at the beginning of the first iteration. Enable activates at the beginning of every iteration.

  Similar to the previous element, there is a done, enable (en), and delay (dly) controls, however, in this case enable is an input (usually fed by SC enable) and done is an output (usually connected with done of the SC element).

- The element present in figure 3.6 (represented as BAU_12) is a Basic Arithmetic Unit (BAU), crucial to the address generation for the memories. Fundamentally, this block is characterized by a load value (ld_val) and an increment value (inc). The load value is loaded to the output (out), when
the load input (ld) is activated. The increment value is added to the value already present on the output, when the enable input (en) is triggered. The enable and load inputs are usually connected to the enable and start outputs of the timing unit (of the loop being used), respectively.

Although BAU units are commonly used for memory addressing, they can also be used to generate inputs for functional units, when the inputs are required to change in accordance with iterations. Yet, if the same BAU is used for both actions, timing complications can arise.

- Despite being a common sight in hardware, the memory blocks require some explanation about subtleties of the target platform. As illustrated in figure 3.7 (represented by MEM), this memory has two ports, A and B, but only one can be used at a time. Both ports can be used for writing or reading. For this purpose, as typically seen, there’s an input to receive the address (address_a and address_b for A and B port, respectively), an input for data to be written (data_in_a and data_in_b for A and B port, respectively), a singular bit input that activates writing (write_a and write_b for A and B port, respectively), and an output for reading data (data_a and data_b for A and B port,
Five memories are used (memory 0 to memory 4). Aside from memory 2, every memory has a capacity of 4096 bytes. Memory 2 has a capacity of 1048 bytes.

- In some cases, operations can have inputs with different timings, which alters the result of the operation but can also have repercussions on the following operations. This issue can be solved by inserting delays to the elements. Another alternative is to instance delay elements (see figure 3.8 in the inputs that need to be delayed. This element already contains a starting delay of one, so any other delay applied to this element will accumulate with that value.

Other elements are typically functional units that perform operations such as addition and multiplication, as illustrated in figure 3.9.
Figure 3.9: Adder and Multiplier elements.
Chapter 4

Implementation

The implementation was preceded by, not only the study of the different codecs options, in order to gather enough information to make the best choice, but also the analysis of the characteristics and functionalities of the targeted platform. With this preparation complete, the implementation was started.

4.1 Thor Video Decoder Implementation

Following the set of tasks illustrated in the previous chapter, the choice of the algorithm was the first step to the implementation. In that regard, an analysis and some experimentations were made on the video codec algorithms available. The search was focused on finding a video codec that required minimal effort for the adaptation to the targeted platform, combined with a good performance. As discussed in the corresponding chapter, Thor video codec was the best choice and, as such, the target algorithm for this thesis.

By being coded in C, Thor already suggests an easier adaptation to embedded systems, since C is highly portable between different platforms. Nevertheless, the adaptation was mainly devoted to the match of libraries used in the algorithm, with the libraries used in the targeted platform.

When the functions were not available, they had to be implemented. A case where this situation repeatedly occurred, was in all functions related with files. These kind of functions are not supported by the Corework's platform, so they were reconstructed by replacing files with vectors. In specific, the bitstream originated from the encoder (containing the information to be decoded), was included in the algorithm in the form of a hexadecimal vector, so it could be processed by the functions developed. Naturally, the decoder's output file was also instanced as a vector.

As usual for video codecs, the decoder expects a command line to assign several different options, namely the name of the input and output file. In the platform, these settings can no longer be handled issuing a command line. To tackle this issue, two different solutions were available:

- One answer was using the interface developed by Coreworks. With some adaptation to the algorithm code, the interface allows the executable file containing Thor video decoder, to be sent and generated on the platform. The interface can then send the input files to the platform, which is
ready do decode. The downside is that the interface is only prepared for audio solutions, therefore, an adaptation would have to be done.

- The other solution was to adapt the algorithm by hard-coding the desired settings and disable the command line option. This means only one executable file is sent, that contains every information, even the input file. The downside is that, whenever the input file or the decoding settings are changed, a new executable file containing all the information has to be created and sent to the platform.

Although the first option was tested, the second option was the one implemented. Mainly, due to the excessive complexity of the adaptation of Coreworks’ interface.

In addition to all these changes, the makefile of the algorithm had to be completely rebuilt in order to include all the libraries used by Corework’s platform, and the correct way to compile them. With these adaptations properly implemented, an executable file is generated, that allows the algorithm to run in the platform.

4.2 Hardware Implementation

In the interest of selecting the candidate procedures to be accelerated among the blocks that constitute Thor video decoder, the main aspect evaluated was the impact that a hardware implementation would have, while using a datapath with reasonable complexity. To this effect, two stages were chosen: Dequantize and Inverse Transform. These stages of Thor video decoder use a series of simple operations, which are easier to implement in hardware and confer a higher probability of achieving acceleration. The Dequantize stage was primarily chosen because it appeared to require a simpler implementation, which was ideal as a first implementation to fully understand the platform’s hardware implementation tools. Furthermore, there is the potentiality of taking advantage of the high correlation between the Dequantize stage and Inverse Transform, since the matrix of coefficients that results from the Dequantize stage is immediately used as input to the Inverse Transform stage. Although the Dequantize stage is not at the level of the same stage of the current video codec standard, the Inverse Transform has a very similar process with its counterpart in HEVC, constituting a more attractive target for acceleration.

4.2.1 Dequantize

The Dequantize stage of Thor video coding algorithm has a low complexity, which enables the block to be entirely translated to hardware, since the resulting datapath has a reasonable complexity. The implemented operations of the dequantize function are described in the pseudo-code presented in figure 4.1, with the following parameters:

- size: Size of the current transform block.
- qp: Variable that represents the quantization parameter specified.
void dequantize (int16_t* coeff, int16_t* rcoeff, int qp, int size)

for i = 0 to i = size do
  for j = 0 to j = size do
    res1 ← coeff[i × size + j]
    res2 ← res1 + index
    res3 ← res2 × scale
    res4 ← res3 ≪ lshift
    res5 ← res4 + add
    res6 ← res5 ≫ rshift
    rcoeff[i × size + j] ← res6
  end for
  j ← j + 1
end for

Figure 4.1: Pseudo-code relative to the dequantize process implemented in the platform.

- coeff: As its name suggests, this matrix contains the information relative to the coefficients.

- rcoeff: Matrix containing the final processing values which in turn will be used in the Inverse Transform stage.

The remaining variables are computations performed before the hardware implementation, involving the quantization parameter and the size of the block.

The designed datapath is separated into figures 4.2 and 4.3, illustrated as implemented in the design tool. The first figure contains the address generation operations for the values read and written from/to memories. To this effect, two timing units tu_rd_0 and tu_wr_0 are instanced, that are responsible for generating the addresses of the memories read and written, respectively, by controlling two BAU units each. Since, there are two for loops, two timing unit loops are used, both with the same limit of size2 for the two timing units.

The timing unit tu_rd_0 controls the BAU units bau_rd_0 and bau_rd_1. The value size is applied to the increment value of BAU bau_rd_0 and zero to the load value. The output of BAU bau_rd_0 feeds the load value of BAU bau_rd_1 and the increment value is established as one. Finally, the output of BAU bau_rd_1 feeds the address of memory m0.

BAUs bau_wr_0 and bau_wr_1 are controlled by the timing unit tu_wr_0 and are projected in an identical way to the reading address generation. Using the same load and increment values, except that the load value of BAU bau_wr_1 is fed by the output of BAU bau_wr_0. The output of BAU bau_wr_1 provides the address of memory m1. The address generation design used for both cases allows a multiplication and an addition to be solved with the use of BAUs, which decreases the processing time of the datapath.

A standard implementation is used for the SC, connecting the done input to the end output of the timing unit that controls the addresses for writing in the memory (tu_wr_0), since this is last operation performed in the datapaths.

In the second figure, all the operations that produce the final result are implemented. Observing the pseudo-code, it is clear that the datapath is a straightforward representation of it. Using the address generated previously, the data from memory m0 is accessed and subsequently used on an addition with
the index value. The result is multiplied with the value scale. Then, the output of the multiplication is shifted to the left by lshift amount. The next output is added to add and the result is shifted to the right by rshift amount. Finally, the result of all these operations is stored in memory m1 in the address fed by the output of bau,wr. The storage of values is made at the end of every inner loop iteration.

Naturally, the design elements used to create the datapaths take a certain amount of time to finish their operation, as shown in table 4.1. To ensure all operations are performed at the correct timing, while using the correct inputs, the design has to be implemented with delays. The delay is either applied directly to the design element, or a delay element is instanced to the desired signal.

<table>
<thead>
<tr>
<th>Design Elements</th>
<th>Data delay (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>4</td>
</tr>
<tr>
<td>Memory read</td>
<td>2</td>
</tr>
<tr>
<td>Memory write</td>
<td>3</td>
</tr>
<tr>
<td>Adder/Accumulator/BAU</td>
<td>1</td>
</tr>
<tr>
<td>Shift(Left and Right)</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4.1: Delays resultant of the design elements.

Using table 4.1, the necessary delays for this design can be determined by adding the delays of all blocks that precede each operation. For the SC case, a delay of 1 clock is assigned as a standard measure to guarantee all the operations are ready to start processing. BAU bau,rd,1 uses control signals from the same timing unit as BAU bau,rd,0, however the output of BAU bau,rd,0 feeds BAU bau,rd,1’s load value, so it has to wait for the end of BAU bau,rd,0 processing, therefore, a delay of 1 clock cycle is applied to BAU bau,rd,1. The same reasoning is applied to BAU bau,wr,1. Concerning the writing of data in memory m,0, the data is delayed by all the operations performed to achieve the final result, which has to be applied to the address and the write enable signal. The operations performed up to the moment the data is stored are two BAUs, one memory read, two additions, one multiplication and two shifts, resulting in a delay value of 16 clock cycles. This delay would be directly applied in timing unit tu,wr,0, however, the address of memory m,0 also suffers a delay of 2 clock cycles originated from two BAUs. So, instead of applying a 16 clock cycles delay to the timing unit tu,wr,0, 14 clock cycles are applied. This way the address of memory m,0 arrives with the correct timing. The only signal left that has an incorrect timing is the writing enable of memory m,0. The signal already has a delay of 14 clock cycles, since it is an output of timing unit tu,wr,0, so another delay of 2 clock cycles has to be applied to the signal.

4.2.2 Inverse Transform

In Thor video decoder, the two dimensional inverse transform [23] is decomposed into two one dimensional transforms [24], with a data transposition between them. One of the transforms is processed on the rows, followed by another one processed on the columns of the TBs. As mentioned before, one of the features of Thor’s Transform stage is the ability to extract smaller matrixes from larger matrixes (embedded structure). This feature allows one dimensional transforms to be further decomposed into
odd and even parts, maximizing hardware sharing. The target of this stage’s first implementation is the matrix-vector multiplication relative to the odd parts, the largest computation in each one dimensional transform. The remaining parts are further decomposed into smaller matrix-vector multiplications, as shown in figure 4.4.

**Odd Parts Inverse Transform Implementation**

Before designing the corresponding datapath, the code was unrolled, introducing a new for cycle to the original algorithm code. This modification makes the resources be reused whenever it is possible, leading to an efficient use of the datapaths’s resources without affecting the datapath latency.

The pseudo code represented in figure 4.5 describes the operations performed in this datapath. The parameters used for this function are as follow:

- **size**: Size of the current transform block.
- **j**: Value representative of which 1-D transform is being performed.
- **tr_matrix**: Transform matrix containing the values used to process the block of coefficients.
- **coeff**: Matrix that contains the information relative to the coefficients, after being processed by the Dequantize block.
**Figure 4.3:** Second part of the dequantize architecture (output computation).

**Figure 4.4:** Thor’s one dimensional inverse transform for an exemplary 32x32 TB

- **Out**: Matrix that contains the final values after the processing of the datapath is complete.

Similarly to the Dequantize stage, the implementation design is divided into figures 4.6 and 4.7. The
```c
void inverse_transform_1d_odd(int16_t* coeff, int16_t* tr_matrix, int size, int j, int* Out)
{
    for (i = 0 to i = length do
        for x = xini to x = xcycles do
            res1 ← x × size
            res2 ← res1 + i
            res3 ← res1 + j
            res4 ← tr_matrix[res2]
            res5 ← coeff[res2]
            res6 ← res4 × res5
            res7 ← res7 + res6
            x ← x + 2
        end for
        Out[i] ← res7
        i ← i + 1
    end for
}
```

Figure 4.5: Pseudo-code relative to the inverse transform process implemented in the platform.

First figure contains the address generation to read and write from/to memories, in addition with the computation of two inputs. Specifically, the values i and x, that have a direct connection with the loops.

Two timing units tu_rd_0 and tu_wr_0 are once again used to control the BAUs that generate the addresses and inputs. Both timing units have two loops (justified by two for cycles) and share the same limits. The xcyles value is used for the inner loop (loop K), while length is used as the limit for the outer loop (loop J).

The timing unit tu_rd_0 controls two BAUs, bau_rd_0, that is used for the outer loop, and bau_rd_1, used for the inner loop, containing load and increment values correspondent to the ones presented in the pseudo-code (load value of 0 and increment value of 1 for bau_rd_0, and a load value of xini and increment value of 2 for bau_rd_1). BAU bau_rd_0 is the one that determines i and BAU bau_rd_1 outputs x.

Timing unit tu_wr_0 controls only one BAU, bau_wr_0 (same increment and load values as bau_rd_0), that generates the address of memory m2. A second BAU is unnecessary since the final value only has to be written in memory m2 in each outer loop iteration. The SC has an identical implementation to the Dequantize stage datapath.

Turning attention to the second figure, the first operation is a multiplication of x (fed by the output of bau_rd_1) and size. The output of the multiplication is ramified into two different additions. One adds i (fed by the output of bau_rd_0), while the other adds j. The outputs of these additions will enter as addresses to read data from the memories m0 and m1. Subsequently, the data read from the memories is multiplied and introduced on an accumulator. The accumulator is enabled for each inner loop iteration and writes its output on m2 at the end of each outer loop iteration. As referenced before, the output of the bau_wr_0 defines the address being written.

Once again, the delays have to be applied with reference to table 4.1. To the SC a delay a 1 clock cycle is applied, for the same reason expressed previously. The first element that requires a delay is the adder add_0, since the inputs arrive with different timings. The data coming from the multiplier mult_0 has a delay of 5 clock cycles resultant of one multiplication and one BAU. On the other hand, the other
input i, fed by BAU bau\_rd\_0 only has a delay of 1 clock cycle (originated from the BAU). As such, a delay of 4 clock cycles has to be applied to the output of BAU bau\_rd\_0.

The next element to be focused is the accumulator acc\_0. In order to ensure that the accumulator only loads data at the moment it receives the first data from the multiplier mult\_1, the load signal has to be activated at the same time the data arrives, with the start signal of an outer loop. Since the accumulator is preceded by one BAU, two multiplications, one addition and one memory read, the resulting delay is 12 clock cycles. Thus, a delay of 12 clock cycles is applied to the signal that controls the accumulator's load signal. Regarding the enable input, the same delay is applied. This way, everytime there is an inner loop iteration, the value at the input is accumulated with the value already present at the accumulator.

Finally, the timings of the inputs of memory m2 are corrected. The operations that precede the storage are composed by one BAU, two multiplications, one addition, one memory read and one accumulator, resulting in 13 clock cycles. In order to time the address and the write enable signal with the data that enters memory m2, a delay of 12 clock cycles is applied to the timing unit tu\_wr\_0. With this implementation, the address that already has a delay of 1 clock cycle (because it is generated by BAU bau\_wr\_0) arrives with the correct timing. The writing enable is a direct output of timing unit tu\_wr\_0, which means it still requires a delay of 1 clock cycle. However, by using the signal of the end of each inner loop iteration instead of the start signal, the signal is automatically delayed by one clock cycle (equivalent of being operated by a BAU).

Odd and Even parts Inverse Transform Implementation

The values xini and xcycles stipulate the beginning and ending step of the inner loop. By changing these values, the even matrix-vector multiplications can also be implemented with the same datapath, since the operations are the same for all sizes. By using the same datapath but defining different values for xini and xcycles, all matrix-vector multiplications of each one dimensional inverse transform are implemented in hardware. Generally this would suggest a bigger acceleration, however, since even matrices are smaller, the computations achieved may not be enough to surpass the latency needed to initialize each datapath.

Kernel Inverse Transform Implementation

A further development was the implementation of a kernel function. The kernel function makes the processing of several datapaths in a row more efficient, by initializing the datapath while the previous one is still running, eliminating the latency needed to initialize datapaths (except for the first datapath). Thus, using the datapath implemented to the odd matrix-multiplication, a kernel function was implemented that contains every matrix-multiplication in each one dimension inverse transform. Even though the datapaths are fundamentally the same, two adaptations had to be performed. Firstly, the registers with xini and xcycles were replaced by constants, relative to each case. Secondly, the starting address where the final result is stored, was implemented with an offset between datapaths. The reason is related with memories being shared by the datapaths, therefore, without this adaptation, there would
be successive overwrites of the results. This adaptation is achieved by changing the load value of BAU bau_wr_0 throughout the different datapaths.

4.2.3 Implementation of the Hardware-specific Code in the Algorithm

Posterior to the design of the datapaths, a simulation was performed to guarantee the datapaths had the correct behaviour throughout all operations. This simulation was performed to the datapaths alone, using an extensive range of inputs, so there is a high certainty that the datapaths are properly designed. Following the simulation, the datapaths were finally included in the decoder’s algorithm. This is fulfilled with a set of functions developed by Coreworks:

- **siw_lock()**: Function that initializes a Sideworks instance.
- **siw_set_state**: Function that stores the value on the specified register. Thedatapath, the register and the value to be applied, are all specified as parameters.
- **siw_ld_vec**: Works in a similar way to the previous function, but to store data in a memory instead. The parameters needed are the datapath, the memory and data to be stored.
Figure 4.7: Second part of the inverse transform architecture (output computation).

- **siw_ld_conf**: Function that loads the datapath to be used. The parameters are the datapath and an index. Each datapath is loaded with a different index, that will be used for further functions.

- **siw_sel_conf**: Function that selects the datapath to operate using the index correspondent to the datapath as parameter.

- **siw_run**: Function that initializes whatever datapath is selected.

- **siw_wait**: Function that forces all processing in the main code to wait, until the datapath process has finished.

- **siw_st_vec**: Function that extracts the data from a memory into a variable. The parameters are the datapath, the memory, the variable, and the size of the extracted data.

- **siw_unlock**: Function that ends the Sideworks instance.
Although in a kernel implementation, all the functions related with storing and extracting information are the same, new functions were introduced to launch the kernel of datapaths:

- `siw ld kernel first conf`: Function that loads the first datapath to be processed. The parameters are the datapath and the correspondent index.

- `siw kernel run`: Function that initializes the kernel, starting with the first datapath. The parameters are the index relative to the first datapath, the number of times the kernel is run, and the number of datapaths.

- `siw kernel wait`: Function that forces all processing in the main code to wait, until the kernel process has finished.

The final tests were done by sending the executable file to the platform, and assessing the behaviour of the decoder. Through the use of a profiler developed by Coreworks, that retrieves the cycles spent on each stage, assessments were also made on the acceleration achieved for each implementation. With this information, optimizations were implemented, relative to the placement of the functions previously described. The idea is to anticipate any function (related with sideworks) that can be done sooner, preventing the unnecessary repetition of certain functions. This optimization is mainly operated on the functions used to store values in registers or memories. For instance, the value `size` does not change for each of the one dimensional inverse transforms, so, rather than storing the value in a register right before the datapath is started, the value can be stored at the beginning of the inverse transform stage. Consequently, the value is stored one time in the entire stage, decreasing the stage's latency.
Chapter 5

Results

In this chapter we present the evaluation results for the implementation on the Corework's platform of the developed accelerator for the Thor video decoder. Three different hardware implementations of accelerators were performed for the Inverse Transform stage and one hardware implementation to accelerate the Dequantize stage. The results relative to the algorithm's implementation in the platform without any acceleration are presented, followed by the results of each of the hardware accelerated implementations and how they impact on the decoder's performance.

To this effect, 7 different video samples with resolutions of either 176x144 (qcif), 352x288 (cif) or 854x480 (FWVGA) were used. First, the video samples have to be encoded using Thor's video encoder, generating a bitstream that contains the data ready to be decoded. As previously mentioned, the bitstream is included as a hexadecimal vector, in the decoder code so it can be processed in the embedded platform. In order to encode the video samples through Thor's video encoder, it is mandatory the video samples are in YUV format. When that was not the case, the video samples were converted to the correct format. A total of 20 frames were encoded for each video sample.

Corework’s embedded processor was specified with a 120 MHz frequency, and a total of 3356 LUTS are used to instance the necessary resources. It was used an Arria V GT FPGA development kit, which has an Arria V GT FPGA.

The first test uses frame-rate as measurement to evaluate the performance of Thor's video decoder on Corework's platform without any hardware acceleration yet. The results were obtained by running the decoder for each video sample, and accessing the total of cycles spent (through Coreworks’ profiler). Dividing the frequency used (120 MHz) by the total of cycles and subsequently dividing by the number frames (20 frames), the frame-rate is obtained, as presented in table 5.1.

The results presented in table 5.1 show that Thor’s video decoder achieves real-time performance for all tested video samples, except highest resolution video sample (7). It is important to notice that, although some video samples have the same resolution, the performance can be very different, as demonstrated by video sample (1) and (2). This can be explained by the amount of motion existent in each video sample. In general, the more motion a video sample has, the worse the achieved performance obtained is.
The second test was performed using the same video samples, in order to determine the speedup achieved by each of the hardware implementations in the accelerated stages. The speedup was determined dividing the processor clock cycles spent for each stage, prior to the hardware implementation, by the processor clock cycles spent after the hardware implementation. In table 5.2 are presented the results of the speedup obtained by each of the hardware implementations used for the inverse transform stage. It can be observed that a speedup up to 1.433, for video sample (4), can be achieved using an hardware accelerator on the inverse transform stage.

<table>
<thead>
<tr>
<th>Video samples</th>
<th>Hardware Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Odd</td>
</tr>
<tr>
<td>(1) grandma_qcif.yuv (176x144)</td>
<td>1,236</td>
</tr>
<tr>
<td>(2) foreman_qcif.yuv (176x144)</td>
<td>1,179</td>
</tr>
<tr>
<td>(3) akiyo_qcif.yuv (176x144)</td>
<td>1,202</td>
</tr>
<tr>
<td>(4) coastguard_cif.yuv (352x288)</td>
<td>1,433</td>
</tr>
<tr>
<td>(5) foreman_cif.yuv (352x288)</td>
<td>1,205</td>
</tr>
<tr>
<td>(6) waterfall_cif.yuv (352x288)</td>
<td>1,236</td>
</tr>
<tr>
<td>(7) big_buck_bunny (854x480)</td>
<td>1,284</td>
</tr>
</tbody>
</table>

Table 5.2: Speedup results obtained for the inverse transform stage, for each hardware implementation performed.

The analyse of table 5.2, shows that the Odd parts implementation obtained the best results, for the whole spectrum of video samples. Regarding the remaining two implementations, even though the difference is small, the Kernel implementation achieves better speedups overall than the Odd-Even parts implementation.

Having the same operations implemented in hardware, the difference in results between Kernel and Odd-Even parts implementations, can be explained by the impact of the kernel function used on the former. The purpose of the kernel function is to eliminate the latency necessary to initialize all datapaths except the first one, this is achieved by starting the next datapath while the previous datapath is still running.

Concerning the Odd parts implementation, the higher speedup achieved in contrast with the other implementations can be explained by this implementation only containing the largest computation in the
one dimensional inverse transform. On the other hand, the Odd-Even parts and Kernel implementations contain the smaller computations as well, that turn out not to be worth implementing in hardware, since this computations are exceedingly low.

<table>
<thead>
<tr>
<th>Video samples</th>
<th>Hardware implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) grandma_qcif.yuv (176x144)</td>
<td>1.029 1.000 1.008</td>
</tr>
<tr>
<td>(2) foreman_qcif.yuv (176x144)</td>
<td>1.015 1.003 1.005</td>
</tr>
<tr>
<td>(3) akiyo_qcif.yuv (176x144)</td>
<td>1.017 1.002 1.009</td>
</tr>
<tr>
<td>(4) coastguard_cif.yuv (352x288)</td>
<td>1.031 1.010 1.026</td>
</tr>
<tr>
<td>(5) foreman_cif.yuv (352x288)</td>
<td>1.021 1.004 1.013</td>
</tr>
<tr>
<td>(6) waterfall_cif.yuv (352x288)</td>
<td>1.031 1.008 1.012</td>
</tr>
<tr>
<td>(7) big_buck_bunny (854x480)</td>
<td>1.079 1.057 1.062</td>
</tr>
</tbody>
</table>

Table 5.3: Speedup results obtained for the decoder, for each hardware implementation performed in the inverse transform stage.

Table 5.3 shows further results related with hardware implementations performed in the inverse transform stage. In this case, the achieved speedup of the implementations is measured regarding the whole decoder. As expected, the speedup values decrease significantly, since the inverse transform stage constitutes only a part of the decoder’s full processing time. Nevertheless, some speedup is achieved, specially by the Odd parts implementation.

In table 5.4 are presented the results of the speedup obtained by the hardware implementation used in the dequantize stage.

<table>
<thead>
<tr>
<th>Video samples</th>
<th>Hardware Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) grandma_qcif.yuv (176x144)</td>
<td>0.94</td>
</tr>
<tr>
<td>(2) foreman_qcif.yuv (176x144)</td>
<td>0.93</td>
</tr>
<tr>
<td>(3) akiyo_qcif.yuv (176x144)</td>
<td>0.92</td>
</tr>
<tr>
<td>(4) coastguard_cif.yuv (352x288)</td>
<td>0.99</td>
</tr>
<tr>
<td>(5) foreman_cif.yuv (352x288)</td>
<td>0.92</td>
</tr>
<tr>
<td>(6) waterfall_cif.yuv (352x288)</td>
<td>0.95</td>
</tr>
<tr>
<td>(7) big_buck_bunny (854x480)</td>
<td>0.98</td>
</tr>
</tbody>
</table>

Table 5.4: Speedup results obtained for the hardware implementation performed in the dequantize stage.

The results of table 5.4 show that the hardware implementation performed in the dequantize stage did not achieve any acceleration, for any of the video samples. This results support the fact that Thor video standard contains a simpler implementation of the dequantize stage. Concretely, the overhead induced by the initialization of sideworks, together with the transfer of data to the datapath, is not compensated enough by the processing time saved on the hardware implementation of the simple operations performed in the dequantize stage.

The overall results show that it is possible to implement a video decoder in Coreworks’ platform as well as achieving speedups through the use of hardware accelerators.
Chapter 6

Conclusions

The increasing diversity of services requiring efficient video compression and the desire for higher video quality and resolutions, motivate the video encoding field of research to be in a continuous state of evolution. Thor video standard is one of the most recent additions to this area.

The targeted platform for the implementation of a video decoder is the platform developed by Coreworks, that combines a 5-stage pipeline embedded processor (Fireworks) with reconfigurable hardware accelerators (Sideworks), that have been mainly used for audio solutions. As a result of the study of different video codecs, Thor’s video decoder was selected to be implemented in the platform, and accelerated through hardware implementations.

The implementation of Thor’s video decoder in the targeted embedded platform without any acceleration, required porting the original C code to the Fireworks embedded processor. Four hardware accelerators were implemented in order to speed up the video decoder. One accelerator for the dequantize stage and three alternatives for the inverse transform stage. Even though the acceleration of the dequantize stage was completely implemented in hardware, no speedup was achieved, due to the small amount of computations executed. Nevertheless, its implementation was useful in order to get acquainted with the development environment and hardware platform. However, the implementation of the hardware acceleration of the inverse transform resulted in some acceleration, especially for the Odds part implementation, which was able to reach a speedup up to 1.43. The Kernel implementation used a kernel function, allowing a quick succession of datapaths to be performed, however, due to the reduced number of computations of the last datapaths, the acceleration was lower than the one achieved by the Odd parts implementation.

For future work, there is a lot of room for further improvements and new implementations. In the hardware implementations performed, one improvement would be to take advantage of the correlation between the dequantize and inverse transform stage and implement a kernel function of datapaths containing each of the stages. Regarding the inverse transform stage, a possible improvement would be to perform each of the one dimensional inverse transforms in a single datapath, and subsequently perform a kernel function with the two dimensions.

Naturally, in a general way, any additional hardware implementations of other stages in Thor video
decoder, would benefit the speedup results obtained (if the implementation results in a speedup of the process). One should also consider the modification of Corework’s communication interface in order to allow video streaming to the platform for decoding.
Bibliography


