POLITECNICO DI TORINO

Master's degree in Electronic Engineering

Master's Degree Thesis

FPGA-Based Framework for Hardware Acceleration of the HEVC Encoder

Supervisor
Maurizio Martina

Candidate
Biagio Feraco

July 2018

Abstract

HEVC (High Efficiency Video Coding), the most recent standard for video compression, has led to an huge increase of the encoding efficiency with the respect to the previous standards but, on the other hand, several tasks have became highly time demanding and power consuming. This thesis proposes a fast, dynamically reconfigurable and flexible hardware accelerator for the SATD (Sum of Absolute Transformed Differences), that is one of the cost functions adopted by HM HEVC reference software. The developed hardware block follows all the specification given by HEVC and can work with every Transform Unit size from 8×8 up to 64×64. The accelerator is fast thanks to a highly parallelized architecture. It exploits a datapath subdivided into 4 pipeline stages for the time optimization for the different SATD sizes. This architecture was compiled and adapted in order to work on an FPGA system called DE1-SoC by Intel Altera. On this board is present a Cyclone V, that includes an ARM processor and the architecture was interfaced with it through its buses. Using the microSD interface it was possible to run a Linux distribution and the usage of the FPGA peripheral was integrated in the HM reference software. After that two main optimizations were introduced, the first one was the clock gating, in order to reduce the power consumption, and the other one was the utilization of the DMA inside the HPS, to increase the throughput of the data from the processor memories to the FPGA. At the end, the adders present in the datapath were substituted with Error Tolerant Adders, with the aim of analyzing the effects of approximation on the Rate Distortion.

Contents

List of Figures

List of Tables

List of Acronyms

AI	All Intra
ALM	Adaptive Logic Module
ALUT	Adaptive Look-Up Table
AVC	Advanced Video Coding
CB	Coding Block
CTB	Coding Tree Block
CTU	Coding Tree Unit
CU	Coding Unit
DCT	Discrete Cosine Transform
DPB	Decoded Picture Buffer
FME	Fractional Motion Estimation
GOP	Group Of Pictures
HAD	(see SATD)
HEVC	High Efficiency Video Codec
IME	Integer Motion Estimation
LAB	Logic Array Block
LDB	Low-delay B
LDP	Low-delay P
ME	Motion Estimation
PB	Prediction Block
PE	Processing Element
PSNR	Peak Signal to Noise Ratio
PU	Prediction Unit
QP	Quantization Parameter
RA	Random Access
SAD	Sum of Absolute Differences
SATD	Sum of Absolute Transformed Differences
SIMD	Single Instruction Multiple Data
SSE	Sum of Squared Errors
TB	Transform Block
TU	Transform Unit

Chapter 1

Introduction

In the last twenty years it has been possible to observe an exponential increase of the demand for video applications. Those types of applications include several categories like HDTV television, video conferencing, online video streaming on mobile and other types of devices, 24 hours video security system and many others. In particular, in the last years the 4K and in some cases 8K resolution TVs are growing in the market. This increase of resolution requires a better definition, that means a better quality for the viewer and this has as effect the need of a larger memory capability in order to sustain the high-definition standards or the larger bandwidth required for real-time application. To fulfill this demand, it was, therefore, required a continuous evolution in video compression standard, to increase the amount of compression. The video compression is a process in which the amount of information contained in a video stream is reduced as much as possible by acting on the internal redundancy. A video stream is composed by a sequence of images, called pictures or frames, reproduced over time with a temporal correlation between one frame and another. In video compression spatial and temporal redundancy reduction is exploited, unlike for image compressions (such as JPEG) in which involves only spatial correlation, because there are not variation of the image over time. Since a lossless compression does not give any advantages in terms of information reduction, a lossy compression is mandatory an it also maintain a very good video quality even if the original video is different to a decoded one. Motion compensation is the way in which the redundancy in a video is removed: in practice when some objects are moving between consecutive frames are found several techniques are applied in order to remove that redundant information. In this chapter will outline an initial digest of the video data representation and how is structured a typical video sequence. Further details can be found in [3].

1.1 Video Signal Representation

The different representations of video signals standards of nowadays strongly depends on past history. They have their origin in the history of the development of analog video signal format, that were different between Europe and United States. Starting from the black-and-white television, pictures were generated by the excitation of the phosphor on the television screen using an electron beam and modulating its intensity in order to generate the image. Each second the electron ray was continuously performing a complete scan all over the screen driven by the video signal. Therefore, a frame is depicted for each complete scan of the ray, making sure that if the number of frames for each second is enough to guaranteed no flickering for the human eye. In color television three kind of phosphors on the screen were introduced: red, green and blue and those were excited by three electron beams, each of them associated to a signal. Having a three times higher bandwidth would have led to backward compatibility problems with the black and white televisions. The solution was the creation of a special composite signal for the color televisions This would have brought to a three times higher bandwidth and would have created problems in backward compatibility to the black-and-white television. Therefore, the decision was to create a composite signal for the color television based on the red, green and blue intensity, well known as RGB. This signal is composed by three components:

- Luminance (Y), that is exactly the former black-and-white signal compatible with Black and White televisions decoder,
- Chrominance C_b and C_r that are the Blue-Difference and Red-Difference chroma components, related to the luminance component.

In Formulas 1.1, 1.2 and 1.3 is explained how to obtain the previous signals, derived from the RGB:

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \tag{1.1}$$

$$C_b = B - Y \tag{1.2}$$

$$C_r = R - Y \tag{1.3}$$

The color television retrieves from the inverse of these formulas the R, G and B signals. The Y, C_b and C_r signals were converted in digital format after the digital revolution. During the seventies a new worldwide sampling standard was introduced by the International Consultative Committee on Radio (CCIR) and it was called CCIR recommendation 601-2. In this standard all sampling that are multiple of the frequency of 3.725 MHz are admitted. Every picture that composes a video is a rectangular array of samples, each of those containing the three components that can be sampled with up to four times the base frequency 3.725 MHz. The sampling rate is is described using

three integers that carry the sampling rate for luminance and chrominance components. The three components are referred in the digital system as Y, U and V. These values are normalized once sampled, so that the resulting value for the luminance Y_s must be in the range between 0 and 1 and the ones for the chrominance C_{rs} and C_{bs} in the range of $-1/2$ and $1/2$. Then, according to the formulas 1.4, 1.5 and 1.6 the normalized values are converted to 8 bits values:

$$Y = 219 \cdot Y_s + 16 \tag{1.4}$$

$$U = 224 \cdot C_{bs} + 128 \tag{1.5}$$

$$V = 224 \cdot C_{rs} + 128 \tag{1.6}$$

Therefore, the Y component is in the range between 16 and 235 while the U and V ones between 16 and 240. Other video types uses more than 8 bits per sample, or *bit depth* (the most diffused ones between them have 10 bits per pixel).

1.2 Video Sequence

As mentioned before, a video contains a time sequence of frames which are compacted using *Motion Compensation*, meaning that instead of include all the samples for the pixels, parts of the frame are searched in previous, subsequent frames and the distance between the two is computed. This value is called motion vector, and it is the information that replace the redundant block. This means that some frames have several coding blocks that use prediction starting from other frames. A reference frame for those that use prediction is periodically inside the sequences in order to avoid a single reference, meaning that the decoding of a sequence does not start always from the first frame. These frames are called Intra Predictive pictures (**I**). The **I** frames make possible to start to watch a video from a point in the middle of the sequence, skipping all the previous pictures. The first frame is obviously an **I** type and reducing the interval between two consecutive **I** pictures reduce the buffering time but also the compression rate, resulting in a bigger sequence. The other types of frames used in motion compensation are Predictive Coded (**P**) and Bidirectionally Predictive coded (**B**) pictures and are used to reduce the temporal redundancy between two frames. The **P** pictures are encoded using motion compensation prediction referring to the previous frames, while the **B** pictures are similar to the **P** but take as reference for the prediction both previous and subsequent frames, and due to this they lead to a better compression efficiency.

Each interval between two consecutive **I** frames makes a Group of Pictures (**GOP**) that is the smallest unit for a random access. It may contain several **P** and **B** frames and it has to be defined to the encoder, depending on the kind of application. In fact Tv broadcasting may require random access frames, because the viewer does not tune

into a program always at its beginning, while for a video conference it is better to have less **I** pictures in order to reduce the encoding delay for the real time transmission.

Inside the **GOP** there are two different sequence order:

- Display Order, that is the chronological order in which the frames are shown on the display,
- Bitstream Order (or Encoding Order) that is used by the encoder and decoder.

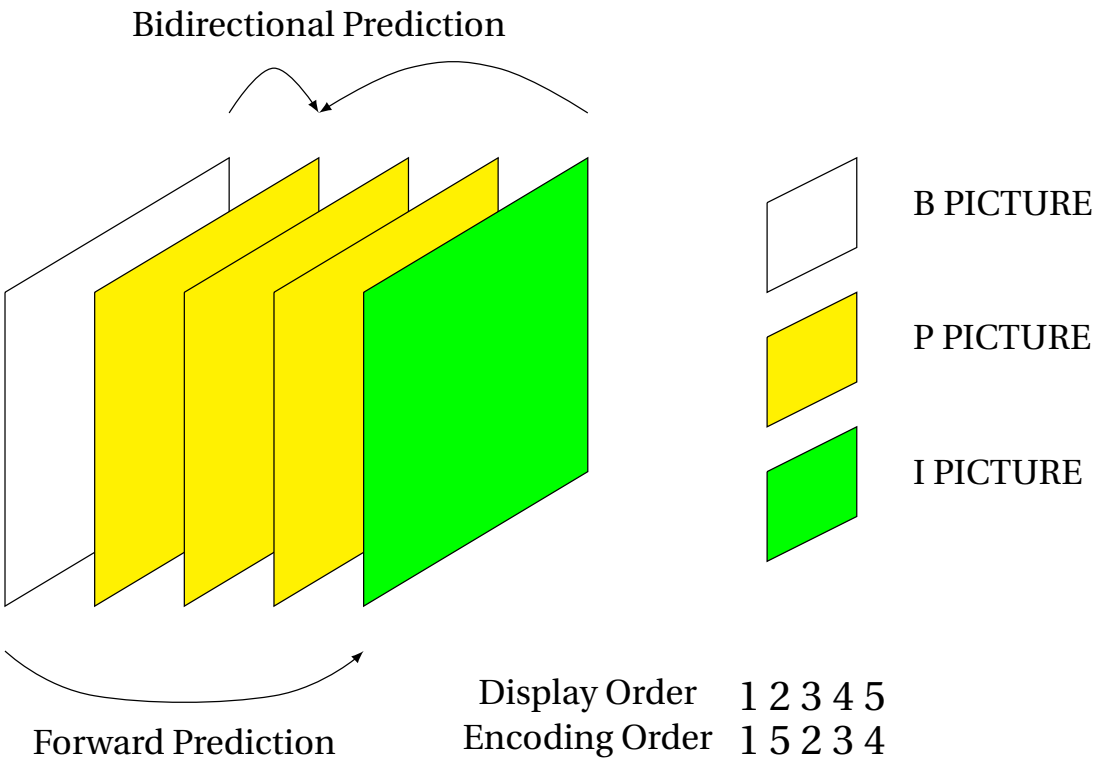


Figure 1.1: A possible structure of a GOP and difference in display and encoding order

Figure 1.1 shows a possible structure of a GOP and the difference between the two orders. The *encoding order* is adopted also by the decoder, that needs to analyze the reference frames first, and then those that are related with.

Chapter 2

HEVC

High Efficiency Video Coding (HEVC or H.265) is the video compression standard approved on 25 January 2013, successor of the H.264/MPEG-4 AVC (Advanced Video Coding), developed by the Moving Picture Experts Group (MPEG) and the Video Coding Experts Group (VCEG) from ITU-T, released with the name of ISO/IEC 23008-2 MPEG-H Part 2 and ITU-T H.HEVC Before HEVC the primary video coder was AVC and, in the last period, mobile devices began to be compatible with high-resolution video. In addition, new ultra high-definition resolutions (for example 4k or 8k), started to appear with the necessity of even more memory. Improving the *coding efficiency*, therefore, became the main target of HEVC developers. The coding efficiency is one of the metrics for evaluate the compression capability of a standard based on the dimension in bytes of the produced stream while maintaining a certain video quality. With AVC the coding efficiency was doubled with the respect to MPEG-2 maintaining the same quality standard, and the same thing happened with HEVC that doubles this parameter again without affecting the quality. Therefore, the encoder output bit-rate is reduced by at least 50%, as it is stated in [4]. The Peak Signal to Noise Ratio (PSNR) is usually associated as parameter that determines the quality of a video. Flexibility is another goal of H.265, in order to be compatible for all the video applications. HEVC standard is defined with the only description on how the encoder output bitstream has be constructed with no specification on the encoder or decoder internal structure, and this is the reason why still today it is in progress a continuous research to improve the implementations on both software or hardware, always with the focus on the improvement for compression efficiency. Of course, the encoding or decoding time reduction is one of the main goal of this type of researches, along with the focus on the power consumption. HEVC, compared to AVC, slightly increased the requirements for memory capacity, with a doubling of the computational resources as well as for the eventual hardware implementation. With HEVC it is easy to create parallel architectures to avoid serialized bottle-neck elements and, therefore, parallelism can bring to faster encoders, and it is becoming more and more important in hardware applications of this kind. H.265 is perfectly backward compatible with all the same

previous AVC applications and resolutions and it gives a particular attention to high frame rates and high-resolution videos. It is also suitable for 4:2:0, 4:2:2 and 4:4:4 luma and chroma resolution ratio and for bit depth of 8 up to 16 bit. Resolution ratio and bit depth are properties of the HEVC profiles: the Main Profile is characterized by a bit depth of 8 bit and a 4:2:0 format.

2.1 Encoder

In Figure 2.1 is represented the high-level reference diagram of a an HEVC standard compliant encoder. The video coding layer of HEVC is based on the typical *hybrid* approach, meaning that it performs inter-picture, intra-picture prediction and 2D transform coding with some key differences that enhance compression. The encoder, at every cycle, selects a frame from the input stream and subdivide it into partitions of blocks. Forn each of those block is then performed a prediction with **Intra** or **Inter** prediction. The first technique is used in I frames and it performs a comparison between blocks in the same picture and, thus, remove spatial redundancy. The second technique, instead, compares blocks of samples present in different frames, exploiting motion compensation and, thus, reduce temporal redundancy as explained in Chapter 1. Both Intra and Inter perform a prediction selecting a frame that has to be compared with a reference one. The results of inter prediction encoder are the motion vector that are sent to the coding stage, while in intra prediction phase it is generated an equivalent results called intra prediction data. After these stages, the Predicted Frame is compared with the original one, and the resulting Prediction Error will be linearly Transformed, Scaled and Quantized. The final stage is the Entropy Coding, which performs a compaction of the stream of the data produced, generating the Encoder output bitstream.

Frames are reconstructed by an internal decoder starting from the resulting coefficients and collects them inside the Decoded Picture Buffer. Deblocking and the SAO filters blocks filter the reconstructed pictures before they are stored into the DPB: while the Deblocking filter realize an attenuation of the discontinuities at the boundaries of the blocks, the SAO filter task is to reduce the artifacts. More informations about the argument can be found in [5] or [6].

2.1.1 Block Partitioning

The partitioning system that was present in the earlier video standards was badsed on the subdivision of the frame in macroblocks, each one composed by a 16×16 luma samples and the relatives 8×8 samples for chroma. However a maximum dimension of 16×16 for large resolutions is not efficient in terms of compression rate, because to assure a minimum quality, details in an high-definition video has to be analyzed with more accuracy. In addition, the compression efficiency for low resolutions requires

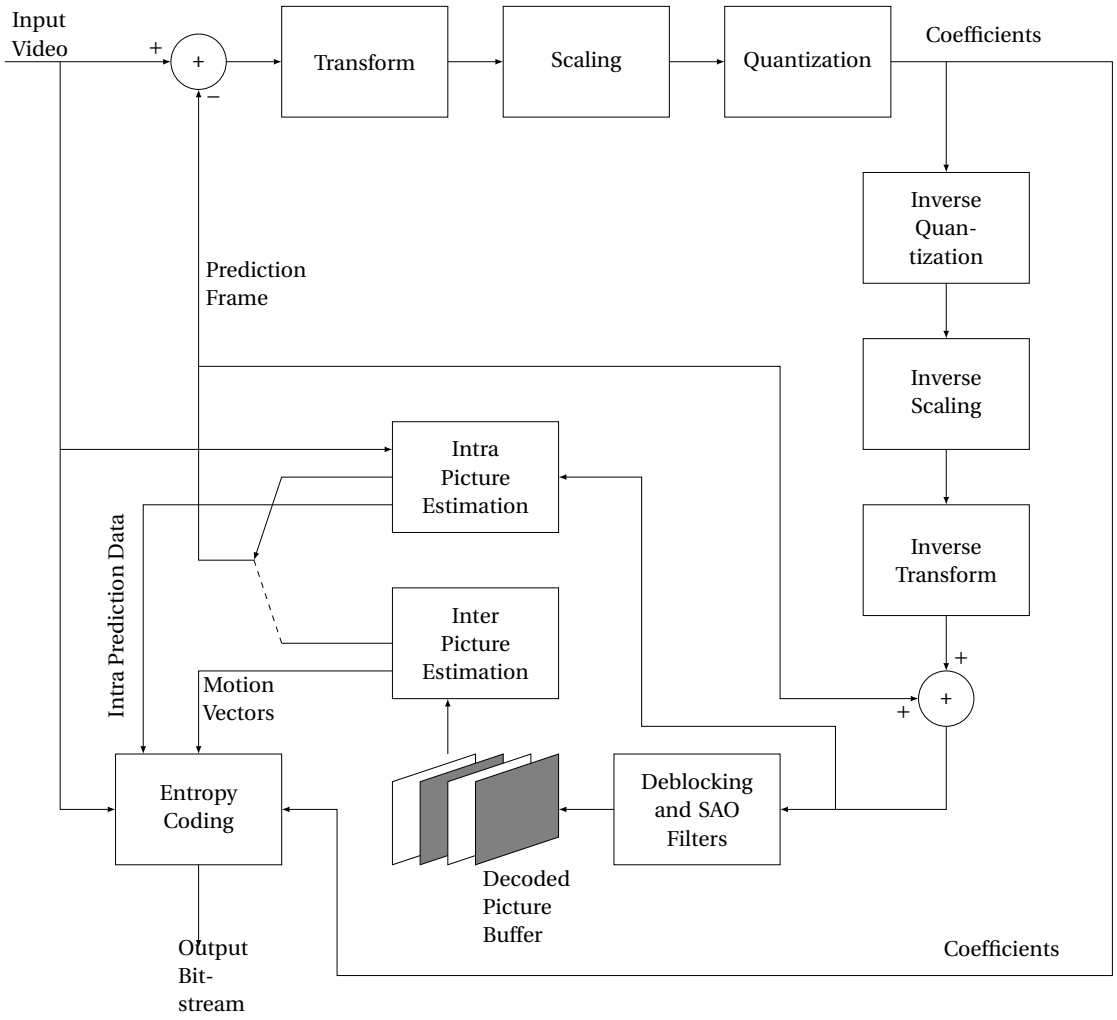


Figure 2.1: HEVC encoder block scheme

small blocks of pixels. The solution adopted in HEVC was the creation of a flexible mechanism for the subdivision of the pictures into variable size sections. The encoder will, therefore, perform a trade-off between the distortion and the bitrate, considering the Formula 2.1:

$$RD_C = D + \lambda \cdot Q \tag{2.1}$$

where D is the distortion metric, Q is the resulting rate and λ is the Lagrange Multiplier [4]. While larger blocks allow to reduce the bit rate, since they contain much more information, reducing the block dimension increases the quantity of information as well as the bit rate. In H.265, each frame is divided into several and disjunct blocks with different creating a quadtree structure with a recursive subdivision until the minimum size allowed: this structure is are called Coding Tree Blocks (CTB) and it

is applied for both luma and chroma samples: in 4:2:0 format, a luma CTB consist in an area of $2M \times 2M$ luma pixels while the two chroma CTBs corresponds to an area of $2M-1 \times 2M-1$ chroma samples, with M that can be 4, 5 or 6, meaning that luma CTB can consist in an area of 16 × 16, 32 × 32 and 64 × 64 samples. The HEVC basic compression parallel processing unit is the Coding Tree Unit (CTU) that is composed by the luma CTB and the corresponding chroma components. Introducing larger CTUs could give good improvements in terms of coding efficiency for high resolutions, but as a consequence the computational complexity could raise, as well as the encoder delay.

Following to the quadtree subdivision scheme, a CTU can be split into different Coding Units (CU) with variable size until the 8 × 8 block size. Each CU contains a luma CB and two chroma CBs. The CU is the processing unit that will decide whether intra or inter prediction is applied on its CBs. In addition, a CTU can be split into several Coding Unit (CU) of variable dimension sides, with the smallest dimension of 8 × 8, that can be further subdivided into Prediction Units (PU) and Transform Units (TU), which the latter is the base element for the intra-picture prediction mode, which it is based on square blocks.

The last subdivision can be applied to a 8 × 8 luma CU, splitting it into four 4 × 4 TUs. The accepted dimensions are those in the range of 4×4 up to 64×64. In total, they are: 4 × 4, 8 × 8, 16 × 16, 32 × 32 and 64 × 64. On the contrary Inter prediction is based on PUs. The subdivision into PUs is independent from the TU subdivision. There are eight different mode to split a CU into PUs since for CUs bigger than 8 × 8 it can be employed asymmetric partitioning modes, as shown in Figure 2.2 and therefore, also rectangular dimension are employed. The smallest PUs have size 8 × 4 or 4 × 8. All the information about the chosen partitioning are included inside the output bitstream so that the decoder can divide the frames in the same way.

2.1.2 Intra Prediction

In Intra-picture prediction the decoded samples from the adjacent blocks at the boundary are used as prediction to produce data in order to reduce the spatial redundancy within the same picture and it is related to I frames. It consists of three steps: reference sample array construction, sample prediction, and post-processing. Several studies and research were conducted in order to minimize these steps the computational requirements, specially for the encoder. Intra prediction is performed executing the the comparison between samples, with a size from 4×4 up to 64×64. The selected TB will be compared with ones on the boundaries of the neighboring decoded TB, until the best match is found. At this point the resulting data is generated selecting one of the 33 angular mode provided by this standard plus one for the planar and one for the DC mode prediction used for smooth image areas.

If no current samples are available because, for example, they are positioned outside the picture or they have not yet been decoded, a default value is taken, otherwise

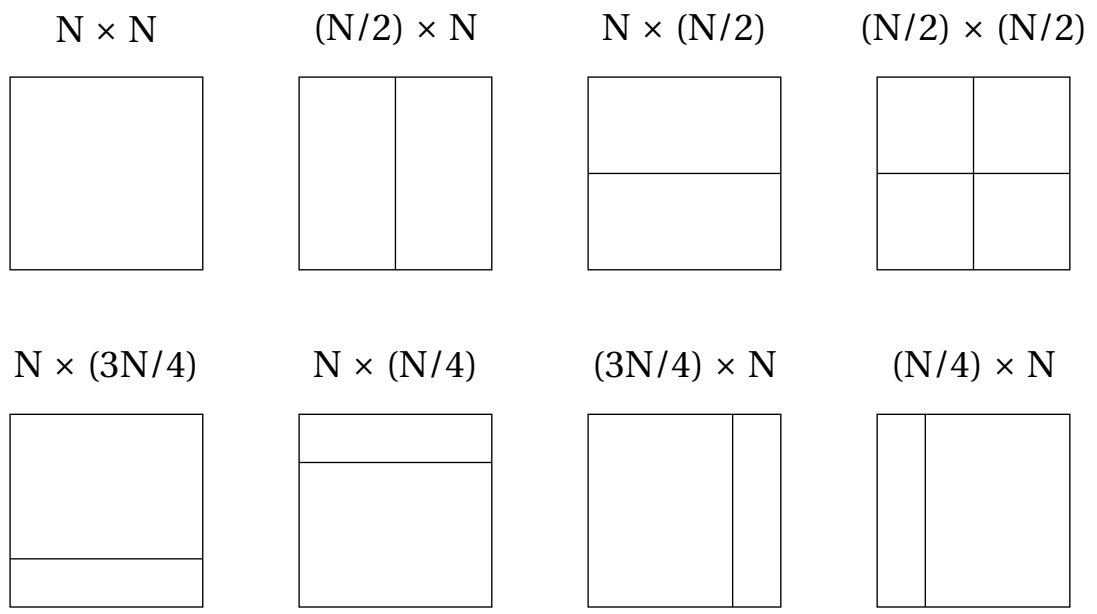


Figure 2.2: CU partitioning modes used for deriving the PUs. The $(N/2) \times (N/2)$ mode and those in the second row can be applied only if the CU size is larger than 8×8 luma samples

if at least one reference sample is present the others will be substituted with its value. The current samples can pass through a smoothing filter (not used in case of DC Prediction) to reduce the discontinuities at the boundaries of the block that can potentially lead to a wrong prediction in terms of angular mode. Other discontinuities in the predicted samples are generated on the boundaries of the TB, and, therefore, it may be applied a further post-processing smoothing step: for DC mode and horizontal or vertical mode, the filtered values will substitute the samples at the boundary, then a block of samples is predicted and finally the best match is found from the result of the formula 2.1, in with D is the result of a cost function that, in case of HM reference software, is **Sum of the Absolute Transformed Differences**, or **SATD**). A complete description of the Intra Prediction process can be found in [7].

2.1.3 Inter Prediction

In Inter-picture prediction the decoded samples from different frames are used as prediction to produce data, by employing motion compensation, in order to reduce the temporal redundancy, and it is referred to the P and B frames. As mentioned in 1.2 for each block, a search from the previously decoded frames is performed. Using a cost function, in this case **Sum of Absolute Differences** or **SAD**) the encoder has to find the best match, that minimize the Formula 2.1, between two blocks of the same

dimension. The result is the motion vector composed by values Δx and Δy indicating the horizontal and vertical displacements between the two blocks position. The reference frames indexes (Δt) together with the motion vectors are, then, provided to the entropy coding unit. Δx, Δy and Δt are called motion data. As mentioned in 2.1.1, Inter-prediction is based on PBs, and not on TBs as for Intra-prediction. The difference from a canonical encoder and HEVC is that in the latter there is provided with two types of inter-picture prediction: the *uni-prediction* and *bi-prediction*. In Bi-prediction the module generates two pairs of motion vector and it computes the average with the result of a more precise prediction. Moreover, the search for the best match is performed in two steps. The first one is called **IME** (*Integer Motion Estimation*), in which the starting from a current block, it will be compared with several reference blocks within a range that is by default of ±64 pixels, and, by employing the sum of squared differences method, the best match will be selected and the motion vector will be calculated. After this step, the integer samples are interpolated to perform the prediction with fractional precision. This step is called **FME** (*Fractional Motion Estimation*). In summary the IME does a coarse search inside the whole search window, while the FME redefines that performing a fine grain search. In HEVC are present interpolation filter with a quarter-pixel accuracy motion vectors for luma and one-eighth-pixel accuracy motion vectors for chroma components.

2.1.4 Transform

This unit realizes a spatial transform on the residual with the aim to compact the differences into less low-frequency coefficients so that the quantizer can neglect those with small amplitude without affecting the video quality and improving the coding efficiency. This unit realizes a spatial transform on all the TB residual, performing a two-dimensional Discrete Cosine Transform (DCT) acting firt of all on the block rows with one-dimensional DCT and then over columns with a second DCT. Only for 4×4 block size the Discrete Sine Transform (DST) instead of DCT because DST is performed, due to 1% of improvement in bit-rate reduction fr intra-picture coding with the rect to the DCT.

2.1.5 Quantization

The quantization phase performs a division of the transformed coefficients by a quantization step (Qstep) and then the result is rounded. Qstep is derived from the input parameter QP (Quantization Parameter) specified in the configuration file. QP can be selected in a range from 0 to 51 for an 8-bit sequence. Increasing by 1 the QP, Qstep grows of 12%. Qstep is equal to 1 when QP is 4. With a low value in QP the video quality reduces and vice versa.

2.1.6 Entropy Coding (CABAC)

Context-adaptive binary arithmetic coding (CABAC) is a lossless compression technique, based on entropy encoding, well adopted in HEVC and in the previous H.264 standards. It provides a much better compression than other encoding algorithms used in video encoding base on entropy.

It is based on arithmetic coding, and:

- It encodes binary symbols, keeping the complexity low and using a probability model that take into account more frequently used bits of any symbol.
- The probability models are dynamically adapted based on local context, allowing better modeling of probabilities, reducing the more usually correlation in locally coding modes.
- It employs a multiplication-free range division, based on quantized probability ranges and probability states.

The steps performed by the CABAC unit are:

- *Binarization*: a sequence of bins (bin) is assigned for each.
- *Context Modeling*: Selection of an adaptive binary probability model.
- *Binary Arithmetic Coder* (BAC): Arithmetic Encoder that performs a compression of the bins into bits according to the selected probability model.

In the Context Modeling can be selected two modes:

- *Regular mode*: the Probability Estimator and Assigner select a probability model for the actual bin.
- *Bypass mode*: bins are coded with equi-probability, since no context model is required.

2.1.7 In-Loop Filters

Before saving the decoded pictures inside the Decoder Picture Buffer, those values will pass through two filters: the *Deblocking* and *SAO filters*. The deblocking filter is applied first and it is used to attenuate discontinuities (or block artifacts) at the PB and TB boundaries. This filter does not work well on the edges of the picture or inside the CUs, but only on the boundaries between CUs samples. The Sample Adaptive Offset (SAO) filter is applied to the output of the deblocking one and has the purpose of removing the ringing artifacts. It works on entire CTUs and, in order to smooth, it adds to the samples negative or positive offset provided in a look-up table generated by the encoder.

2.2 HEVC HM Reference Software

The HM software provides a reference implementation of both HEVC encoder and decoder. It is an open-source code written in C++ and developed by JCT-VC (Joint Collaborative Team on Video Coding) from ITU-T. All the simulations of this work were performed using the HM 16.15 version [8]. JCT-VC provides also several reference test sequences [9] and Common Test Conditions. Those test conditions are referred to a list of video sequences for different resolutions and frame rates which are considered as a reference. In the common test conditions is defined the complete set of encoder configurations that has to be used in experiments with the HM code. They can be classified into four types of configurations:

- All Intra (AI): on each frame is performed the Intra-picture prediction, therefore there is no Motion Estimation usage and every picture is an I type.
- Random Access (RA): it uses a GOP composed by both B and I pictures and I frames are inserted periodically in the sequence. It is the main configuration for video broadcasting.
- Low-delay P (LDP): The GOP is composed by type P frame, while only the first frame is an I type. It is the main configuration for videoconferencing.
- Low-delay B (LDP): The GOP is composed by type B frame, while only the first frame is an I type. It is the main configuration for videoconferencing.

Chapter 3

Methods

In the previous Chapter, it was stated that HEVC is able to increase by a factor of 2 the compression capability with respect the previous AVC video maintaining the same video quality, since several new features were introduced: more block sizes, new intra and inter prediction modes, more complex interpolation filters, more efficient in-loop filters, and other features. On the other hand, the complexity of the encoder increased as well as the computational cost that raised with a factor between 40% to 70% with the respect to the AVC encoder, as stated in [10]. This is mainly due to the higher encoder decision space to be explored. Moreover the growth in the memory access number that is 3.86 times higher with the respect to H.264, as explained in the [10] analysis, and the increase of the power consumption and temperature in ASIC or processor encoders were the other important factors. In fact the more the functions implemented are complex, the higher is the density of transistors need to implement them. Besides, as said in [11], the encoding time itself is very slow and far from real-time applications. These reasons led to several researches that have as objective the reduction of the complexity and the computational effort of the encoder, acting on the encoding process in order to near the real-time coding especially for high resolution sequences with an high frame rate, or in order to reduce the power consumption and temperature of the HEVC hardware. In the solution defined in [10], the author suggests the design of several low power accelerators that could help to reduce the computational time of intensive functions and a particular HEVC memory hierarchy. He also suggests the creation of new software algorithms in order to limit the computational load, with no several effects on the video quality. In this work is part of the research for the creation of a fast system for HEVC. This system will be integrated on an FPGA that has a microprocessor inside that will run the software and activate the system when required. Therefore this system has to compute the high-latency processes using several hardware accelerators.

3.1 Complexity in HEVC

In [11] is present a complete analysis of the HM encoder (in this case study was used the HM 8.0 version), using AI and RA as common test conditions. In All-Intra configuration a significant amount of time is spent for the rate-distortion optimized quantization (RDOQ), performed by the TComTrQuant class, transforms contribution is 9%, while TComPrediction and TComPattern classes for Intra prediction further accounts for close to 16%. In the random-access configuration it can be noticed that motion estimation takes a significant portion of encoding time, due to the computation of sum of absolute differences (SAD), sum of absolute transformed differences (SATD) and other distortion metrics present in TComRdCost class which their contribute is about 40% of overall encoding time. Furthermore the TComInterpolationFilter class accounts for 20% of encoding time. One of the most time consuming metrics is the SATD, as states in [12]. While the Sum of Absolute Differences (SAD) is the sum of absolute values that is applied to the residual block (a difference between the original block and a reference block) ,Sum of Absolute Transformed Differences (SATD) is the sum of absolute values of the coefficients obtained when the Hadamard Transform is applied to the residual block. SAD and SATD are different metrics to estimate the distortion between two video blocks in mode decision stage of video encoders. SATD achieves better distortion estimation than SAD, but it is more complex. For this reason, in HEVC reference software [8], SAD is applied in the most frequently executed steps in video encoder, e.g. Integer pixel Motion Estimation (IME), while SATD is applied (when enabled) for intra prediction mode decision and for Fractional pixel Motion Estimation The SATD based on 8×8 Hadamard Transform contribute is in the range between 9% and 19% of total execution time of HEVC video encoder. It happens because the HEVC encoder software computes less 4×4 Hadamard SATD more often with 8×8 Hadamard Transform than withwhen considering ultra-high resolution videos. HM encoding time is very high. It could lead to several hours of computation. Therefore in order to speed up the process, several improvements either for software and hardware are essential. The analyzed profiling made by [13] reveals that the the cost functions, together with the interpolation filter and the quantization contributes for the 80% on the total encoder time. Those cost functions, that are the main contribution, are the Sum of Absolute Transformed Differences (SATD), the Sum of Absolute Differences (SAD) and the Sum of Squared Errors (SSE). The SATD metric, compared with the SAD one, is more computationally complex, and, therefore, the latter is the most used in the video encoder, especially in Integer pixel Motion Estimation (IME), while the former is applied in Intra Prediction mode decision and (when it is enabled) for Fractional pixel Motion Estimation (FME), that are the process inside the encoder that require an higher accuracy. In fact, as stated in [14] when SATD metric is enabled in FME, the bit rate is reduced up to 2.2%, with an increase on the video quality of 0.16 dB.

HM reference software uses two main function to calculate the SATD. The first one

is the 8 × 8 Hadamard Transform that is applied on all the blocks that are composed by 8 × 8 blocks of samples, while the second one is the equivalent for 4 × 4 blocks.

From the profiling of [12], while the contribution of the 4 × 4 is up to 2% of total execution time of HEVC video encoder, the one based on 8 × 8 Hadamard Transform contributes in 9% to 19%, due to the fact that the 4 × 4 valuation is less performed than the other one, especially in high resolution video. To overcome this, several solution were adopted in the HM code, as the utilization of SIMD function (SSE for Intel, NEON for ARM). Starting from the concept of [12], the final decision was to implement an hardware accelerator for the SATD metric, trying to improve as much as possible the time performances.

3.2 SATD Accelerator

The SATD accelerator that has to be implemented on the FPGA must be able to perform in hardware all the SATD functions types inside the HM reference software. It has to dispose of the following features:

- Fast in to order to speed-up the encoder execution,
- Dynamically reconfigurable to compute the SATD for all the TU sizes re-using the same hardware resources, curtailing also the power consumption that must be taken into account because it could increase a lot due to the high intensive functions computation.
- Presence of a local on-chip memory buffer in order to reduce the accesses from the off-chip main memory, and to have a continuous load of data, as mentioned in [10], or better an internal memory cache to store parts of the pixel blocks to be analyzed.

One of the most critical constraints during the hardware implementation will be the requirements on the memory bandwidth and it will be one of the main focus for this work. The system has to perform the SATD operation when requested, and a Controller must handle the communication as a Master. For the validation this Controller will be emulated in a VHDL testbench, and later on will be directly the software. It has to communicate to the SATD accelerator the size of the TUs on which performing the cost function and sending into its local memory buffer the TUs samples.

3.3 Design Flow

This Chapter concludes the theory part of the thesis. The following Chapter will describe the creation base architecture for the SATD accelerator starting from the specifications listed in this Chapter. This architecture will be compliant with all the features

explained in 3.2. Chapter 6 will present the validation of the architecture by means of proper testbenches and software programs. Chapter 7 describes how the synthesis works on the Altera Quartus software and presents the first results. In chapter 8 will be described the steps in order to insert the hardware peripheral into the HM reference software. Chapter 9 describes the introduction of two optimizations for the base architecture: Clock Gating and DMA Controller. The first one is employed to reduce the dynamic power, while the second is the usage of a Direct Memory Access, that is instructed from the processor in order to send the data towards the FPGA on-chip memories. Chapter 10 presents all the works done to insert approximation inside the SATD accelerator by using Error Tolerant Adders and discusses the results. Finally, Chapter 11 will present the conclusive results and the future works.

Chapter 4

DE1-Soc and Quartus Prime

The entire project started with the purpose of the final implementation on a FPGA. The system that was selected was the **DE1-SoC Development Kit**.

In this system is included an Intel® Altera System-on-Chip (SoC) FPGA, which contains a dual-core Cortex-A9 embedded cores. The SoC includes two parts: the effective FPGA chip and the HPS, that includes the processor and the levels of cache, and all the interfaces. In addition two external DDR RAM memory are present in the system.

The relevant Hardware Contents of the Development Kit are:

- USB to UART (for the command-line interface);
- 1 GB (2x256Mx16) DDR3 SDRAM on HPS;
- Micro SD Card Socket on HPS (to store the Linux OS);

  The FPGA chip mounted on the board is a Cyclone V. A simplified block diagram of Cyclone V SoC is depicted in the Figure 4.1. As said before, Cyclone V SoC architecture consists of a Hard Processor System (HPS) and FPGA are combined togheter in te same chip, improving a lot the performances and the time-to-market.

The HPS features:

- a microprocessor unit (MPU),
- 64 kB on-chip RAM (HPS-OCR),
- booting ROM,
- SDRAM controller (SDRAMC),
- DMA controller (DMAC),
- Others HPS peripherals.

To connect HPS and FPGA, three bridges are present:

- HPS master and FPGA slave:

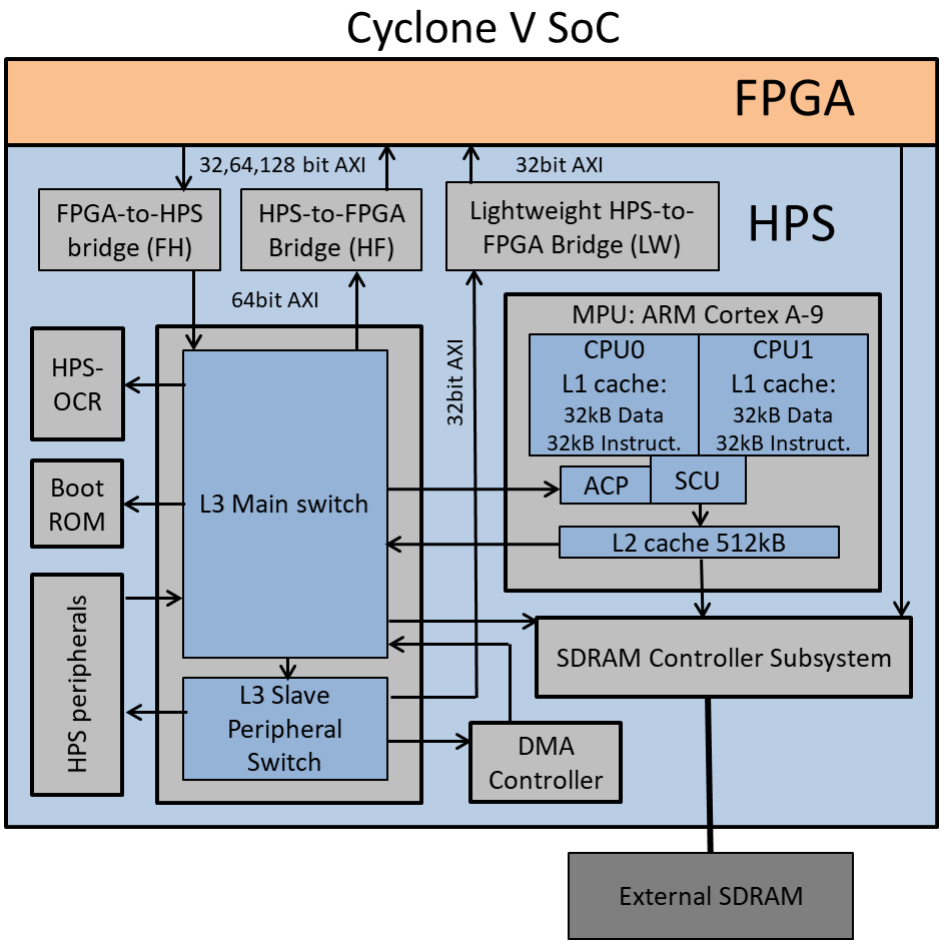


Figure 4.1: DE1-SoC Overview [1]

  - HPS-to-FPGA (H2F) bridge, a high-performance bus with data width is configurable as 32, 64 or 128 bit.
  - Lightweight HPS-to-FPGA (LWH2F) bridge, a 32-bit bus connected to the L3 Slave Peripheral Switch, with low performances and used mainly for configuratios.

- HPS slave and FPGA master:
  - FPGA-to-HPS (F2H) bridge.

4.1 Quartus Prime

In order to work with this type of FPGA Board it is fundamental to work with **Quartus Prime**, that is the software provided by Altera that allow users to compile, synthesize and upload your configuration on the board. Before starting the realization of the architecture, it was necessary to create the environment for interfacing the FPGA to the ARM core.

The Intel Quartus Prime software design flow comprises of the following high-level steps:

- Design creation.
- Apply the constraints.
- Compilation.
- Timing constraint (SDC).
- Configure of the design on the board.

4.2 Platform Designer

Quartus Prime contains several tools, and the most important for this work was **Platform Designer** (previouly known as Qsys). Using this tool is possible to create the exact environment in which insert the design. By the graphical user interface it is possible to interconnect your custom design to the bridges and also to add configurable components from the IP libraries of the software.

Figure 4.2: Platform Designer Interconnections Overview

The creation of a new component is essential in order to generate the interface of the FPGA design to the HPS. Inside this tool it is possible to select how many type of interfaces you want and correctly map them on the physical addresses.

4.3 Linux on the DE1-SoC

On the DE1-SoC Computer it can be mounted the Linux Operating System. Programming the microSD card, the OS will be booted by the ARM Cortex-A9 dual-core present into the chip.

The Linux Distribution used for this project is one provided by Intel Altera on its websites. In the manual provided [15] by Altera every step is exlained in the details, and it is possible to set up the microSD in order to have the system ready to use. Any technical content about the DE1-SoC, Cyclone V and its features, for example related to the IP libraries, was taken from [16].

Chapter 5

Architecture Description

5.1 Introduction

The developed architecture is an hardware peripheral fully compatible with the interfaces to the ARM processor able to compute the SATD operation on block sizes based on 8×8 block (listed in 5.1). The circuit contains an interface in order to configure it. The 4x4 block size SATD is not implemented in the final version because after a test on the timing it has been proved that it is not convenient to do it in hardware. As explained in [12], SATD is a metric to estimate distortion between two blocks for mode decision (Rate-Distortion Optimization) in video encoding. The generic equation for the SATD calculation is given by the following formula:

$$SATD = \sum_{i,j} \left| HT(i,j) \right| \tag{5.1}$$

In 5.1 $HT(i,j)$ is the $(i,j)^{th}$ coefficient obtained after applying an Hadamard Transform to the residual block that is the difference between the original block(a block to be encoded) and a reference block. The reference block can be an intra-predicted block to be evaluated for intra mode decision or an interpolated block for FME.

In HEVC reference software HM [8], the blocks smaller than 8×8 samples are evaluated with 4×4 Hadamard Transform. The 8×8 Hadamard Transform is applied to evaluate blocks of 8×8 samples or larger. The 8×8 Hadamard Transform ($HT_{8\times8}$) of a residual block Y is defined as:

$$HT_{8\times8} = H \cdot Y \cdot H^T \tag{5.2}$$

In which the 8×8 Hadamard matrix is:

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 \\ 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix} /4 \tag{5.3}$$

One of the techniques that allow to increase the timing performances is to apply **parallelization** and in order to do so, the block that performs the 8×8 Hadamard Transform is replicated several times. As first attempt a $\times 8$ parallelization was performed, but this type of implementation was unfeasible on the Cyclone V, because more LABs than those that are present in the FPGA (32070 LABs) were necessary and, therefore, a $\times 4$ was the final solution.

The basic idea therefore is to have an architecture that takes as input the two CU, extracts the residual and performs the matrix product with the Hadamard matrix and then computes the sum of each component of the resulting matrix.

Since the architecture works on different matrix dimensions it takes different clock cycles to calculate the result, varying from a minimum delay with 8x8 size to a maximum for 64x64 size. Moreover it works with two bridges for the communication of the input data, configuration and result, therefore the communication must be signaled with many acknowledge signals.

5.2 Algorithm Description

The architecture is composed by four main blocks, and each of them can compute the differences, the Hadamard Transform, the absolute values and the sum of 64 pairs of samples in parallel. The main inputs are coming from two fluxes of a reference and a current CU, and they are a sequence of matrices of pixels (with equal dimension for both reference and current streams) inside a single Coding Unit. The architecture analyzes these matrices as a set of 8×8 elements. For example a 32×32 matrix is evaluated as a set of 16 elements. Therefore the 8×8 is the base unit. Having as much as possible base unit working in parallel is a solution to reduce the elaboration of all the base unit that a CU is composed. The resources of the FPGA limited the parallelism to a degree of four, meaning that at each cycle up to 4 base units from the reference CU and 4 base units from the current are sent to the core. For example, a single 8×8 base unit must be sent entirely in only one cycle; a 16×16 block, that is composed by 8 base units can be also passed in only one cycle: instead a 64×64 block, that is composed by 64 base units has to be sent in 16 cycles. A 64×16 block, that is composed by 16 base units has to be sent in 4 cycles as for the 32×12 block that has the same number of base unit. Knowing that, it is possible to reduce the all possible cases of block 8×8 dimension Each base unit produces in parallel a result and then an adder tree sums all those values, and these result are further passed to another adder tree, depending on the type of the block under test. The datapath is therefore composed by 4 main units (MU) that works in parallel and each of them computes a 8×8 SATD in one cycle. This type of system requires two matrix of data that are composed

in a way that each MU must be related to an 8×8 CU in an unequivocal way. The memory structure requires therefore a reorganization of the block, meaning that in order to parallelize the computation of a single MU the entire block of pixel must be provided at the input of the structure at each cycle. With this type of reorganization it is possible to fed the datapath with four CU for each clock cycle. This means that for a simple 8×8 CU only 1 cycle is needed to feed one of the four MU, and the same for a 16×16 CU to feed all the 4 MU instead for a 32×32 and 64×64 CUs it takes respectively 4 and 16 cycles to send the block to the datapath. It can be notice that not all the MUs are always used. In fact in the 8×8 case (and only in that case) three out of four MU are not used, therefore in that case only one MU is activated and the others do not work.

5.3 Top Level

In the Figure 5.2 is shown how the architecture is inserted in the structure that Quartus generates. As said before, this architecture works with two clock domains. The clock speed for the memories, as said in 5.5.4, is the maximum clock speed that the Altera memory block can reach, while for the datapath, after several attempts, was set to 66.66 MHz. In Chapter 7 all the processes handled during the synthesis that brought to this settings will be explained. In Figure 4.2 is present the elements that are inserted and how they are interconnected:

- The Hard Processor System
- Clock Source
- PLL
- The Custom Component, that contains the architecture.

5.3.1 Hard Processor System

As explained in Chapter 4, inside the Hard Processors System (HPS) is present a MPU, that includes two ARM Cortex-A9 32-bit processors and two levels of caches. In addition the MPU has a Micro SD interface with $\times 4$ data lines, that can be used to boot a Linux OS. Therefore, after the introduction of this component in the Platform Designer environment, the tool generates the export for the latter connections to the external Double Data Rate 3 RAM memories. The DDR3 devices connected to the HPS are the exact same model as the ones connected to the FPGA. The capacity is 1 GB and the data bandwidth is in 32 bit, comprised of two $\times 16$ devices with a single address/command bus. The signals are connected to the dedicated Hard Memory Controller for HPS I/O banks and the target speed is 400 MHz. During the configuration of the HPS component, used bridges and its parallelism were specified. The configuration for the HPS can be found in [17]. In this case were used two of the three available: the HPS-to-FPGA AXI Bus and the HPS-to-FPGA Lightweight AXI Bus. The first one is used for the transmission of the data on the memories on the FPGA and therefore a parallelism of 128 bit (the maximum parallelism) was selected, while the second one is a low performance Bus and it is used mainly for configurations, and due to this reason a parallelism of 8 bit has been selected.

5.3.2 Clock Source and PLL

The Figure 5.1 shows the default frequency of all external clocks to the Cyclone V SoC FPGA. To reduce the jitter, a clock generator is used, providing four clock signals of 50 MHz for the logic and a clock source of 25 MHz for the HPS clock. In addition to the direct clock source, it is possible to use the PLL clocks from the datapath as source for the FPGA architecture. In this case one of the four clock signals has been chosen as clock source.

[16]In order to improve the performances, the usage of a different clock speed was necessary, and therefore the Phase-Locked Loop (PLL) from Altera IP library was inserted in the system. In the case of Intle Altera, the IP library provide a tool in which is possible to configure: the clock source, the type of PLL (Integer or Fractional), the number of output clock to generate and the relative clocking frequency.

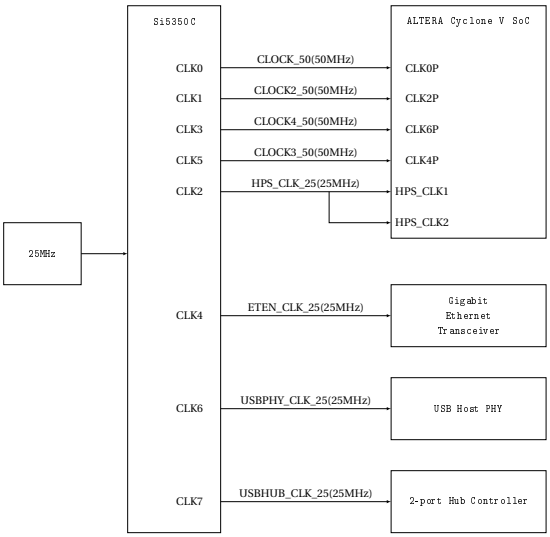


Figure 5.1: Block diagram of the clock distribution on DE1-SoC

5.3.3 Custom Component

All the previous elements must be interconnected with the interfaces that has to include the architecture. Another tool from Platform Designer helps to create all the interfaces, with the aim to ease the communication between the blocks. To ease the bus management, Intel Altera provides an Avalon Memory Mapped to AXI interface. The Avalon MM protocol is much more simpler, and it can be handled using an FSM. In the Figures 5.10a and 5.10b is shown how the Timing Sequence that the custom component must be compliant with. After the creation of the component, a new VHDL file is generated, with a precomputed entity that is conformed to the configuration.

At the end of the configuration and everything is setup, the `generate` command creates all the files that handles the intercommunication and a final top entity, that must be manually configured, with the interconnections toward the physical pins of the FPGA.

5.4 Architecture Structure

The architecture is divided into two parts:

- **Datapath**: it contains the memories to store and, as explained later, it is parallelized and pipelined in order to increase the performances and satisfy the constaints of the resources.

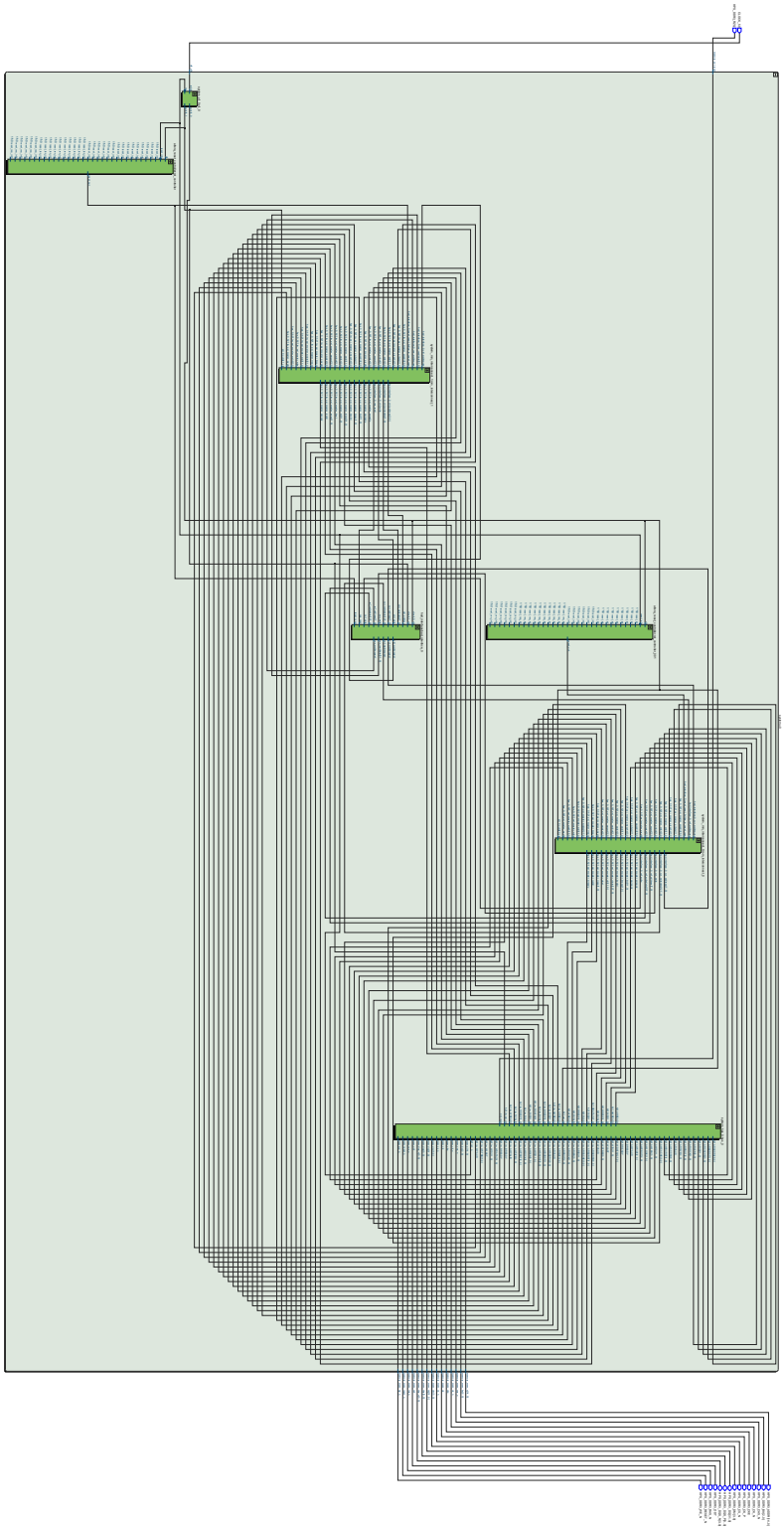


Figure 5.2: System Overview

- **Control Unit**: it contains the Finite State Machine that manages the pipeline inside the Datapath, and also controls the interface with the Light Weight Bridge.

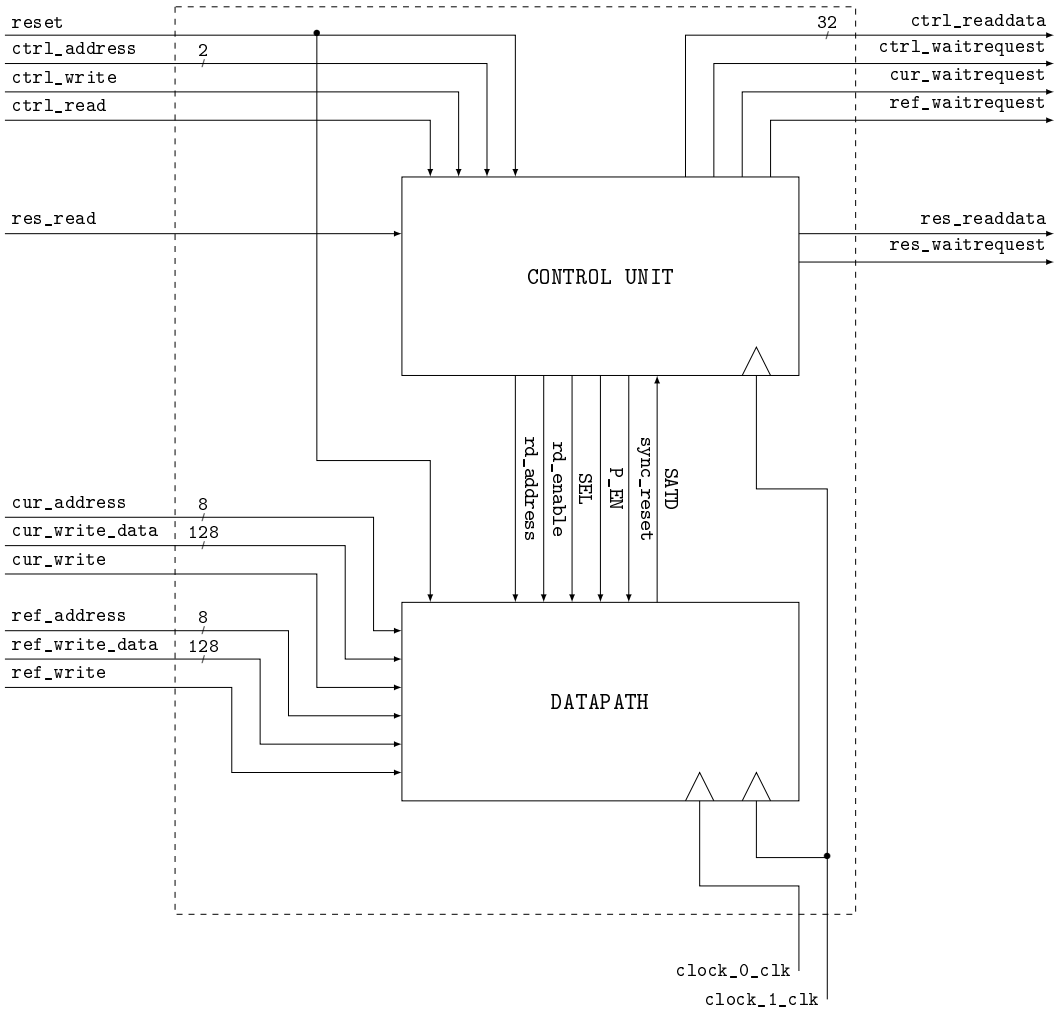


Figure 5.3: Top Level Overview

5.5 Datapath

Figure 5.4 shows the datapath of the architecture. There are 4 MUs that works in parallel and each one receives 64 pair of samples (512 from the current CU and 512 from the reference CU) for a total of 1024 bits. Except from the 8×8 case, there are 4 MUs results for each column o be summed.

In order to reduce the critical path and therefore raise the clock frequency several pipeline stages are inserted in the structure. An accumulator is used at the end of the adder tree in order to calculate the result in the cases of blocks composed by more than 4 8×8 base blocks

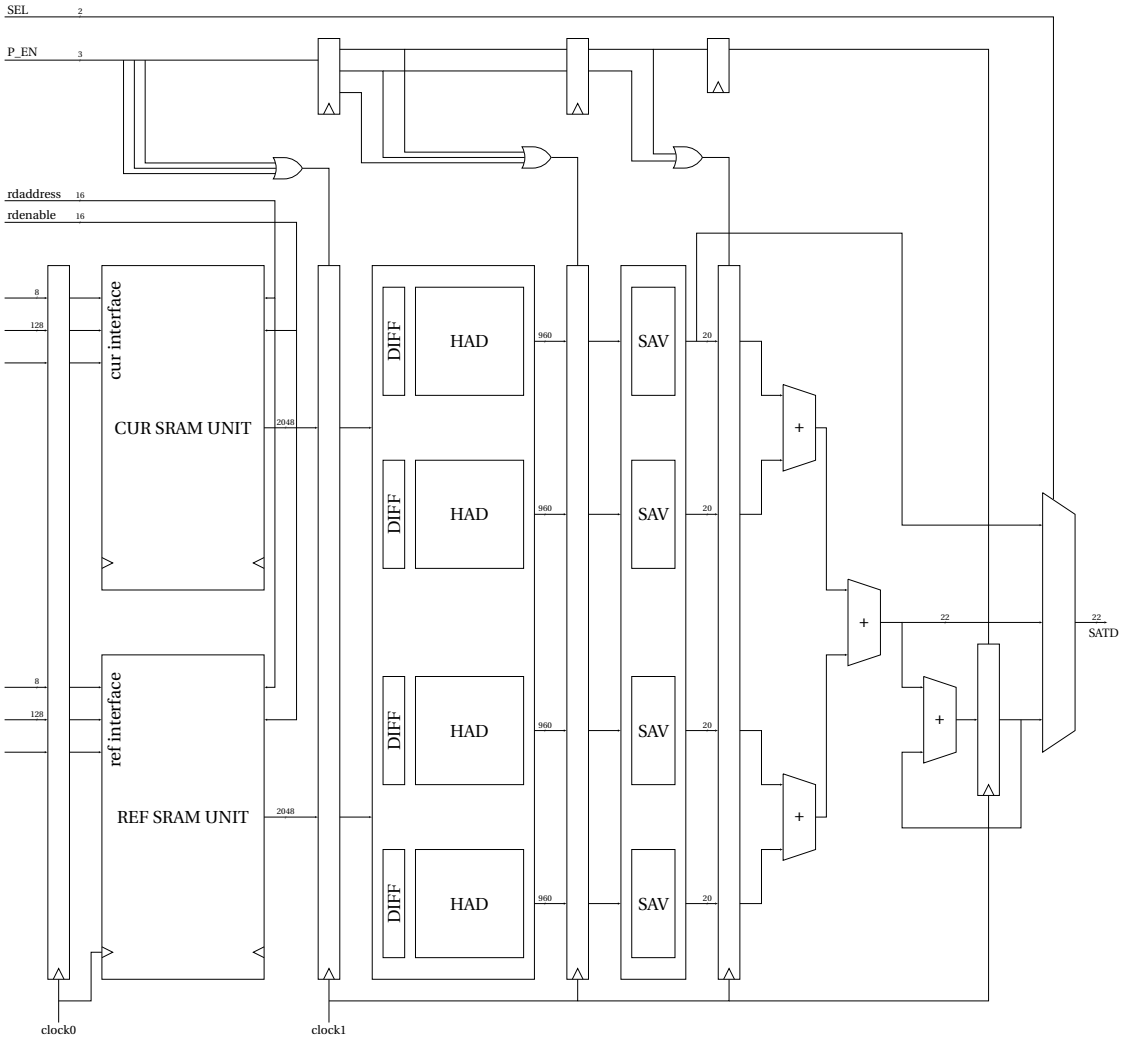


Figure 5.4: Datapath Overview

because those require more than one cycle to load the entire set of samples from the memory system. The datapath is pipelined, therefore at each cycle the datapath is fed with a new group of blocks. For what concerns the 8×8 case, the SATD in available at the output of the structure before reaching the adder tree end. The 16×16 case enters in the clock tree but it exits before reaching the accumulator. At the output is present a multiplexer that is used to select the adder tree exit. The selection signal is controlled by the Control Unit (see 5.6) which manages also the synchronous resets in order to clear the pipeline stages and the accumulator. The pipeline stages help to reduce the combinational path, and therefore an increase of the slack. The final result is then provided at the output of the multiplexer. At the beginning of the datapath the values are precharged on the first pipeline stage.

5.5.1 Main Unit

In the Figure 5.5 is shown the architecture of each MU in the datapath.

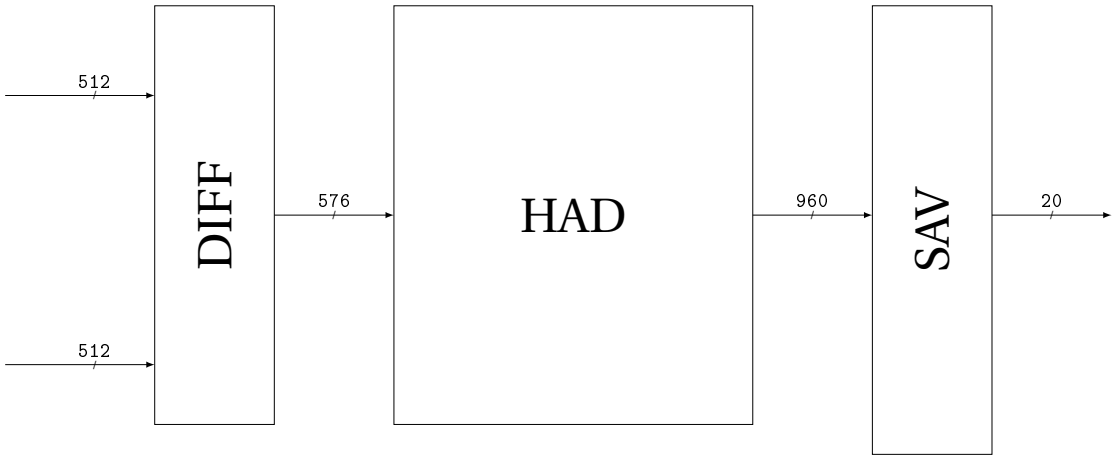


Figure 5.5: Main Unit

A Main Unit is a structure able to compute a complete 8×8 SATD. Thus, it receives 64 samples (512 bits) from the current PU and 64 samples from the reference PU, calculate the residual making the subtraction between the relatives samples, performs the matrix product with the Hadamard matrix (that is composed by 1 and -1, therefore it is composed by additions and subtractions), calculates the absolute value for each sample and all those values are added together with an adder tree. At the input of the datapath the samples have a parallelism of 8 bits, therefore each adder and register has the minimum size to have a correct result without overflow. At each stage of the cascade of operators the parallelism increase by one bit with the exception of the absolute value operator that does not modify the parallelism. After the initial subtractor there are 6 level of additions and subtractions so that at the input of the adder tree the parallelism is 15 bits. Then after the absolute value operator, in the adder tree the parallelism increase to 21 bits. In the further adders the parallelism raise up to the final accumulator register that has the maximum parallelism of 25 bits.

Figure 5.6: HAD Unit

5.5.2 Pipeline Stages and Enable signals

The very first pipeline stage is immediately before the datapath, in order to precharge the values to be ready for the very first cycle after the start of the calculation. In order to increase the clock frequency of the datapath the second pipeline stage is inserted in the Main Unit, between the Hadamard matrix product and the block that computes the absolute value and that contains the final adder tree. After this block a further pipeline stage is present and the next one is after the adder tree that computes the addition between the four results coming from the MUs. The final pipeline stage is after the accumulator. This means that in total there are 4 pipeline stage, but depending on the dimension of the block under test, the clock cycles to have the result ready are different, and so the enable signals and the MUX must be configured. Each pipeline register has an enable signal in order to be controlled by the Control Unit (see 5.6). From this Control Unit 3 signals come and reaches the first pipeline stage. These signals are propagated through the further stages depending on the configuration of the control register. This three signals can have the following configurations:

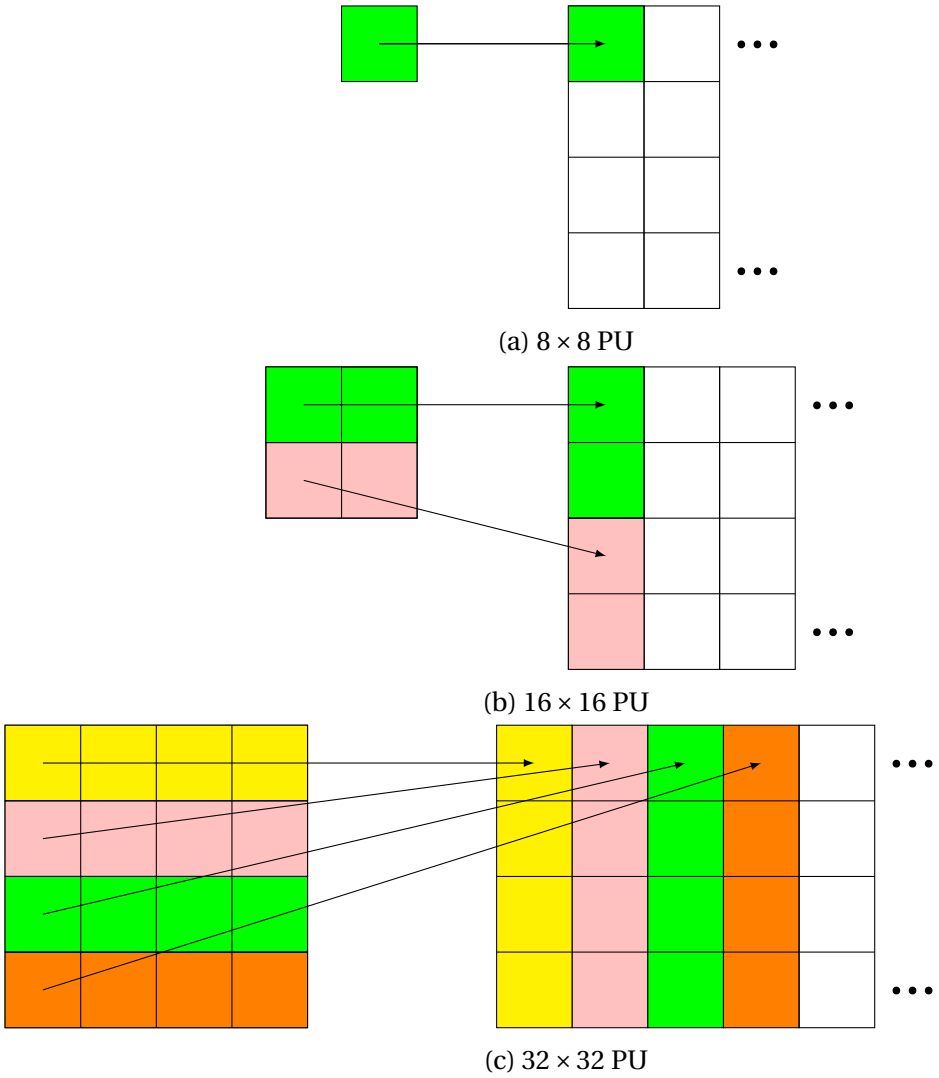
- "001", in case of 8×8 TU;
- "010", in case of 16×16 or 32×8 TUs;
- "100", for the other cases.

After the second stage, in which the signals are delayed together with the pipeline stage, at the input of the Enable of the registers arrives the `OR` of those, while the two MSB are delayed together with the pipe, and so on. In that way the registers are activated independently on the configuration, and the the other stages are enabled only if the we are in the corresponding case. Only the MSB arrives at the Enable of the the last pipeline stage. Thanks to this configuration, all the registers sample only when necessary and the commutations in the combinational logic cone downstream of these registers are blocked, so that the dynamic power is highly reduced.

5.5.3 Memory System

As said in 5.2 each MU is univocally associated to one 8×8 of a 64×64 PU, in a sense that receives it always receives data from the same input which contains a 8×8 reference block and a 8×8 current block. The memory at this point must be structured with two memory bank for each MU and both the memories must contain the maximum amount of data that is 64×64 bytes. The maximum parallelism of the HPS to FPGA bridge is 128 bit, meaning that 128 bit will be the width of the data input for the memories. Therefore each memory bank can be made up by 16 locations with 128 bits of data each, and in this way each MU is related to 4 banks. Either in the memory that stores the reference block and in the one that stores the current block are present 16 of this banks, and in this way it is possible to store a complete 64×64 block. In the Figure 5.7 is shown this relation between the RAM banks and the MUs.

The Control Unit generates the Chip Select signals in order to activate only the banks of RAM that are needed. These enable signals activate the memory bank for the reading process. This peculiar organization was performed in order to have the memory banks working in parallel: in fact they all receive the same address from the Control Unit that select one of the 16



(a) 8 × 8 PU

(b) 16 × 16 PU

(c) 32 × 32 PU

Figure 5.7: Relation Between Memory and MUs

location and, in this way, the memories are able to produce at the output up to 4096 bits for the reference block and 4096 bits of the current in only one clock cycle. Beside the fact that there is a reduction of the latency, his type of parallelism has the advantage of reducing logic of the decoder at the input of the memories, and therefore the area is minimized. Each memory stores the reference and the current PU, and therefore they can be seen as caches. In fact after that the reference block is sent, only the current block must be sent each time to calculate the SATD.

5.5.4 Memory Implementation

Inside the Cyclone V SoC includes several embedded emory blocks, with a flexible design. Depending on the time of application, the dimensions of the memory block array change.

The Cyclone V devices contain two types of memory blocks:

- 10 Kb M10K blocks-blocks of dedicated memory resources. This praticual type is is deal for larger memory array block.

- 640 bit memory logic array blocks (MLABs)-enhanced memory blocks that are configured from dual-purpose logic array blocks (LABs).

Intel Altera provides an IP cores to implement the memory modes: the ALTSYNCRAM. The selection depends on the target device, memory modes, and features of the RAM and ROM. Using the IP tool it is possible to generate a custom IP RAM with the preferred configuration depending on the capabilities of the chip. As said before, it is required a dual port SRAM memory that has 128 bits of data width at the input and 4096 bits at the output, with the possibility to enabled the desired banks. Unfortunately using this tool it is not possible to create such a memory, therefore a single 16 locations with 128 bits bank is created and it is used to generate the desired SRAM structure, that is shown in the Figure 5.11b. The port 1 is dedicated only for the writing operation while the port 2 only for the reading. As said in 5.5.3, the reference block is written only once at the beginning of the procedure and then it is only written, while the current block is written at each SATD operation. The port 1, as shown in Figure 5.11b, is handled by the HPS to FPGA bridge and it used to write the samples to be used in the SATD operation. The port 2 instead is handled by the Control Unit of the architecture. Due to fact that the datapath works at a different clock frequency with the respect to the one used to write the memory, for this type of memories need also to be dual clock memories. From the datasheet, the dual clock SRAM memories can reach up to 150 MHz for both reading and writing routines. Unfortunately this clock speed can be reached only if the memory is used as the standard structure that the IP tool generates. In this case the memory banks are interfaced with a decoder, that introduces a combinatorial logic between write and data signals coming from the bridge and the input of the banks. This additional logic cause a corruption of the data in the memory because the setup and hold times are not satisfied. In order to overcome this problem, a stage of pipeline was introduced between the signals coming from the bridge and the input to the decoder. In this way was possible to reach the above mentioned clock frequency. The memory system behaves like a FIFO and it is the HPS that is in charge to handle the memory hierarchy made up by the off-chip DDR3 SDRAM and the internal SRAM system. When there is a new SATD to be calculated, it must write into the current SRAM memory the new current PU and, if also the reference PU change, it update it. The combinations of PUs in which the architecture is used are 18 and they are listed in Table 5.1.

However, some of those cases require the same number of cycles in order to be calculated (for example 32×8 and 8×32 cases can be grouped together with 16×16) as they need the same procedure to be calculated), therefore the effective number of cases to be analyzed and implemented on the peripheral in order to cover all the possibilities was 8.

8×8	32×8	32×24
8×16	32×16	32×32
8×32	16×64	64×32
16×8	24×32	64×48
16×16	48×64	32×64
16×32	64×16	64×64

Table 5.1: List of all the block dimension compatible with the Architecture

5.6 Control Unit

The Control Unit handles the datapath command signals, part of the memory system, the configuration of the control register and the requests from the user for the result, by managing the opportune handshake signals. In the Figure 5.8 is shown the block diagram of the Control Unit.

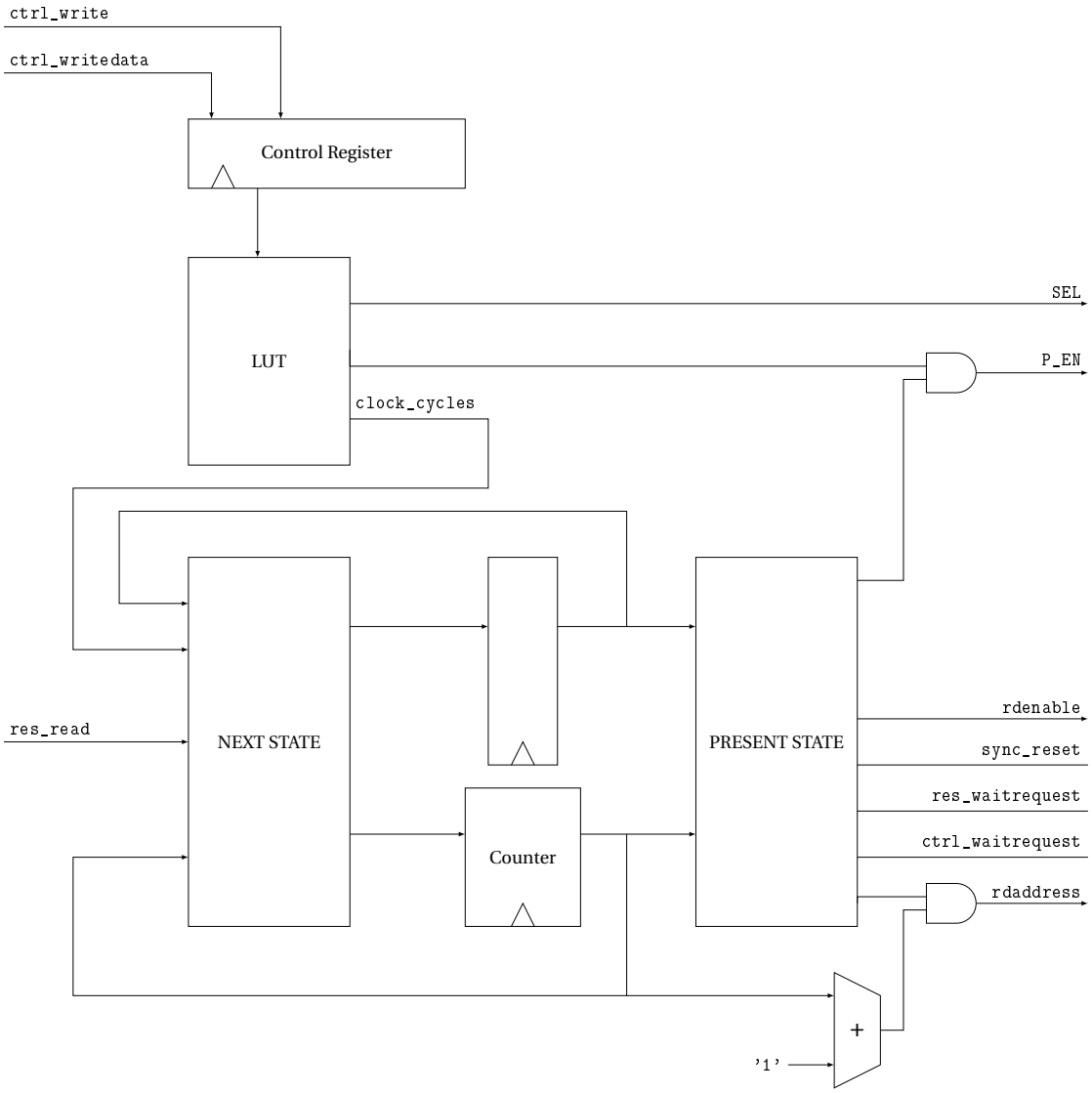


Figure 5.8: Control Unit Overview

It has two main interfaces toward the bus and it generates the enables and the addresses for the memory units, the `SEL` signal for the MUX at the output, a synchronous reset and it receives from the datapath the result, in order to handle the communication with the result interface. The SATD architecture must know the dimension of the PUs in order to understand

how many clock cycles are necessary to calculate the result and to properly select the MUX. This parameters for every SATD type are hardwired thanks to a Look-Up Table (LUT). This LUT receives the input from a Control Register, that is programmed using the `ctrl` interface, and depending on its value it returns the number of clock cycles that are needed, the selector for the multiplexer inside the datapath and the enable signals to manage the pipeline registers. the Control Unit manages also several handshake signals, that are part of the Avalon MM protocol. By the fact that the architecture works as a slave, in order to communicate to the master either that the slave is not ready to be written by the master or that the data produced by slave is not ready to be read by the master. The generation of all the signals is controlled by a Finite State Machine (FSM) that it is kept as simple as possible, in order to not complicate the calculation and to reduce as much as possible the latency for the generation of the result.

5.6.1 Control Finite State Machine

In the Figure 5.9 is shown the State Graph of the FSM.

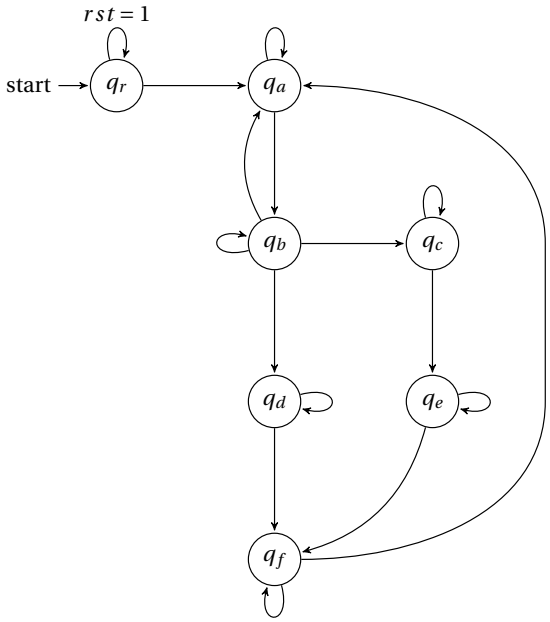


Figure 5.9: FSM State Graph

For the first reset state, the machine switches to the q_a that produces the `sync_reset` for the registers in the datapath, and it waits for a request from the `res` interface for the result. When it comes, the FSM switches to q_b which start producing the enable signals for the pipeline and the addresses in order to read from the memories (it sends essentially the value from a counter that counts the number of cycles). At this point, depending on the type of PU, it passes to either q_d or q_c and based on the values of *clock cycles* and provided by the LUT the q_c q_d and q_e wait until the result is ready, passing to the last state q_f. The `ctrl_waitrequest` is the handshake signal related to the interface that programs the Control Register and it is

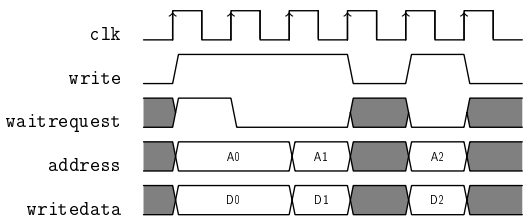
0 only when the architecture is not working. In this way there is not the risk that the Control Register changes its value during a computation. The `res_waitrequest` is the handshake signal that related to the `res` interface. When the master requests to read the result, it waits until the it is 0. That means that this value it is always asserted, and only when the FSM reaches the state in which the data is ready, it set to 0. When the result is produced, the FSM starts again from the first state, waiting for the next calculation.

5.7 VHDL Implementation

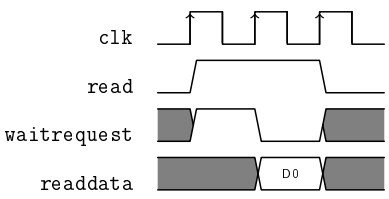
The hardware accelerator was realized in VHDL. All the main dimension and values used into the architecture were specified in a package of constants. The hierarchy follows the previous explanation: after the creation of the entity using Platform Designer, it was created the MU block, the datapath, the Control Unit and then the top entity. The adders inside the datapath and the subtractor and the block that performs the absolute value was described in behavioral way letting the synthesis tool to use the IP components, that are well optimized for the FPGA. Later on the adderes were substituted with the approximated version, described in Chapter 10. The SRAM is an IP of the Altera library used for the synthesis: if it is described in VHDL, the synthesizer automatically should recognize it as SRAM memory and try to force the usage of the internal memory blocks. If it is not possible, the synthesizer will try to use the internal logic to create the memory cells, but this leads to a long time for the compilation, and a not so optimized result.

5.7.1 Avalon Memort Mapped Slave Protocol

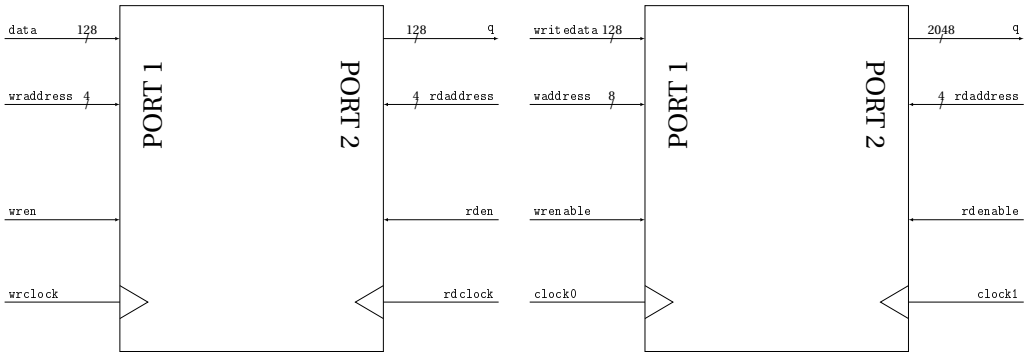
There are two different handshaking protocols for read and write operations. Since the peripheral acts as slave, a Master Write corresponds to a Slave Read and a Master Read corresponds to a Slave Write. The Avalon-MM interface is synchronous. Each Avalon-MM interface is synchronized to an associated clock interface. Signals may be combinational if they are driven from the outputs of registers that are synchronous to the clock signal. This specification does not dictate how or when signals transition between clock edges. In Figures 5.10a and 5.10b it is represented the waveform of the protocol in case of write and read.



(a) Example of Write Waveforms



(b) Example of Read Waveforms



(a) Logic interface of a single memory bank as it is provided from the library IP

(b) Structure of the memory system

5.7.2 Internal RAM Memory

The Intel Altera library offers several IP cores to implement memory modes. Using Quartus it is possible to generate the desired memory with all the available features. For this project were used Dual Clock RAM memories in order to manage the communication from the outside with a different clock speed from the one used internally. Therefore, it was possible to operate on only one part of the architecture at a time and once the clock frequency of the write port was fixed, it was not more changed.

In Figures 5.11a is shown the single memory bank generated from the IP library, composed by 16 location of 128 bit.

In Figure 5.11b is represented, instead, the memory system composed by 16 memory banks. `waddress` is the write address and it is composed by 8 bit: the 4 LSBs are used fore address the internal locations of each bank, while the remaining 4 MSB are used to select one of the 16 memory banks. Therefore at the input of Port1 it is present a decoding logic that has to be considered for the timing constraints to avoid setup and hold violations and, thus, data corruption.

Chapter 6

Architecture Validation and Debug

The validation of the architecture had followed two main stages:

- A testbench
- Software C Program

In addition a Matlab scripts was prepared in order to generate a database that was used for validate the architecture during the simulation.

The first test checks the correct behavior of the architecture acting on the interfaces. The simulations were carried out with Mentor Graphics Modelsim. The second test instead strains the accelerator and emulates a true SATD minimum search (as it is inside the HEVC encoder) and therefore it tests SATDs of only one type in burst mode. In addition to te second test it was performed a timing comparison of the SATD computation between the software function and the FPGA peripheral using the internal timer.

6.1 VHDL Testbench

The Platform Designer generated component is a black box with all the interfaces through the bridges in its entity. The Avalon interfaces follow simple communication protocols such as a read and a write signal, write and read data bus and several handshake signals, as shown in Fig 5.10a and 5.10b.

This component contains the designed architecture that must be tested and in order to do so it is necessary to provide at the input the data to be stored in the RAM memories and the trigger the signals that start the computation, and finally check the result. The testbench was made therefore following the Avalon protocols and, in order to provide consistent data, it was created a MatLab script that was able to generate a subset of random numbers in a uniform distribution within the range of 16 to 235 according the YUV format speciïňĄcations. The same data set of data was provided either to the Testbench code and to a custom C code that was written taking the functions for the SATD calculation from the HM reference software. Comparing the two results was the initial step in order to verify the correct functioning of the structure. The Figure 6.1 shows the structure of the VHDL Testbench. It is composed by three main parts:

- Writer
- Control
- Clock Generation

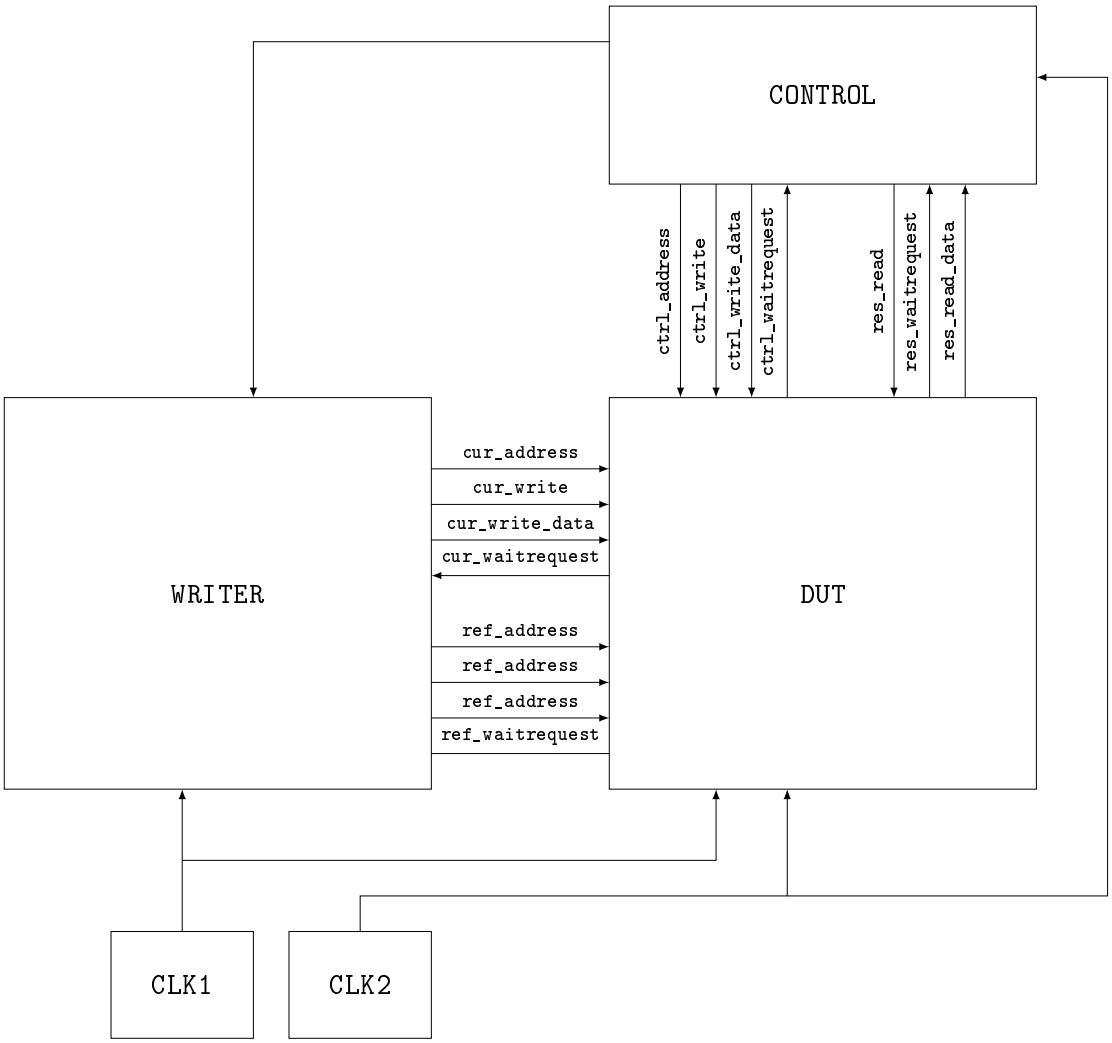


Figure 6.1: Testbench block scheme

6.1.1 Writer

The writer process works in the regime of the faster clock (150 MHz) and so it handles the `CUR` and `REF` interfaces. When it receives from control process the order to write a PU, it starts writing on the memory units. Depending on the dimension of the PU specified it writes either 64, 256, 1024 or 4096 samples, for each memory, taken from two files generated by MatLab.

6.1.2 Control

The control instead works in the regime of the slower clock (66.6 MHz). First of all it programs the Control register using the `CTRL` interface, in order to select the size of the PU. After that it generates the signals in order to instruct the writer. At the end of the write, it starts the request to receive the result by asserting the `res_read` signal. When the result is ready, the generated output is then written on a file, and later compared with the output produced by a software program that uses the same input data. This software program was the starting point for the next test.

6.2 Software Test Program

When the VHDL simulation started to produce correct results, it was the moment to verify the behavior with an on-board test. In order to do so it was necessary first of all to compile the project and to upload the `.rbf` file that is the configuration file for the FPGA. After that in the previous C code the sending of the data to and from the FPGA was integrated. In Chapter it will be described how to access to the FPGA from user level. All the cost function used by the HM reference software are contained in the class `TComRdCost`, which contains, in addition to the Sum of the Absolute Transformed Differences (SATD), the Sum of the Absolute Differences (SAD) and the Sum of the Squared Errors (SSE) metrics. Each time that a new prediction must start, the function `setDistParam` is called. This function sets the dimension of the PU, the pointers to the reference and current block and other parameters like the bit depth ans so on in the class `DistParam`. In addition to this, it sets the type of function that the function `DistFunc` points. The function that computes the SATD is `xGetHADs`. As explained in 5.1, depending on the dimension of the block this function calculates the SATD by evaluating blocks smaller than 8×8 samples with 4×4 Hadamard Transform and blocks of 8×8 samples or larger with the 8×8 Hadamard Transform. Knowing that the dimension of the block that the peripheral can analyze is in the range between 8×8 and 64×64, the code was modified in the first `if` sentences, in which was inserted the statement for the usage of the peripheral. Therefore in the test program was composed by four function:

- `setDistParamSW` and `xGetHADsSW` that respectively set the variables of the parameters and calculate the SATD as in the HM reference software;
- `setDistParamFPGA`, which first of all sets the variables of the parameters and at the same time configure the peripheral by sending the corresponding value for the Control Register. Then the content of the reference block is sent to the FPGA because, as said before, it is the same for all the comparisons.
- `xGetHADsFPGA` in which is present the effective calculation preceded by the sending of the current block to the FPGA.

Using this functions were created many test program with the aim of verify the correct functioning of the structure. In the first test program was just inserted a simple call of those function giving the same inputs as reference and current block. The result was therefore provided to the standard output. In this way it was possible to check the functionality of the

structure in the operative mode and comparing the result with the corrected one produced by the software functions. Subsequent versions of the program were improved in order to stress the peripheral. In fact it was created a loop that emulates a search within a frame, as it occurs in Intra Prediction, using as data input the samples generated using MatLab. In this case the results of the FPGA calculated values and the Software ones were inserted in an array. At the end of the computation, a comparison of the two arrays was computed, and flag produced on the standard output was signaling if the overall computation was correct or not. In the case of failed computation, the arrays were debugged, analyzing in which cases the FPGA peripheral was producing an error. For example, it was at this point that the problems related to data corruption on the memories mentioned 5.5.4 was discovered. In fact running the test program with the clock of the memories set at 150 MHz was producing a wrong result, while using a reduced clock frequency, it was perfectly working. The reason was that the memory instantiation was different to the one created by the tool that generates the IP RTL of the RAM blocks. In fact in order to obtain the desired RAM structure it was necessary to create a memory composed by 16 location of 128 bits, and then instantiate it in a component that is composed by 16 of them, because is the datapath there are 4 MUs and each of them is connected to 4 memory banks in order to receive a complete 8×8 block. At the input of this component is present a decoder, and this additional logic causes a corruption of the data in the memory because the setup and hold times are not satisfied. At this point was included a pipeline stage at the input of the interfaces of the memories, and after a second test, this bug was solved.

6.3 Timing Measurement

A further modification to the software code was implemented in order to test the performances in terms of time of the peripheral, as one of the main focus of this work was the creation of an hardware accelerator. The C time functions are a group of functions in the standard library of the C programming language implementing date and time manipulation operations. They provide support for time acquisition, conversion between date formats, and formatted output to strings. Those functions are defined in the *time.h* header file. By the usage of the function `clock_gettime()` it was possible to make misurations for both the SW and FPGA function with the aim of making a timing comparison. Two type of test were performed:

- The first one is based on a single block, and each time the data is sent to the FPGA. In addition the average time is more and more reduced when higher dimension blocks are taken into account.
- The second one recreates an emulation of a search: software with and within the peripheral is tested and, in the case of peripheral usage, the current block is sent only once to the memory, thus further reducing the time Encoder.

It is possible to notice that this solution in this case is far moe better than software one.

Block Dimensions	Software Time (ns)	FPGA Peripheral Time (ns)	ΔT (%)
8 × 8	19045	6206	-67.4
16 × 8	39654	11639	-70.0
16 × 16	79145	21849	-72.39
32 × 8	79523	22071	-72.24
32 × 16	158018	42624	-73.0
32 × 24	228502	61072	-73.2
32 × 32	316196	83969	-73.4
64 × 16	316400	83936	-73.5
64 × 32	632481	167418	-73.5
64 × 48	914330	241060	-73.6
64 × 64	1267051	331850	-73.8

Table 6.1: Experimental Time Measurements for each single block dimensions

Block Dimensions	Software Time (ns)	FPGA Peripheral Time (ns)	ΔT (%)
8 × 8	670097	109534	-83.6
16 × 8	1338515	200372	-85.0
16 × 16	2668982	379678	-85.7
32 × 8	2679274	380912	-85.7
32 × 16	5340850	740012	-86.1
32 × 24	8368034	1143228	-86.3
32 × 32	10676977	1462590	-86.3
64 × 16	10649720	1460058	-86.3
64 × 32	21346267	2897915	-86.4
64 × 48	32018153	4367719	-86.3
64 × 64	42726355	5766809	-86.5

Table 6.2: Experimental Time Measurements for each block dimensions emulating a search

Chapter 7

Synthesis

7.1 Introduction

The base design of the hardware accelerator was synthesized before the insertion of the optimization that will be presented in the Chapter 9 to understand the improvements and eventually drawbacks generated by them. The project was synthesized by using the Quartus Prime software. The overall synthesis is composed by five steps:

- Analysis and Synthesis
- Fitter (Place and Route)
- Assembler
- TimeQuest Timing Analysis
- Power Analysis.

In the synthesis stage of the compilation flow, the Quartus II software performs logic synthesis to optimize design logic and performs technology mapping to implement the design logic in device resources. After the synthesis stage a single project database that integrates all your design files in a project is generated. Quartus II Analysis and Synthesis, together with the Quartus II Fitter, incrementally compiles only the parts of the design that change between compilations, reducing concretely the compilation time.

7.2 Analysis and Synthesis

In Appendix A.1 is reported the overall resources usage. The number of total Logic utilization (ALMs) is 14967 (47% of the total ALMs on the device). In a previous attempt it was performed an $\times 8$ parallelization of the MUs that would have reduce the cycles need to calculate a TU composed by a number of 8×8 blocks, but this type of implementation needed 34490 ALMs and, therefore, it was rejected during synthesis, because of the limited number of resources that is 32070 for this particular chip. It is also reported the number of used PLL, that in this

case is 2: one is generating the clock for the memories and the other one the clock for the internal datapath and control unit.

7.3 TimeQuest Timing Analyzer

The Quartus II TimeQuest timing analyzer allow to analyze the timing characteristics of your design. In addition, a SDC contraint file can be provided to the tool in order to make a complete time analysis.

In this case it was necessary to specify two false path in the `.sdc` file. In fact the architecture is working with two clock domains, and this influences the Static Time Analysis, resulting in a wrong result for the Recovery Time.

The TimeQuest Analyzer performs a multi-corner analysis depending on the actual timing characteristics of the chips that are subject to PVT (Process, Voltage, Temperature) variations:

- Process: after the manufacturing process, not all the chips are equal. For a given FPGA speed-grade, some chips will be faster, some slower.
- Voltage: higher Vcc increase the speed performances of the chip, while a lower Vcc degrades it. In this analysis is considered the minimum supported Vcc operating conditions, which is 1110 mV.
- Temperature: also a lower T increase the speed performances, vice versa for an high temperature. In this analysis were considered 0 °C and 85 °C as test conditions for the temperature.

Therefore four type of report were generated:

- Corner 1: Slow model - 85 °C - 1110 mV: the worst case scenario.
- Corner 2: Slow model - 0 °C - 1110 mV.
- Corner 3: Fast model - 85 °C - 1110 mV.
- Corner 4: Fast model - 0 °C - 1110 mV: the best case scenario.

In order to fit all the constraints, it was mandatory to focus on the worst case scenario, therefore only the Corner 1 was analyzed. In Appendix A.2 is reported the Timing Report Summary. It can be noticed that the slack related to `clock0` (for the memories) is 0.024 ns, and therefore there were no margins for improvements. On the contrary the setup related to to `clock0` (for the CU and internal datapath) is 2.060 ns, meaning that the clock frequency could be slightly increased. However this was limited by the PLL, because generating a frequency that is not a multiple of the reference (50 MHz) led to a generation of a clock frequency for the voltage-controlled oscillator (VCO) greater than the allowed threshold.

7.4 Power Analysis

The Power Consumption Evaluation is very complex to execute in this type of environment. In Quartus II is present the PowerPlay tool, which based on the architecture and depending on the usage of the HPS, realize an estimation of the power consumed. However, this type of evaluation is not precise, and to be more accurate it need the `.vcd` file with the estimation of the inputs, generated from the simulation of the architecture with a testbench, but in this case the generation of the testbench was not trivial and practically was not the main purpose for this work, knowing that this is a work-flow for the study of prototypes for future ASICs realization. Therefore, due to the low confidence of the power analysis, the results for the power estimation were omitted.

For a real power evaluation the whole board could be monitored by plugging a power adapter into a current sensor. The current sensor output would then be applied to the on-board ADC and read by the FPGA. For example, it might be used the Adafruit 1164, equipped with a TI INA169 current sensor. The sensor board has a bandwidth of 100 kHz, gain of 1 V out per 1 A input, and with an output noise of about 7×10^{-5} V. Because the current is measured for the whole board, separating out the FPGA contribution will require careful calibration. However, this could be a framework for a future research on this topic.

Chapter 8

Integration in HM Reference Software

8.1 Introduction

This chapter will describe the steps for the development of a program that can run under Linux on the DE1-SoC. This passage was essential in order to integrate the usage of the hardware accelerator inside the HM reference software. The approach used in this case is called *cross compilation*, in which the program is written and compiled on a host computer, and then the resulting executable is transferred onto the Linux file system that is inside the microSD card.

8.2 Access to the FPGA

Programs running on the ARM processor of the Cyclone V SoC device under Linux can access hardware devices that are implemented in the FPGA through either the HPS-to-FPGA or the Lightweight HPS-to-FPGA bridge. These bridges are mapped to regions in the ARM memory space. When an FPGA-side component (such as an IP core) is connected to one of these bridges, the component's memory-mapped registers are available for reading and writing by the ARM processor within the bridge's memory region.

When programs are being run under Linux it is not as straightforward to access memory-mapped I/O devices because Linux uses a virtual-memory system, and therefore application programs do not have direct access to the processor's physical address space.

To access physical memory addresses from a program running under Linux, you have to call the Linux kernel function `mmap` and access the system memory device file */dev/mem*. The `mmap` function, which stands for memory map, maps a file into virtual memory. You could, as an example, use `mmap` to map a text file into memory and access the characters in the text file by reading the virtual memory address span to which the file has been mapped. The system memory device file, */dev/mem*, is a file that represents the physical memory of the computer system. An access into this file at some offset is equivalent to accessing physical memory at the offset address. By using `mmap` to map the */dev/mem* file into virtual memory, we can map physical addresses to virtual addresses, allowing programs to access physical addresses. In the following section, we will examine a sample Linux program that uses `mmap` and */dev/mem* to access the Lightweight HPS-to-FPGA (*lwhps2fpga*) bridge's memory span and communicate

with an IP core on the FPGA. More information are available in [15].

8.3 HM Reference Program

As mentioned in 6.2 the two functions at stake are `setDistParam` and `xGetHADs`. While the first one sets the dimension of the PU, the pointers to the reference and current block and other parameters like the bit depth ans so on in the class `DistParam`, the second one, reported in Appendix B.2, effectively computes the SATD.

The Table 8.1 lists all the function and the relatives source files that are used during the Intra Prediction. In particular the effective call of the `xGetHADs` is performed in the function `estIntraPredLumaQT`. In fact after the call of the function `predIntraLumaAng` in which it is present the analysis of the 35 modes of the HEVC standard, the SATD is calculated in order to evaluate the rate-distortion cost among the selected modes. In Figure 8.1 is shown the flow chart of the Intra Prediction inside HM.

Library	TLibCommon	TLibEncoder
Source file	TComPattern.cpp	TComPrediction.cpp
	TComRDcost.cpp	TEncSearch.cpp
Functions	xCompressCu	estIntraPredLumaQT
	estIntraPredChromaQT	xRecurIntraCodingQT
	xRecurIntraChromaCodingQT	initAdiPattern
	predIntraLumaAng	xGetHAD
	xUpdateCandListx	RecurIntraCodingQT
	fillReferenceSamples	xPredIntraPlanar
	xPredIntraAng	predIntraGetPredValDC

Table 8.1: Main functions in Intra Prediction

For what concerns the FME, the SATD calculation for the fine-grain search is embedded in the function `xPatternRefinement`, also part of the TEncSearch class. To integrate the usage of the FPGA peripheral into HEVC reference software HM 16.15 it was necessary to create a new class that did not substitute the original one, but instead starts the computation using the peripheral when is necessary. In fact the peripheral can work only with a bit depth of 8 (while the HM is structured in order to work with higher bit depth). Moreover it can manage only the blocks with dimension from 8×8 to 64×64.

This class was called `FPGAParam` and includes all the configurations and functions for the usage of the peripheral. `FPGAParam` class contains many variables that must be configured the start and they have to be destroyed at the end of the execution: pointers to the FPGA bridges in fact are mapped in the constructor and those mappings are removed in the destructor. In Figure 4.2 is reported in the column *Base*, the base address on which the peripheral in memory mapped on the AXI bus. This will be the offset that must be added to the address mapped on the virtual memory in order to have the reference to send and receive the data from the user level. As explained in 5.5.3, a frame is composed by an array of pixel. When the Intra-Prediction is working, to each PU that has to be analyzed are assigned several parameters, in

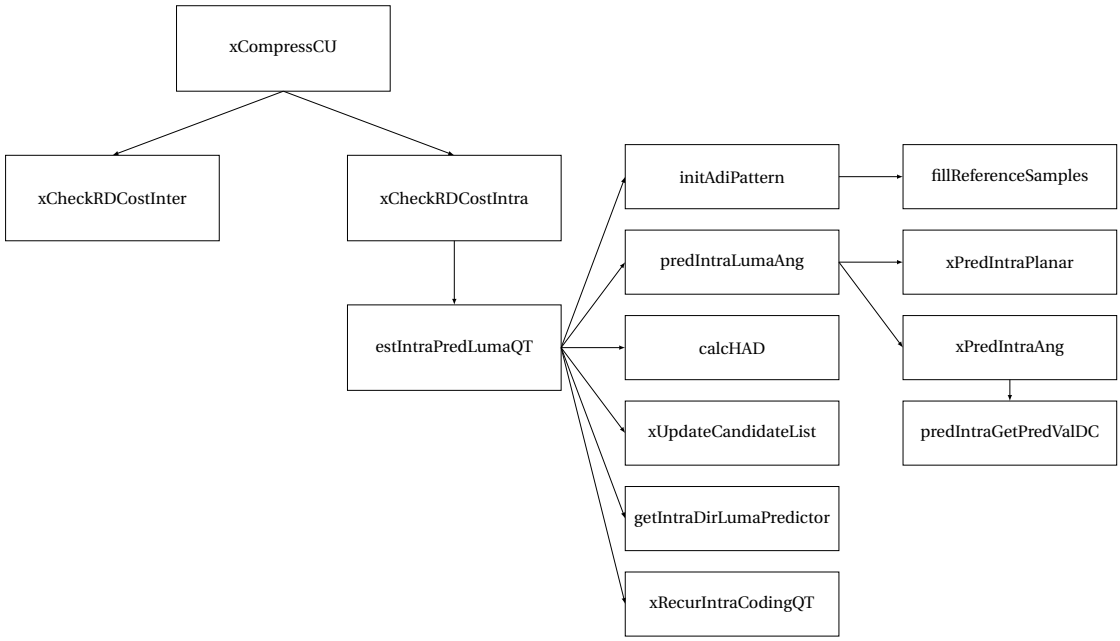


Figure 8.1: Flow chart showing important functions in Intra Prediction

order to collocate it within the memory. Those parameters are:

- `piOrg` and `piCur`, that are the pointers to the first pixel in the above-left square of the reference and current PU,
- `iRows` and `iCols`, that are the dimensions of the reference and current PU,
- `iStrideCur` and `iStrideOrg`, that are the stride values (the distance between a pixel and the one on the next rows and on the same column).

In order to use the peripheral, was necessary to reorder each block in a buffer, in order to have each 8×8 blocks contiguous in the SRAM memory. Depending on the dimension of the block, the reordering changes because the positions of the 8×8 blocks in an higher block change. Therefore each sending of the data is preceded by the reorganization of the block in a buffer. This block in then sent to the FPGA using `memcpy()` function and the result is requested.

In summary the actions performed by the function `setDistParam` are:

- Set the variables relates to the dimension of the block, stride value, bit depth and the type of function (either SAD, SSE, SATD with software function and SATD using the peripheral),
- Configuration of the Control Register, unless it is not already configured with the actual value.
- Reorder of the reference block in a buffer.

- Move data from the buffer to the Reference Block.

The actions performed by the function `xGetHADs` are:

- Set the variables relates to the dimension of the block, bit depth and and also the type of function,
- Depending on the previously set dimension of the block it activates the peripheral.
- Reorder of the current block in a buffer.
- Move data from the buffer to the Reference Block.

A further version was impemented to validate the functioning of the driver. To do this the driver is integrated directly in the default version of `xGetHAD` function, a comparison between the result provided by the software version and the one coming from the peripheral is made. For each wrong result, the program annotes it into a file, specifing the block dimension, the correct and the wrong resut.

8.4 Timing Measurements

To evaluate the efficiency of the hardware acceleration, a comparison in term of time was performed. The proposed implementation was tested in the HEVC test model HM 16.17. HM was configured in *All-Intra* mode, and run for quantization parameters (QP) 22, 27, 32 and 37 for the sequences of type D (416×240). For each configuration, the test was performed three times and the resulting time was the average. Other two timing measurements were performed for both *Random Access* and *Low Delay*, but in those cases only one encoding was executed due to the extremely long duration of the test. This results for *All-Intra* configuration are shown in the Table 8.2, while those for *Random Access* and *Low Delay*. are represented in Tables 8.3a and 8.3b. It can be noticed that the integration of the peripheral driver in the HM software did not produced the expected results as it was for the previous test. There are many reasons for this, and the most probable is that the usage of this peripheral inside the software led to an congestion of the memory instruction, creating a slowdown of the system.

Video Sequences	QP	Default Encoding Time (s)	FPGA Encoding Time	ΔT(%)
RaceHorses	22	3041	2895	-4.8
	27	2694	2541	-6.5
	32	2345	2223	-5.2
	37	2062	1933	-6.3
BasketballPass	22	4687	4762	+1.6
	27	4090	4190	+2.4
	32	3625	3683	+1.6
	37	3274	3330	+1.7
BQSquare	22	6826	6920	+1.3
	27	5968	6007	+0.6
	32	5194	5237	+0.8
	37	4600	4653	+1.1
BlowingBubbles	22	5922	5996	+1.2
	27	5165	5264	+1.9
	32	4450	4510	+1.3
	37	3799	3857	+1.5
Average		4233	4250	+0.4

Table 8.2: Time results for recommended video sequences in All Intra

Video Sequences	QP	ΔT(%)
RaceHorses	22	+1.1
	27	+0.6
	32	+0.4
	37	+1.3
BasketballPass	22	+4.1
	27	+3.7
	32	+2.9
	37	+2.7
BQSquare	22	+5.2
	27	+1.6
	32	1.0
	37	+0.7
BlowingBubbles	22	+1.0
	27	+1.0
	32	0.0
	37	+0.3
Average		+1.7

(a) Time results for recommended video sequences in Random Access

Video Sequences	QP	ΔT(%)
RaceHorses	22	+0.4
	27	+0.1
	32	+0.1
	37	+0.3
BasketballPass	22	+3.8
	27	+2.7
	32	+1.6
	37	+1.9
BQSquare	22	+3.3
	27	+1.3
	32	0.0
	37	+0.6
BlowingBubbles	22	+1.5
	27	-3.0
	32	-4.1
	37	-1.3
Average		+0.5

(b) Time results for recommended video sequences in Low Delay

Chapter 9

Optimizations

One of the main target of this work was to explore the potential of this FPGA. At the end of the final implementation of the architecture and the operative functioning was tested, it was the time to optimize the work using the additional feature that the system provides. These optimization are Clock Gating and the usage of the Direct Memory Access. The first one is the most used technique to reduce the dynamic power consumption while the second one is the fastest methods in order to reduce the latency when moving data from a part of memory to another.

9.1 Clock Gating

The architecture is strongly parallelized since it can employ up to 4 MUs that can compute four 8×8 SATD at the same time. These blocks are not always used and moreover in the datapath is using several pipeline stages among which the first ones have a huge parallelism (the first pipeline stage consists of two register each of them composed by 2048 bits). Therefore a good improvement in the dynamic power is to adopt the **Clock Gating** technique. Every register does not have to sample its inputs its clock is disables and, thence, the number of commutations is strongly reduced. Clock gating needs a circuit level modification. In this FPGA Board by Altera, the insertion of the clock gating is not performed at synthesis time with the usage of some extra command, but instead the only way to introduce this type of optimization is to manually insert the clock gating block. The Clock Control Block is an Altera IP and it is called ALTCLKCTRL. It is a dynamic clock buffer that allows you to enable and disable the clock network and dynamically switch between multiple sources to drive the clock network.
The Clock Control Block are defined as:

- Global Clock Network: a clock can reach all the parts inside the chip.
- Regional Clock Network: a clock can drive a quadrant of the chip.
- External Clock-Out Path: clock path from the output of the PLL to the dedicated pins.

The main idea was therefore to use this IP block with the aim of reduce the dynamic power consumed. The design must contain an enable condition in order to use and benefit from

clock gating, but this type of logic must not consume more power than the one that was saved with the usage of the clock gating technique. Therefore in this work, a coarse-grain clock gating was adopted, meaning that the clock coming out of the PLL was gated by the ALTCLKCTRL block and then distributed toward all the registers. In this way the enable logic can be very simple, meaning that it can be handled by a simple FSM that has an ON-OFF mechanism. The FSM has to understand when the clock has to be activated, remembering that in order to reduce the latency for the calculation of the result, the value of the first address on the memory will precharged on the first pipeline register. In Figure 9.1 is shows a functional timing waveform example for clock-output enable. Clock enable is synchronous with the falling edge of the input clock. Knowing this, it is important to activate the register at least one clock cycle before the start of the evaluation and in that sense, the clock can be activated when the current block memory is written. As mentioned in the previous chapters, the writing of the memories anticipates the functioning of the structure, and the last memory that has to be written between the two is the current block one.

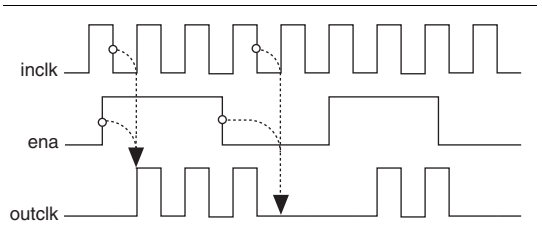


Figure 9.1: Clock Enable Timing

In fact once the clock enable rises, the clock is activated after an half clock cycle, meaning that, in order to preserving the precharge of the initial data on the first pipeline register, this clock enable must be activated earlier. In this case the `cur_write` enable is provided to the FSM in order to signal that the memory is being written and the clock can be enabled. At this point the clock is enabled until the computation is not finished, but if a new computation is ready, the clock is left enabled.

As stated in 7.4, due to the low confidence, the power analysis it was omitted.

This version of the hardware accelerator was tested using both the verification C program used in 6.2, but was also verified on the modified version of HM, that is discussed in (?).

9.2 Direct Memory Access

As explained in Chapter 4, one of the features of the HPS is the DMA controller (DMAC). The utilization of a DMA can be very efficient if the purpose is to move data from processor memories and FPGA. In fact in this case the quantity of data that must be sent to the FPGA may vary from 64 B to 4096 B for each transfer. This depends of course on the dimension of the block and it must be considered if the main intent is to reduce the throughput of the accelerator.

9.2.1 Linux Kernel Module

A complete and very detailed work on this type of FPGA was done by the Electronic Technology Department of the University of Vigo and it is well explained in [1]. This team exploited the performances of the HPS-FPGA bridges making a full detailed study on the timing with different configurations. These experiments represent the core of HPS-FPGA transfer rate measurements when using the HPS as master to move data and they provide a good overview of the device behavior. In particular they realized a *Kernel Module* with the capability of program the DMA Controller PL330 available on the Hard Processor System. This Loadable Kernel Module (LKM) moves data between a Linux Application running in User Space and a memory or other kind of peripheral in the FPGA using the DMA. At the end of this test it was established that for data sizes bigger than 128 B this method is faster than moving data with the processor using `memcpy()` function. The module uses the char driver interface to connect application space and the kernel space. It creates a node in */dev* called */dev/dma_pl330* and support for the the typical file functions is given: `open()`, `close()`, `write()` and `read()`. This way reading or writing to an FPGA address using the DMA is as easy as reading or writing into a file. The LKM also exports some variables using sysfs in */sys/dma_pl330/* to control its behavior. Among the different methods in order to move data from the user level to the FPGA, in Figure 9.2 is shown the one used adopted in this thesis.

When using ACP data is copied into L2 cache controller in coherent way so it is automatically coherent for the processor. The LKM contains the following variables to control its behavior. This variables are exported to the file system using sysfs (in */sys/dma_pl330/*). This variables control the basic behavior of the transfer:

- `use_acp`.
- `prepare_microcode_in_open`: PL330 DMA Controller executes a microcode defining the DMA transfer to be done.
- `dma_buff_padd`: the physical address for the FPGA.

The char device driver interface functions are:

- `dev_open`: called when `open()` is used. It prepares the DMA write and read microcode if `prepare_microcode_in_open`=1. To prepare the write microcode it uses cached buffer (if `use_acp`=1) as source, `dma_buff_padd` as destiny and `dma_transfer_size` as transfer size.
- `dev_write`: when using `write()` function the data is copied from the application using `copy_from_user()` function to cached buffer (if `use_acp`=1). `dev_read`: called when using `read()` to read from the FPGA.
- `dev_release`: called when callin the `close()` function from the application.

Transfer rates between HPS and FPGA when HPS is the master were measured for different combinations of values of the following parameters:

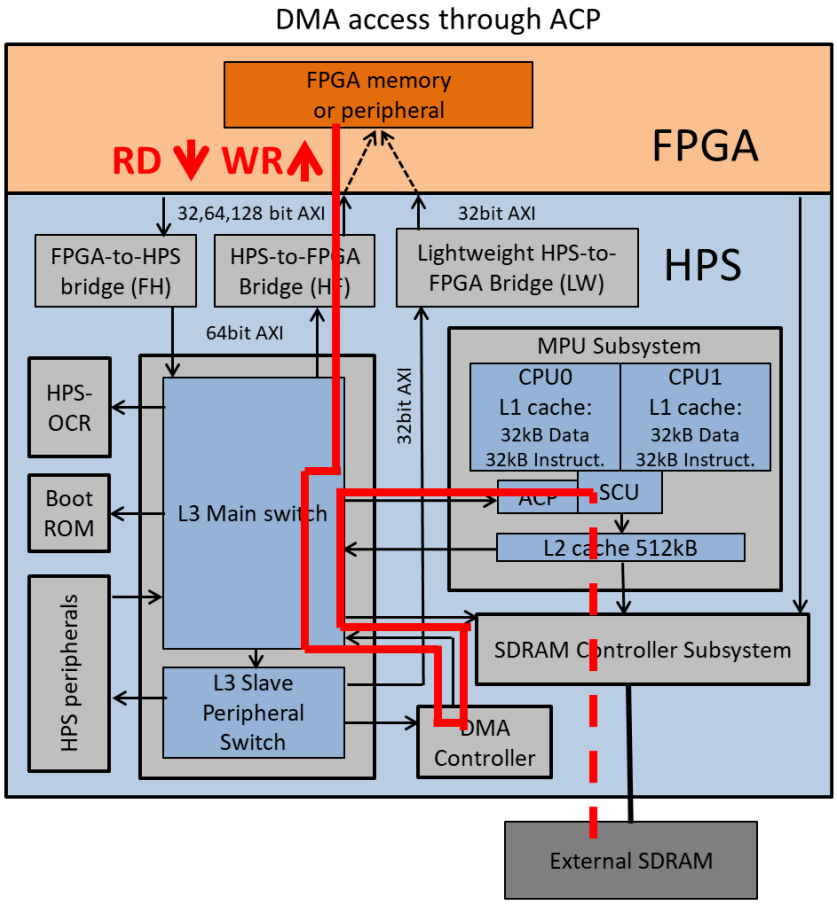


Figure 9.2: Data Route from HPS to FPGA [1]

- **OS** or **Baremetal**: evaluation on the transfer performances using a Linux-based OS for Intel FPGA devices, and as a Baremetal application running in one of the ARM cores.
- **Master starting AXI bus transfers**: a comparison of the performances when one of the ARM cores controls data transfer(`memcpy()`), and exploiting the DMA controller
- **Data size** from 2 B to 2 MB
- **FPGA frequency** from 50 Hz to 50 MHz.

Tests were repeated 100 times and mean value is given as result.

9.2.2 Compilation and Utilization

The Kernel Module was directly compiled on the board, after few modification on the /textit-makefile present in the repository of the project, that was compiled using the compiler in *Intel FPGA SoC EDS*. When the command *insmod* is executed to insert the Module into the kernel, the `DMA_PL330_LKM_init` function that is the initialization of the DMA Controller performs:

- Reservation of the Channel 0 to beused in DMA transaction,
- ioremaps HPS-physical address (is is used to store the DMAC microcode),
- allocates cached buffer using `kmalloc()`,
- exports the control variables using *sysfs* in */sys/dma_pl330/*,
- creates the char device driver interface in */dev/dma_pl330/*,
- configures ACP and enables PMU to be accessed from user space.

9.2.3 Integration for SATD accelerator

Several modification in the code were necessary for the integration of the Kernel Module in the peripheral driver. As mentioned before, this module was intended to use for the dispatch of the data of the block to the FPGA using the DMA controller. Having this type of procedure, can be useful because it moves away from the processor the workload of accessing the memory.

This type of optimization was tested only in the verification code, in order to compare the single block test to first of all check the functionality, but also to analyze the timing performances. After few modification of the actual code of the module in order to make it compatible with the distribution of Linux used in this project (this group used a different version called Angstrom), the variables mentioned in 9.2.1 were introduced in the code. The `use_acp` was set to 1, in order to maintain the coherency with process. Between the two memory present in the peripheral, the only one that receives the data using the DMA is the *current block* one. The reason is that every time the physical destination address has to change, the variable in */sys/dma_pl330/*, and this can introduce a not negligible overhead. Therefore the `dma_buff_padd` variable was fixed at the beginning and it does not change. The function `write` triggers the start of the sending of the data to the memory and at the end the result is retrieved using the Lightweight bus as before.

9.2.4 Time Measurements

After the development of this new type of configuration, the next step was to compare the performances given by the use of this driver. Analyzing the resuts of the tests in terms of timing done by [1], the employment of the DMA driver on the AXI bridge was attested to be better than `memcpy()` for a workload greater than 256 B, but knowing that the cases are limited, all the four cases were tested and how was convenient.

Tables 9.1 shows the time comparison between the calculation of a single block performed by software functions, using the FPGA peripheral with the usage of the DMA and using the `memcpy()`. The same criteria are present in Table 9.2, in which is performed a comparison between the time results of the emulation of the search for each type of block. It is possible to notice that the usage of DMAC improves the performances only in the last two (or three) cases. This contradicts what is stated in [1] because, in theory, it should produce better results starting from a the 16×16 block. It is also true that the software was not well adopted for this scope, leading to a not perfect improvement in terms of timing. in addiction the results for

the last cases are really impressive, leading to an improvement of the 25% on the FPGA-CPU computing, and, therefore, of the 89.8% on the software function.

Block Dimensions	Software (ns)	FPGA CPU (ns)	FPGA DMA	ΔT (%)
8 × 8	19045	6206	42467	+584.2
16 × 8	39654	11639	46522	+299.7
16 × 16	79145	21849	55996	+156.3
32 × 8	79523	22071	54902	+148.7
32 × 16	158018	42624	72661	+70.5
32 × 24	228502	61072	87254	+42.8
32 × 32	316196	83969	104442	+24.3
64 × 16	316400	83936	104216	+24.1
64 × 32	632481	167418	169497	+1.0
64 × 48	914330	241060	235248	-2.4
64 × 64	1267051	331850	300329	-9.5

Table 9.1: Experimental Time Measurements for each single block dimensions

Block Dimensions	Software (ns)	FPGA CPU (ns)	FPGA DMA	ΔT (%)
8 × 8	670097	109534	496945	+353.6
16 × 8	1338515	200372	626642	+212.7
16 × 16	2668982	379678	914615	+140.8
32 × 8	2679274	380912	790370	+107.4
32 × 16	5340850	740012	1097967	+48.3
32 × 24	8368034	1143228	1274091	+11.4
32 × 32	10676977	1462590	1646212	+12.5
64 × 16	10649720	1460058	1487139	+1.8
64 × 32	21346267	2897915	2461069	-15.1
64 × 48	32018153	4367719	3428765	-21.5
64 × 64	42726355	5766809	4322303	-25.0

Table 9.2: Experimental Time Measurements for each block dimensions emulating a search

Chapter 10

Approximation

A further analysis that was done in this work was to check the effective improvements and the effects of introducing in the architecture adders that are error tolerant. An error tolerant adder can introduce a certain quantity of error, depending on his type and implementation, and, if the task does not requires a perfect result, using this type of adder can give better results first of all in terms of delay and then of course of power and area. The basic idea of an approximate adder is to not propagate the carry through the whole Full Adders (like what happens in a proper Ripple Carry Adder) but instead break the carry chain and therefore create several segments of adders. In this way the critical path is highly reduced, but also the power consumed, because if with this improvement the delay is over reduced, a further reduction of the voltage can be applied meaning that there will be a quadratic reduction of the dynamic power. For what concerns the area, with the respect to a normal Ripple Carry Adder there are no essential changes. If we consider instead a different implementation like a Carry Look-Ahead adder, where there is a dedicate logic that propagate and generate the carry, it is possible to have a percentage of area saving due to the removal of the part that propagate the carry from a segment to another.

For this project the choice of the adder was crucial, because several adders and subtractors are in cascade in order to produce the result, and high percentage of error at the output of one adder would be propagated the result at the output would be compromised. As sad before the SATD are used as a metric in order to have a much more precise comparison between the reference block and the candidate one, and therefore the an adder that introduces an high quantity of error cannot be selected. In [18] is present a complete and exhaustive analysis of the different types and versions of approximate adders and the related characteristics in terms of delay, power and area.

10.1 Adder Selection

Between the overall solutions analyzed in this paper, the one that introduces the least quantity of error is the Error Tolerant Adder II (ETAII), that is proposed in [19]. The ETAII is essentially segmented Carry Select Adder in which every segment receives the carry from the previous segment but the propagation of the carry is interrupted, resulting in a massive reduction of

the critical path, that essentially will pass only through the carry generate-propagate block of one segment. It is shown in Figure 10.1, where n is the adder size, k is the size of the carry and sum generators. The carry signal from the previous carry generator propagates to the next sum generator. Therefore, ETAII utilizes more information to predict the carry bit. In addition to ETAII, several other error tolerant adders (ETAs) have been proposed by the same authors of [19].

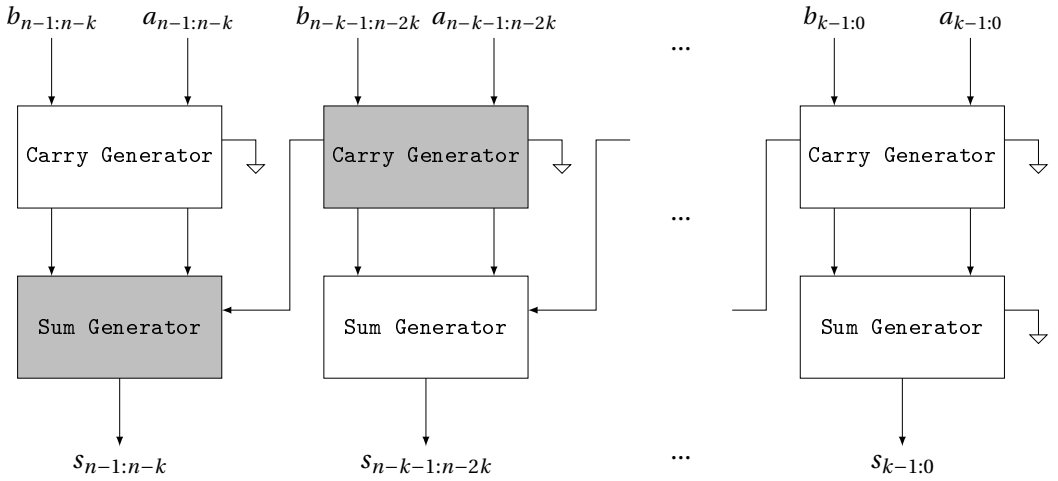


Figure 10.1: The Error-Tolerant Adder type II (ETAII) [19]: the carry propagates through the two shaded blocks.[19]

Although ETAII achieves good performances in both power and speed, the degraded accuracy for large input operands may still restrict its use. A modified structure was hence introduced to further improve the accuracy performance of ETAII. In this modified version, that is called ETAIIM [2], the higher order bits should be more accurate than the lower order bits as they play a more important role in representing a number. Therefore, for the higher order bit positions, more input bits should be considered when calculating the carry signals. In this structure, shown in the Fig 10.2 the first three carry generators are cascaded together to generate the carry signals for the two highest blocks. In this way, the carry signal for the highest block is generated by the preceding 12 bits and the carry signal for the second block is generated by the preceding 8 bits and so on. The rest of the circuit is the same as that of ETAII.

10.2 Implementation

As well explained in the previous chapters, the structure of the architecture the bit-length of a pixel is 8 but with at each stage of addition/subtraction it increases by 1 bit, and therefore the dimension of the adders must increase as well. The problem related to this was to create a generic adder that has the most significant bits in a fixed precision, instead the lower bits that follows the ETAIIM idea of the segmentation and increase its dimensions coherently. After the design of this architecture, it was substitute to the adder and subctractors already present in the design.

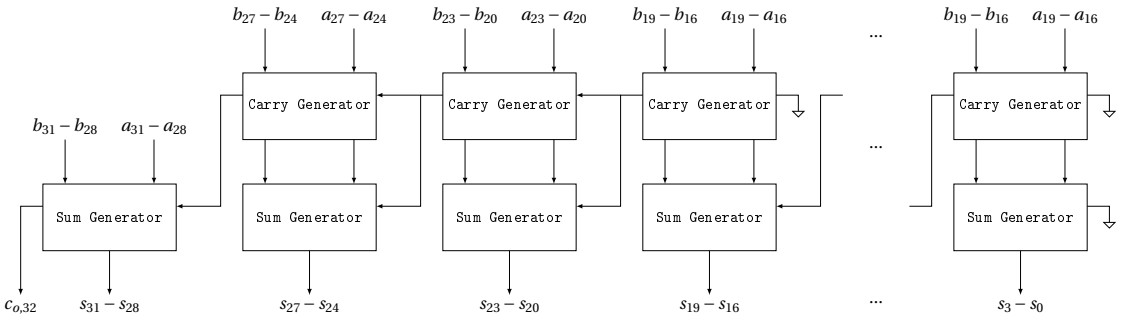


Figure 10.2: Block diagram of modified ETAII (ETAIIM) [2]

10.3 Results

The proposed implementation was tested in the HEVC test model HM 16.17 [8]. HM was configured in *All-Intra* mode, *Random-Access* mode and *Low Delay B* mode, and run for quantization parameters (QP) 22, 27, 32 and 37. Tables 10.1, 10.2 and 10.3 show the results of these experiments, in terms of $BD-Rate$ and $BD-PSNR_Y$ [20] for All-Intra and $BD-Rate$ and $BD-PSNR_{YUV}$ for the other two configurations, in which the peripheral is compared with HM. Using the Bjøntegaard metrics, the table shows the average differences in rate-distortion performance and the resulting bit rate efficiency. From the Table 10.1 it is possible to notice that in All-Intra mode, the impact of on both Bit rate and Rate Distortion is limited. Therefore, this type of approximation could be the starting point for a deeper analysis on this case study. In fact this modification did not carried any improvements for what concerns the time. After the synthesis the overall slack did not substantially changed, mainly due to the high optimization of the IP library for the adders on the FPGA. In general, an approximate adder is faster than a precise one, therefore, a modification like that could lead to important time improvements on an ASIC solution. Instead in Random Access and Low Delay (Table 10.2 and 10.3) the effect of the approximation is much more relevant, meaning that the FME requires an higher precision in order to perform the fine-grain search. As already stated, this type of work is considered also a Framework, and one of the purpose of this type of test is to provide a useful template for the future people that will want to study the real effect on HM of their own implementation. In Figure 10.3 it is shown the overall differences in between the default and the approximate solutions in terms of Rate Distortion.

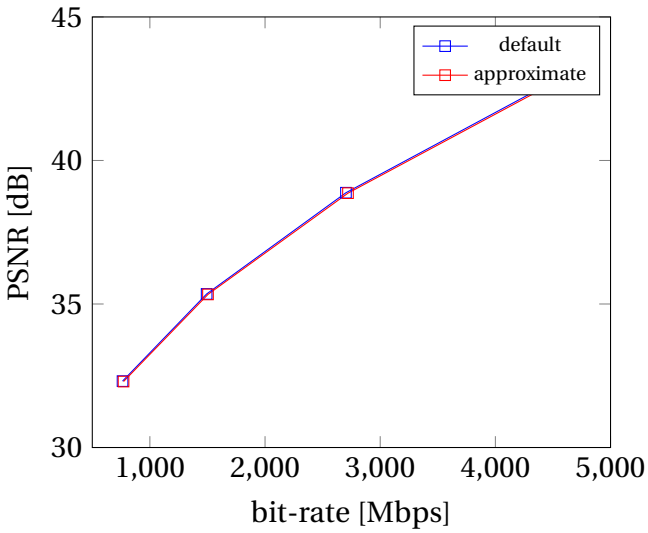


Figure 10.3: Comparison of the RD-cost

Class	Video Sequences	$BD-Rate(\%)$	$BD-PSNR_Y(dB)$
C	RaceHorses	+1.1606	-0.0668
	PartyScene	+0.4041	-0.0292
	BasketballDrill	+1.8556	-0.0911
	BQMall	+1.2160	-0.0683
D	RaceHorses	+0.9970	-0.0620
	BQSquare	+0.3770	-0.0299
	BlowingBubbles	+0.4001	-0.0276
	BasketballPass	+1.1853	-0.0737
Average		+0.9494	-0.0384

Table 10.1: Experimental results for recommended video sequences for All-Intra configuration

Class	Video Sequences	$BD-Rate(\%)$	$BD-PSNR_{YUV}(dB)$
D	RaceHorses	+6.9573	-0.3116
	BQSquare	+3.2680	-0.1428
	BlowingBubbles	+5.7415	-0.2079
	BasketballPass	+4.0849	-0.1904
Average		+5.0129	-0.2131

Table 10.2: Experimental results for recommended video sequences for Random Access configuration

Class	Video Sequences	$BD-Rate(\%)$	$BD-PSNR_{YUV}(dB)$
D	RaceHorses	6.5574	-0.3082
	BQSquare	6.9035	-0.2886
	BlowingBubbles	5.9241	-0.2182
	BasketballPass	4.1454	-0.1885
Average		5.8826	-0.2508

Table 10.3: Experimental results for recommended video sequences for Low Delay configuration

Chapter 11

Conclusion

11.1 Summary

The subject of the presented thesis is an hardware accelerator for FPGA that includes its own on-chip RAM memories able to perform all the SATD cases composed by 8×8 blocks, listed in Table 5.1, full compatible with the HEVC standard. In addition, a driver for HM reference software is implemented and tested. The entire project, including the Clock Gating optimization and the Approximate solution, is synthetized similarly as the base architecture as explained in Chapter 7 using the same constraints. The clock frequency related to the memories is kept at 150 MHz, while for the second clock is 66.66 MHz.

11.2 Results Analysis

During this work several tests were performed to evaluate the functioning, quality and the finally the performances of the peripheral. As it can be notice from the timing test, the results were quite different. The stand alone tests, in which the peripheral was compared with the software software computation in terms of times, were more than acceptable, with an average time reduction of 72.3% for the single block and of 85.8% on the emulation of the search. On the other hand the test perormed on the modified version of HM reference software did not reflect the huge improvement seen before. Many reasons could affect the system with the result of no time improvements. One of the main reason could be the fact that the HM reference software, as said in [11], employs only one thread on a processor. The copy of the block from the user level to the FPGA lead to a congestion of the memory instruction that highly affects the performances. In fact, as stated in [11], the `memcpy` function overall usage is comparable with the `TEncSearch` class (that contains the functions for both Inter and Intra Prediction). To overcome these problems, many solutions can be adopted:

- Embed the usage of the peripheral directly inside the functions that do the Intra Prediction and the FME. In fact, both of them, after the prediction, make a copy of the predicted block, with additional parts of the original block in the case of Intra Prediction. Therefore, the solution is to insert the sending of the reference block to the peripheral inside those function directly, instead of make such a copy.

- A dedicated thread to manage the sending of the data.

11.3 Future Works

The presented work is a complete hardware accelerator that contains local on-chip memory able to perform the SATD cases in the range of 8×8 to 64×64 described inside the HEVC standard. The number of pins and the input bandwidth constraints were taken into account during the design in order to provide an accelerator that is fully compatible with the FPGA adopted.

The clock gating control selectively switches off the clock signal to unused parts of the architecture reducing the Dynamic Power. A brief explaination on how the DMA can be use to send data as fast as possible to and from the FPGA was presented.

The final purpose of this work is not only to be considered as a creation of this type of architecture but it is also a complete and very helpful framework in order to work with HEVC using this kind of FPGA, and, in general, for the creation of more complex peripherals that has to interface with an Operative System. The steps that were needed to produce the `.rbf` file are identical for mostly of the similar FPGA chip from Altera and the entire project can be easily readopted to one of those. In fact the Avalon Memory Mapped interfaces that are used in this project are compatible with all the FPGA of the same family that contains, as the Cyclone V, the embedded ARM processor. The Intel FPGA are subdivided in four series: while Cyclone and Arria are essentially designed for prototyping and their performances are in the midrange, the Stratix series can lead to the considerably best results, because of the high performances.

This project will be upladed on the GitHub platform with the purpose of sharing the realization of this hardware accelerator, giving the possibility to those interested in working on it, and hopefully improve it. In addition, as said before, this is a complete and complex workflow for those type of applications in which the FPGA has to be interfaced with the core. This thesis, in addition to the official manuals, is a useful reference that could help whoever wants to implement a kind of peripheral that must be interfaced with a Linux OS powered from the internal ARM core.

11.4 Workflow Definition

The workflow that was drawn up in order to work with this kind of environment can be summarized in the following steps:

- **Definition the the work**: in this step the idea of what it has to be implemented must be defined.
- **Generation of the Quartus environment**: the system that will contain the design has to be realized using Quartus and Platform Designer.
- **Generation of interal HDL for the Custom Component**.
- **Testing** the component using a testbench that acts on the generated interfaces.

- **Synthesis** of the final component and test on-board with a C software.
- **Integration of the driver** in the required application.

Appendix A

Reports

A.1 Resources Usage Summary

Listing A.1: Resources Usage Summary

```
+-----------------------------------------------------------------------------+
; Fitter Resource Usage Summary                                               ;
+---------------------------------------------+-----------------------+-------+
; Resource                                    ; Usage                 ; %     ;
+---------------------------------------------+-----------------------+-------+
; Logic utilization                           ;                       ;       ;
; (ALMs needed / total ALMs on device)        ; 14,967 / 32,070       ; 47 %  ;
; ALMs needed [=A-B+C]                        ; 14,967                ;       ;
;   [A] ALMs used in final placement [=a+b+c+d]; 15,638 / 32,070       ; 49 %  ;
;     [a] ALMs used for LUT logic and registers; 3,602                ;       ;
;     [b] ALMs used for LUT logic             ; 11,036                ;       ;
;     [c] ALMs used for registers             ; 1,000                 ;       ;
;     [d] ALMs used for memory                ;                       ;       ;
;         (up to half of total ALMs)          ; 0                     ;       ;
;   [B] Estimate of ALMs recoverable          ;                       ;       ;
;       by dense packing                      ; 1,009 / 32,070        ; 3 %   ;
;   [C] Estimate of ALMs unavailable [=a+b+c+d]; 338 / 32,070          ; 1 %   ;
;     [a] Due to location constrained logic   ; 0                     ;       ;
;     [b] Due to LAB-wide signal conflicts    ; 0                     ;       ;
;     [c] Due to LAB input limits             ; 338                   ;       ;
;     [d] Due to virtual I/Os                 ; 0                     ;       ;
;                                             ;                       ;       ;
; Difficulty packing design                   ; Low                   ;       ;
;                                             ;                       ;       ;
; Total LABs:  partially or completely used   ; 2,307 / 3,207         ; 72 %  ;
;   -- Logic LABs                             ; 2,307                 ;       ;
;   -- Memory LABs (up to half of total LABs) ; 0                     ;       ;
;                                             ;                       ;       ;
; Combinational ALUT usage for logic          ; 29,088                ;       ;
;   -- 7 input functions                       ; 16                    ;       ;
;   -- 6 input functions                       ; 189                   ;       ;
;   -- 5 input functions                       ; 244                   ;       ;
;   -- 4 input functions                       ; 254                   ;       ;
;   -- <=3 input functions                     ; 28,385                ;       ;
; Combinational ALUT usage for route-throughs ; 974                   ;       ;
;                                             ;                       ;       ;
; Dedicated logic registers                   ; 9,315                 ;       ;
;   -- By type:                               ;                       ;       ;
```

```
;      -- Primary logic registers                ; 9,203 / 64,140          ; 14 %   ;
;      -- Secondary logic registers              ; 112 / 64,140            ; < 1 %  ;
;    -- By function:                             ;                         ;       ;
;      -- Design implementation registers        ; 9,218                   ;       ;
;      -- Routing optimization registers         ; 97                      ;       ;
;                                                ;                         ;       ;
; Virtual pins                                   ; 0                       ;       ;
; I/O pins                                       ; 73 / 457                ; 16 %  ;
;    -- Clock pins                               ; 1 / 8                   ; 13 %  ;
;    -- Dedicated input pins                     ; 0 / 21                  ; 0 %   ;
; I/O registers                                  ; 226                     ;       ;
;                                                ;                         ;       ;
; Hard processor system peripheral utilization   ;                         ;       ;
;    -- Boot from FPGA                           ; 1 / 1 ( 100 % )         ;       ;
;    -- Clock resets                             ; 1 / 1 ( 100 % )         ;       ;
;    -- Cross trigger                            ; 0 / 1 ( 0 % )           ;       ;
;    -- S2F AXI                                  ; 1 / 1 ( 100 % )         ;       ;
;    -- F2S AXI                                  ; 1 / 1 ( 100 % )         ;       ;
;    -- AXI Lightweight                          ; 1 / 1 ( 100 % )         ;       ;
;    -- SDRAM                                    ; 1 / 1 ( 100 % )         ;       ;
;    -- Interrupts                               ; 0 / 1 ( 0 % )           ;       ;
;    -- JTAG                                     ; 0 / 1 ( 0 % )           ;       ;
;    -- Loan I/O                                 ; 0 / 1 ( 0 % )           ;       ;
;    -- MPU event standby                        ; 0 / 1 ( 0 % )           ;       ;
;    -- MPU general purpose                      ; 0 / 1 ( 0 % )           ;       ;
;    -- STM event                                ; 0 / 1 ( 0 % )           ;       ;
;    -- TPIU trace                               ; 1 / 1 ( 100 % )         ;       ;
;    -- DMA                                      ; 0 / 1 ( 0 % )           ;       ;
;    -- CAN                                      ; 0 / 2 ( 0 % )           ;       ;
;    -- EMAC                                     ; 0 / 2 ( 0 % )           ;       ;
;    -- I2C                                      ; 0 / 4 ( 0 % )           ;       ;
;    -- NAND Flash                               ; 0 / 1 ( 0 % )           ;       ;
;    -- QSPI                                     ; 0 / 1 ( 0 % )           ;       ;
;    -- SDMMC                                    ; 0 / 1 ( 0 % )           ;       ;
;    -- SPI Master                               ; 0 / 2 ( 0 % )           ;       ;
;    -- SPI Slave                                ; 0 / 2 ( 0 % )           ;       ;
;    -- UART                                     ; 0 / 2 ( 0 % )           ;       ;
;    -- USB                                      ; 0 / 2 ( 0 % )           ;       ;
;                                                ;                         ;       ;
; M10K blocks                                    ; 128 / 397               ; 32 %  ;
; Total MLAB memory bits                         ; 0                       ;       ;
; Total block memory bits                        ; 65,536 / 4,065,280      ; 2 %   ;
; Total block memory implementation bits         ; 1,310,720 / 4,065,280   ; 32 %  ;
;                                                ;                         ;       ;
; Total DSP Blocks                               ; 0 / 87                  ; 0 %   ;
;                                                ;                         ;       ;
; Fractional PLLs                                ; 1 / 6                   ; 17 %  ;
; Global signals                                 ; 4                       ;       ;
;    -- Global clocks                            ; 4 / 16                  ; 25 %  ;
;    -- Quadrant clocks                          ; 0 / 66                  ; 0 %   ;
;    -- Horizontal periphery clocks              ; 0 / 18                  ; 0 %   ;
; SERDES Transmitters                            ; 0 / 100                 ; 0 %   ;
; SERDES Receivers                               ; 0 / 100                 ; 0 %   ;
; JTAGs                                          ; 0 / 1                   ; 0 %   ;
; ASMI blocks                                    ; 0 / 1                   ; 0 %   ;
; CRC blocks                                     ; 0 / 1                   ; 0 %   ;
; Remote update blocks                           ; 0 / 1                   ; 0 %   ;
; Oscillator blocks                              ; 0 / 1                   ; 0 %   ;
; Impedance control blocks                       ; 1 / 4                   ; 25 %  ;
; Hard Memory Controllers                        ; 1 / 2                   ; 50 %  ;
; Average interconnect usage (total/H/V)         ; 19.3% / 18.3% / 22.3%   ;       ;
; Peak interconnect usage (total/H/V)            ; 45.9% / 46.8% / 55.1%   ;       ;
; Maximum fan-out                                ; 8690                    ;       ;
```

```
; Highest non-global fan-out                    ; 4096                  ;       ;
; Total fan-out                                 ; 140928                ;       ;
; Average fan-out                               ; 3.50                  ;       ;
+-----------------------------------------------+-----------------------+-------+
```

A.2 Time Reports

Listing A.2: Timing Report

```
+-----------------------------------------------------------------------------+
; Slow 1100mV 85C Model Fmax Summary                                           ;
+-----------------------------------------------------------------------------+
; Fmax        ; Restricted Fmax ; Clock Name                                   ;
+-----------------------------------------------------------------------------+
; 68.47 MHz   ; 68.47 MHz       ; ...|general[1].gpll~PLL_OUTPUT_COUNTER|divclk ;
; 150.56 MHz  ; 150.56 MHz      ; ...|general[0].gpll~PLL_OUTPUT_COUNTER|divclk ;
; 1184.83 MHz ; 717.36 MHz      ; ...|afi_clk_write_clk                        ;
-------------------------------------------------------------------------------+


+-----------------------------------------------------------------------------+
; Slow 1100mV 85C Model Setup Summary                                          ;
+-----------------------------------------------------+-------+---------------+
; Clock                                               ; Slack ; End Point TNS ;
+-----------------------------------------------------+-------+---------------+
; u0|...|general[0].gpll~PLL_OUTPUT_COUNTER|divclk      ; 0.024 ; 0.000         ;
; system:u0|...|hps_sdram_pll:pll|afi_clk_write_clk     ; 1.730 ; 0.000         ;
; u0|...|general[1].gpll~PLL_OUTPUT_COUNTER|divclk      ; 2.060 ; 0.000         ;
+-----------------------------------------------------+-------+---------------+


+-----------------------------------------------------------------------------+
; Slow 1100mV 85C Model Hold Summary                                           ;
+-----------------------------------------------------+-------+---------------+
; Clock                                               ; Slack ; End Point TNS ;
+-----------------------------------------------------+-------+---------------+
; system:u0|...|hps_sdram_pll:pll|afi_clk_write_clk     ; 0.143 ; 0.000         ;
; u0|...|general[1].gpll~PLL_OUTPUT_COUNTER|divclk      ; 0.223 ; 0.000         ;
; u0|...|general[0].gpll~PLL_OUTPUT_COUNTER|divclk      ; 0.227 ; 0.000         ;
+-----------------------------------------------------+-------+---------------+


+-----------------------------------------------------------------------------+
; Slow 1100mV 85C Model Recovery Summary                                       ;
+-----------------------------------------------------+-------+---------------+
; Clock                                               ; Slack ; End Point TNS ;
+-----------------------------------------------------+-------+---------------+
; u0|...|general[0].gpll~PLL_OUTPUT_COUNTER|divclk      ; 2.767 ; 0.000         ;
; system:u0|...|hps_sdram_pll:pll|afi_clk_write_clk     ; 3.375 ; 0.000         ;
; u0|...|general[1].gpll~PLL_OUTPUT_COUNTER|divclk      ; 9.161 ; 0.000         ;
+-----------------------------------------------------+-------+---------------+


+-----------------------------------------------------------------------------+
; Slow 1100mV 85C Model Removal Summary                                        ;
+-----------------------------------------------------+-------+---------------+
; Clock                                               ; Slack ; End Point TNS ;
+-----------------------------------------------------+-------+---------------+
; system:u0|...|hps_sdram_pll:pll|afi_clk_write_clk     ; 0.470 ; 0.000         ;
; u0|...|general[0].gpll~PLL_OUTPUT_COUNTER|divclk      ; 1.096 ; 0.000         ;
```

```
; u0|...|general[1].gpll~PLL_OUTPUT_COUNTER|divclk      ; 4.000 ; 0.000          ;
+-------------------------------------------------------+-------+---------------+


+-------------------------------------------------------------------------------+
; Slow 1100mV 85C Model Minimum Pulse Width Summary                              ;
+-------------------------------------------------------+-------+---------------+
; Clock                                                 ; Slack ; End Point TNS ;
+-------------------------------------------------------+-------+---------------+
; system:u0|...|pll_write_clk_dq_write_clk              ; 0.608 ; 0.000          ;
; system:u0|...|hps_sdram_pll:pll|afi_clk_write_clk     ; 0.623 ; 0.000          ;
; u0|...|general[0].gpll~FRACTIONAL_PLL|vcoph[0]        ; 1.666 ; 0.000          ;
; u0|...|general[0].gpll~PLL_OUTPUT_COUNTER|divclk      ; 2.091 ; 0.000          ;
; u0|...|general[1].gpll~PLL_OUTPUT_COUNTER|divclk      ; 7.111 ; 0.000          ;
; CLOCK_50                                              ; 9.670 ; 0.000          ;
+-------------------------------------------------------+-------+---------------+
```

Appendix B

C Code

B.1 Modified SetDistParam Function

Listing B.1: `SetDistParam` function

```
Void TComRdCost::setDistParam( DistParam& rcDP, Int bitDepth, const Pel* p1, \\
                               Int iStride1, const Pel* p2, Int iStride2, \\
                               Int iWidth, Int iHeight, FPGAParam* rcFP, \\
                               Bool bHadamard )
{
  rcDP.pOrg          = p1;
  rcDP.pCur          = p2;
  rcDP.iStrideOrg    = iStride1;
  rcDP.iStrideCur    = iStride2;
  rcDP.iCols         = iWidth;
  rcDP.iRows         = iHeight;
  rcDP.iStep         = 1;
  rcDP.iSubShift     = 0;
  rcDP.bitDepth      = bitDepth;
  rcDP.DistFunc      = m_afpDistortFunc[ ( bHadamard ? DF_HADS : DF_SADS ) + \\
                       g_aucConvertToBit[ iWidth ] + 1 ];
  rcDP.m_maximumDistortionForEarlyExit = std::numeric_limits<Distortion>::max();
  rcDP.useFPGA     = false;

  if ( (bitDepth == 8) && (bHadamard != 0) && (( (( iHeight % 8) == 0 ) && \\
     ((iWidth % 8) == 0 )) && (( iHeight < 65) && (iWidth < 65))) ) {
    rcDP.useFPGA     = true;

  Void *h2p_lw_base_ctrl_addr = rcFP->m_h2p_lw_base_ctrl_addr;

  UINT_SOC* on_chip_RAM_addr_REF = rcFP->m_on_chip_RAM_addr_REF;
  Int transfer_size =  iHeight *iWidth;

      // Configuration of the FPGA peripheral
      // - 0x00 ---> 8x8
      // - 0x01 ---> 16x16 or 32x8
      // - 0x02 ---> 32x32 or 64x16
      // - 0x03 ---> 64x64
      // - 0x04 ---> 16x8
      // - 0x05 ---> 32x16
      // - 0x06 ---> 32x24
      // - 0x07 ---> 64x32
      // - 0x08 ---> 64x48
```

```
switch (iWidth) {
case 8:
    switch (iHeight) {
      case 8:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x00;
        break;
      case 16:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x04;
        break;
      case 32:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x01;
        break;
    }
    break;
  case 16:
  switch (iHeight) {
      case 8:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x04;
        break;
      case 16:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x01;
        break;
      case 32:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x05;
        break;
      case 64:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x02;
        break;
    }
    break;
  case 32:
    switch (iHeight) {
      case 8:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x01;
        break;
      case 16:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x05;
        break;
      case 24:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x06;
        break;
      case 32:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x02;
        break;
      case 64:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x07;
        break;
    }
    break;
  case 64:
    switch (iHeight) {
      case 16:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x02;
        break;
      case 32:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x07;
        break;
      case 48:
        *((uint8_t *)h2p_lw_base_ctrl_addr)=0x08;
        break;
      case 64:
```

```
              *((uint8_t *)h2p_lw_base_ctrl_addr)=0x03;
              break;
            break;
          }
          break;
      default:
          *((uint8_t *)h2p_lw_base_ctrl_addr)=0x00;
            break;
      }

    char buffer[transfer_size];

    // Re-Ordering of the Block
    int  x, y, i, j, k, h, v, n;
    int  iOffsetOrg = iStride1<<3;
    i=0;
    j=0;
    h=0;
    v=0;
    n=0;
    //-------------
    for ( y=0; y<iHeight; y+=8 ) {
      for ( x=0; x<iWidth; x+=8 ) {
        v=x+h;
        for ( k=0; k<8; k++ ) {
          n = v + j;
          buffer[i]   = p1[n];
          buffer[i+1] = p1[n+1];
          buffer[i+2] = p1[n+2];
          buffer[i+3] = p1[n+3];
          buffer[i+4] = p1[n+4];
          buffer[i+5] = p1[n+5];
          buffer[i+6] = p1[n+6];
          buffer[i+7] = p1[n+7];
          i +=  8;
          j += iStride1;

        }
        j=0;
      }
      h += iOffsetOrg;
    }
    //----------

    // Sending to the FPGA

  memcpy((void*) on_chip_RAM_addr_REF, &buffer, transfer_size);

  rcDP.m_FPGAParam = rcFP;

  }
}
```

B.2 xGetHAD

Listing B.2: `xGetHAD` function

```
Distortion TComRdCost::xGetHADs( DistParam* pcDtParam )
{
  if ( pcDtParam->bApplyWeight )
```

```
  {
    return TComRdCostWeightPrediction::xGetHADsw( pcDtParam );
  }
  const Pel* piOrg       = pcDtParam->pOrg;
  const Pel* piCur       = pcDtParam->pCur;
  const Int  iRows       = pcDtParam->iRows;
  const Int  iCols       = pcDtParam->iCols;
  const Int  iStrideCur  = pcDtParam->iStrideCur;
  const Int  iStrideOrg  = pcDtParam->iStrideOrg;
  const Int  iStep       = pcDtParam->iStep;
  Int  x, y;
  Distortion uiSum = 0;
  if( ( iRows % 8 == 0) && (iCols % 8 == 0) )
  {
    Int  iOffsetOrg = iStrideOrg<<3;
    Int  iOffsetCur = iStrideCur<<3;
    for ( y=0; y<iRows; y+= 8 )
    {
      for ( x=0; x<iCols; x+= 8 )
      {
        uiSum += xCalcHADs8x8( &piOrg[x], &piCur[x*iStep], \\
                               iStrideOrg, iStrideCur, iStep );
      }
      piOrg += iOffsetOrg;
      piCur += iOffsetCur;
    }
  }
  else if( ( iRows % 4 == 0) && (iCols % 4 == 0) )
  {
    Int  iOffsetOrg = iStrideOrg<<2;
    Int  iOffsetCur = iStrideCur<<2;
    for ( y=0; y<iRows; y+= 4 )
    {
      for ( x=0; x<iCols; x+= 4 )
      {
        uiSum += xCalcHADs4x4( &piOrg[x], &piCur[x*iStep], \\
                               iStrideOrg, iStrideCur, iStep );
      }
      piOrg += iOffsetOrg;
      piCur += iOffsetCur;
    }
  }
  else if( ( iRows % 2 == 0) && (iCols % 2 == 0) )
  {
    Int  iOffsetOrg = iStrideOrg<<1;
    Int  iOffsetCur = iStrideCur<<1;
    for ( y=0; y<iRows; y+=2 )
    {
      for ( x=0; x<iCols; x+=2 )
      {
        uiSum += xCalcHADs2x2( &piOrg[x], &piCur[x*iStep], \\
                               iStrideOrg, iStrideCur, iStep );
      }
      piOrg += iOffsetOrg;
      piCur += iOffsetCur;
    }
  }
  else
  {
    assert(false);
  }
  return ( uiSum >> D_P_A(pcDtParam->bitDepth-8) );
}
```

B.3 Modified xGetHAD Function

Listing B.3: `xGetHAD` modified function

```
Distortion TComRdCost::xGetHADs( DistParam* pcDtParam )
{

  if ( pcDtParam->bApplyWeight )
  {
    return TComRdCostWeightPrediction::xGetHADsw( pcDtParam );
  }
  const Pel* piOrg       = pcDtParam->pOrg;
  const Pel* piCur       = pcDtParam->pCur;
  const Int  iRows       = pcDtParam->iRows;
  const Int  iCols       = pcDtParam->iCols;
  const Int  iStrideCur  = pcDtParam->iStrideCur;
  const Int  iStrideOrg  = pcDtParam->iStrideOrg;
  const Int  iStep       = pcDtParam->iStep;

  const Bool useFPGA  = pcDtParam->useFPGA;

  Int  x, y;

  Distortion uiSum = 0;
  if (useFPGA) {

      const FPGAParam* pcFPGAParam = pcDtParam->m_FPGAParam;
      //const Int  transfer_size = pcFPGAParam->m_transfer_size;
    const Int  transfer_size = iRows * iCols;
    const Void* on_chip_RAM_addr_CUR =
          pcFPGAParam->m_on_chip_RAM_addr_CUR;
    const Void* h2p_lw_base_res_addr =
          pcFPGAParam->m_h2p_lw_base_res_addr;

    char buffer[transfer_size];

    //Re-Order of the Block
    int  a, b, i, j, k, h, v, n;
    int  iOffsetCur = iStrideCur<<3;
    i=0;
    j=0;
    h=0;
    v=0;

    for ( b=0; b<iRows; b+=8 ){
      for ( a=0; a<iCols; a+=8 ) {
        v=a+h;
        for ( k=0; k<8; k++ ) {
          n = v + j;
          buffer[i]   = piCur[n];
          buffer[i+1] = piCur[n+1];
          buffer[i+2] = piCur[n+2];
          buffer[i+3] = piCur[n+3];
          buffer[i+4] = piCur[n+4];
          buffer[i+5] = piCur[n+5];
          buffer[i+6] = piCur[n+6];
          buffer[i+7] = piCur[n+7];
          i +=  8;
          j += iStrideCur;

        }
        j=0;
      }
```

```
    h += iOffsetCur;
  }
  //---------

  // Send Data of Current Block to FPGA

  memcpy((void*) on_chip_RAM_addr_CUR, &buffer, transfer_size);

  // Retrieve the Result

  uiSum = (Distortion)(*((uint32_t *)h2p_lw_base_res_addr));
}
else {
  if( ( iRows % 8 == 0) && (iCols % 8 == 0) )
  {
    Int  iOffsetOrg = iStrideOrg<<3;
    Int  iOffsetCur = iStrideCur<<3;
    for ( y=0; y<iRows; y+= 8 )
    {
      for ( x=0; x<iCols; x+= 8 )
      {
        uiSum += xCalcHADs8x8( &piOrg[x], &piCur[x*iStep], \\
                               iStrideOrg, iStrideCur, iStep );
      }
      piOrg += iOffsetOrg;
      piCur += iOffsetCur;
    }
  }
  else if( ( iRows % 4 == 0) && (iCols % 4 == 0) )
  {
    Int  iOffsetOrg = iStrideOrg<<2;
    Int  iOffsetCur = iStrideCur<<2;

    for ( y=0; y<iRows; y+= 4 )
    {
      for ( x=0; x<iCols; x+= 4 )
      {
        uiSum += xCalcHADs4x4( &piOrg[x], &piCur[x*iStep], \\
                               iStrideOrg, iStrideCur, iStep );
      }
      piOrg += iOffsetOrg;
      piCur += iOffsetCur;
    }
  }
  else if( ( iRows % 2 == 0) && (iCols % 2 == 0) )
  {
    Int  iOffsetOrg = iStrideOrg<<1;
    Int  iOffsetCur = iStrideCur<<1;
    for ( y=0; y<iRows; y+=2 )
    {
      for ( x=0; x<iCols; x+=2 )
      {
        uiSum += xCalcHADs2x2( &piOrg[x], &piCur[x*iStep], \\
                                iStrideOrg, iStrideCur, iStep );
      }
      piOrg += iOffsetOrg;
      piCur += iOffsetCur;
    }
  }
  else
  {
    assert(false);
  }
```

```
  }
  return ( uiSum >> D_P_A(pcDtParam->bitDepth-8) );
```

Bibliography

[1] R. F. M. Jose Farina Rodriguez, Juan Jose Rodriguez Andina, "Uvigo DTE FPSoC," https://github.com/UviDTE-FPSoC, 2017.

[2] M. Weber, M. Putic, H. Zhang, J. Lach, and J. Huang, "Balancing adder for error tolerant applications," in *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*, May 2013, pp. 3038–3041.

[3] K. Sayood, *Introduction to Data Compression, Fourth Edition*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[4] J. R. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan, and T. Wiegand, "Comparison of the Coding Efficiency of Video Coding Standards - Including High Efficiency Video Coding (HEVC)," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1669–1684, Dec 2012.

[5] V. Sze, M. Budagavi, and G. J. Sullivan, *High Efficiency Video Coding (HEVC): Algorithms and Architectures*. Springer Publishing Company, Incorporated, 2014.

[6] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1649–1668, Dec 2012.

[7] J. Lainema, F. Bossen, W. J. Han, J. Min, and K. Ugur, "Intra Coding of the HEVC Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1792–1801, Dec 2012.

[8] J. C. T. on Video Coding (JCT-VC), "HEVC Test Model (HM) Version 16.15." http://hevc.hhi.fraunhofer.de/.

[9] Joint Collaborative Team on Video Coding (JCT-VC), "HEVC Test Model (HM) Version 16.15," ftp://ftp.tnt.uni-hannover.de/testsequences/.

[10] M. Shafique and J. Henkel, "Low power design of the next-generation high efficiency video coding," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2014, pp. 274–281.

[11] F. Bossen, B. Bross, K. Suhring, and D. Flynn, "Hevc complexity and implementation analysis," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1685–1696, Dec 2012.

[12] E. Silveira, C. Diniz, M. B. Fonseca, and E. Costa, "SATD hardware architecture based on 8×8 Hadamard Transform for HEVC encoder," in *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, Dec 2015, pp. 576–579.

[13] F. Saab, I. H. Elhajj, A. Kayssi, and A. Chehab, "Profiling of HEVC encoder," *Electronics Letters*, vol. 50, no. 15, pp. 1061–1063, July 2014.

[14] G. Correa, P. Assuncao, L. Agostini, and L. A. da Silva Cruz, "Performance and Computational Complexity Assessment of High-Efficiency Video Encoders," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1899–1909, Dec 2012.

[15] Intel Altera, "Using Linux on the DE1-SoC," https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/DE1-SoC-UP-Linux/Linux.pdf.

[16] Intel FPGA, "Documentation: Manuals," https://www.altera.com/support/literature/lit-manual.html.

[17] Altera Wiki, "Intro to Altera SoC Devices for HW Developers Workshop - Configure the HPS — Altera Wiki," http://www.alterawiki.com/wiki/Intro_to_Altera_SoC_Devices_for_HW_Developers_Workshop_-_Configure_the_HPS, 2014, Accessed 1-December-2015.

[18] H. Jiang, J. Han, and F. Lombardi, "A Comparative Review and Evaluation of Approximate Adders," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, ser. GLSVLSI '15. New York, NY, USA: ACM, 2015, pp. 343–348.

[19] N. Zhu, W. L. Goh, and K. S. Yeo, "An enhanced low-power high-speed Adder For Error-Tolerant application," in *Proceedings of the 2009 12th International Symposium on Integrated Circuits*, Dec 2009, pp. 69–72.

[20] G. Bjontegaard, "Calculation of average PSNR differences between RD-Curves," 01 2001.