Parallel Implementation of Sample Adaptive Offset Filtering Block for Low-Power HEVC Chip

by

Luis A. Fernández Lara

B.S., Massachusetts Institute of Technology (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

© Massachusetts Institute of Technology 2015. All rights reserved.

Author ................................................................

Department of Electrical Engineering and Computer Science
May 15, 2015

Certified by ............................................................

Anantha P. Chandrakasan
Joseph F. and Nancy P. Keithley Professor of Electrical Engineering
Thesis Supervisor

Accepted by ...........................................................

Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee
Parallel Implementation of Sample Adaptive Offset Filtering Block for Low-Power HEVC Chip

by

Luis A. Fernández Lara

Submitted to the Department of Electrical Engineering and Computer Science on May 15, 2015, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents a highly parallelized and low latency implementation of the Sample Adaptive Offset (SAO) filter, as part of a High Efficiency Video Coding (HEVC) chip under development for use in low power environments. The SAO algorithm is detailed and an algorithm suitable for parallel processing using offset processing blocks is analyzed. Further, the SAO block hardware architecture is discussed, including the pixel producer control module, 16 parallel pixel processors and storage modules used to perform SAO. After synthesis, the resulting SAO block is composed of about 36.5 kgates, with an SRAM sized at 6KBytes. Preliminary results yield a low latency of one clock cycle on average (10 ns for a standard 100Mhz clock) per 16 samples processed. This translates to a best case steady state throughput of 200 MBytes per second, enough to output 1080p (1920x1080) video at 60 frames per second.

Furthermore, this thesis also presents the design and implementation of input/output data interfaces for an FPGA based real-life demo of the before-mentioned HEVC Chip under development. Two separate interfaces are described for use in a Xilinx VC707 Evaluation Board, one based on the HDMI protocol and the other based on the SD Card protocol. In particular, the HDMI interface implemented is used to display decoded HEVC video in an HD display at a 1080p (1920x1080) resolution with a 60Hz refresh rate. Meanwhile, the data input system built on top of the SD Card interface provides encoded bitstream data directly to the synthesized HEVC Chip via the CABAC Engine at rates of up to 1.5 MBytes per second. Finally, verification techniques for the FPGA real-life demo are presented, including the use of the on-board DDR3 RAM present in the Xilinx VC707 Evaluation Board.

Thesis Supervisor: Anantha P. Chandrakasan
Title: Joseph F. and Nancy P. Keithley Professor of Electrical Engineering
Acknowledgments

First I would like to thank my thesis supervisor, Professor Anantha Chandrakasan, for giving me the opportunity to work on this thesis and be part of the Energy Efficient Circuits and Systems Group at MIT. His generous support, invaluable advice and overall guidance have been an essential component in the journey that has been this thesis.

This thesis would not exist without a tremendous amount of support from Mehul Tikekar, who was always there to provide a seemingly interminable amount of knowledge and good will. Best of luck and thank you.

Thank you to my friends here at MIT and back home, of whom I cannot understated the importance of their support and overall genius. I feel extremely proud to have a chance to share and grow with such a group of magnificent individuals.

A special thank you to Bara Badwan for being an interesting person, to Marc Kaloustian for sharing my interests, to Francine Loza for worrying about my nutrition, to Demetra Sklaviadis for her reassurance and to Elisa Castañer for sending that fateful Facebook message in 2010.

Finally I want to thank my family for their unwavering guidance, belief and care. My mom, dad and sisters have always been there for me and are the driving force behind my life. Thank you to my mom, who has kept me sane all these years, and to my dad, who on my first day at MIT told me to "never take a step back". They are the rock on which I have stood on all my life and, more than ever, these past 5 years. This thesis is dedicated to you.
Contents

1 Chapter 1: Sample Adaptive Offset Filter Design and Architecture 15
  1.1 Introduction ................................................. 15
  1.2 Sample Adaptive Offset Filter .............................. 16
    1.2.1 Edge Offset ........................................ 19
    1.2.2 Band Offset ......................................... 20
    1.2.3 SAO Data Structures ................................. 21
  1.3 Processing Algorithm ...................................... 22
  1.4 Architecture Details ..................................... 25
    1.4.1 Input Generator ..................................... 25
    1.4.2 Pixel Producer ..................................... 26
    1.4.3 Pixel Processor ..................................... 27
    1.4.4 SRAM Module ....................................... 28
    1.4.5 Verification ........................................ 29
    1.4.6 Integration ......................................... 30
  1.5 Results and Analysis ..................................... 31

2 Chapter 2: HEVC Chip FPGA Demo Interface Implementation 35
  2.1 Introduction ............................................... 35
  2.2 FPGA’s and Xilinx VC707 Evaluation Board Overview 37
    2.2.1 Xilinx VC707 Evaluation Board and Xilinx Virtex 7 FPGA 37
  2.3 HEVC Chip Interfaces ....................................... 38
    2.3.1 HDMI Interface ..................................... 38
    2.3.2 SD Card Interface ................................... 41
2.3.3 Clock Generation and Domains .................................... 43
2.4 Input Generation and SD-Cabac Interface ............................. 44
  2.4.1 Bitstream Details .................................................. 45
  2.4.2 SD-CABAC Interface .............................................. 46
  2.4.3 Verification ......................................................... 51
2.5 Results ........................................................................... 53

3 Chapter 3: Conclusion ......................................................... 55
  3.1 Contribution ............................................................. 55
  3.2 Future Work ............................................................ 56
List of Figures

1-1 Comparison of HEVC and other compression standards [11]. HEVC achieves a higher signal to noise ratio compared to other standards with the same bit-rate. .................................................. 17

1-2 Block diagram of the HEVC decoding process [5]. Notice that the SAO filter output populates the decoded picture buffer, which is the step immediately before the display of video. ......................... 17

1-3 Subjective results of applying SAO filter [5]. Notice several artifacts that appear as 'ghost lines' disappear once the SAO algorithm is applied. ................................................................. 18

1-4 EO class 1-D patterns (horizontal, vertical, 45° diagonal and 135° diagonal). SAO is limited to only 4 possible patterns in order to reduce complexity of comparisons done. ........................................ 19

1-5 EO categories definition. SAO disallows sharpening; thus only positive offsets are applied for categories 1 and 2, and only negative offsets are applied for categories 3 and 4. ................................. 20

1-6 A CTU consists of the three color component (Y, Cr and Cb) CTBs put together. ................................................................. 21

1-7 Detail of offset of input pixel block for processing. Notice that the processed block is a shifted version of the input samples, using samples from the current input block as well as samples from the left and top neighboring input blocks. ........................................ 22
1-8 Detail of samples processed and samples stored in a given 4x4 input sample block. Notice that the samples stored in the small register file are also saved in the SRAM in case we are in the bottom block of the CTU being processed.

23

1-9 Frame is scanned through a horizontal raster scan and each CTB is scanned through a double vertical raster scan. This method allows for hardware storage elements to be reused while processing a complete frame, thus reducing space requirements.

24

1-10 Block diagram for the SAO block. Notice the 16 parallel pixel processors, key feature in guaranteeing a low latency in the processing of samples.

26

1-11 SAO software verification flowchart. The input and output of the SAO reference software model is extracted in order to verify the custom implementation of the SAO algorithm described in the paper.

30

2-1 Block Diagram for the HEVC Chip FPGA Demo. Notice we include the SAO filter presented in Chapter 1, while the SD Card and HDMI Interfaces are highlighted as well.

36

2-2 Physical layout of the VC707 Evaluation Board [17]. Note the HDMI output port marked by number 18, the SD Card port marker by number 5 and the DDR3 RAM marked by number 20.

36

2-3 Block diagram for the ADV7511 chip [2]. Highlighted are the signals provided by the HDMI interface module.

40

2-4 Input/Output diagram for the SD Card Interface. Signals on the right correspond directly to a pin on the SD Card, while signals on the left are used by the applications.

43

2-5 Example memory mapping for 2 different bitstreams loaded into an SD Card. The hexadecimal numbers on the left represent the address of the blocks that contain the specified data.

47
2-6 Block diagram for the SD-CABAC Interface. *DecodedBin* and *DecodedBin*\textsuperscript{2} are the outputs of the CABAC Engine that correspond to the decompressed bitstream data.

2-7 State Machine diagram corresponding to the reading of bitstream data from the SD Card interface and supplying it to the CABAC Engine.

2-8 State Machine diagram corresponding to the initialization and reset sequence for the CABAC Engine.

2-9 Block diagram corresponding to the verification system using the DDR3 RAM interface and UART. Notice we use a FIFO to 'pack' individual bit bins decoded from the CABAC Engine into 64-byte groups for improved performance.
List of Tables

1.1 Conditions for EO Category Classification ........................................... 20
1.2 Storage required for one SAO processing core ................................. 29

2.1 Timing parameters for 1080p 60Hz video output [6] ......................... 41
2.2 Clock domains for HEVC Chip FPGA Demo ................................. 44
2.3 State variables for CABAC Engine ................................................. 46
2.4 CABAC Test Bitstreams Parameters ................................................. 52
2.5 FPGA Utilization Percentages ......................................................... 53
Chapter 1

Chapter 1: Sample Adaptive Offset Filter Design and Architecture

1.1 Introduction

The emergence of the network as the bottleneck in the transmission of video content has accelerated the development of more advanced video compression codecs. High Efficiency Video Coding (HEVC), the most recent of these codecs, promises substantial performance improvements over H.264. Among these improvements are increased resolution, new loop filtering blocks and roughly double the compression at comparable picture quality. In turn, HEVC requires much more computational processing power than its predecessors, with a substantial 2x to 10x computational power requirement increase [11]. Such increase in computational power requirement has led to the development of various dedicated chips to streamline the decoding and encoding of HEVC video.

The Energy Efficient Integrated Circuits and Systems Group at MIT has developed an HEVC decoder chip. However, since the chip was completed, the standard was finalized with several changes, which make the existing chip incompatible with the finalized standard [14]. Some companies (Broadcom, Qualcomm, Ericsson), have developed chips that implement HEVC, but the majority have had limited exposure or are limited to trade shows or announcements. Overall, there is much work to be
done to demonstrate, verify and analyze the behavior of the HEVC standard.

As is the case with other video compression standards such as the currently popular H.264, HEVC is applied in a two way process: first raw video is compressed in order to be transmitted (encoding) and then it is decompressed (decoded) when the data has reached the target device for viewing. Among the innovations in HEVC is the addition of the Sample Adaptive Offset filter (SAO), a loop filtering block designed to smooth artifacts created by the aggressive compression applied by HEVC on the encoding side.

This chapter presents a processing algorithm and a hardware architecture for the implementation of the SAO filter as part of a dedicated HEVC decoder chip designed for low power environments. This chip is planned as a successor to an already existing HEVC decoder chip, which can decode up to 4Kx2K resolution video efficiently, consuming only 78mW of power [14]. Applications for a dedicated HEVC Chip are numerous - especially given modern trends towards on-the-go video consumption. One can imagine laptops, cellphones and dedicated streaming devices (such as an Apple TV or a Google Chromecast) using an HEVC Chip to efficiently decode a high-definition video stream.

Given this low power environment design constraint, the implementation described in this chapter aims to achieve high throughput and low latency, while maintaining a reasonable area use.

In this chapter, Section 1.2 describes the details of the SAO filter algorithm and Section 1.3 introduces the processing algorithm to be used in the hardware architecture described in Section 1.4. Finally, Section 1.5 presents Results and Analysis.

### 1.2 Sample Adaptive Offset Filter

HEVC employs more aggressive encoding schemes in order to achieve performance improvements over H.264 in terms of bit rate reduction. Compared to H.264, HEVC allows for transforms with size up to 32x32, while H.264 is limited at 8x8. Also, HEVC uses up to 8-tap interpolation for luma samples and 4-tap interpolation for chroma
Modern video coding standards try to remove as much redundancy from the coded representation of video as possible. One of the sources of redundancy is the temporal redundancy, i.e. similarity between the subsequent pictures in a video sequence. This type of redundancy is effectively removed by the motion prediction. Another type of redundancy is spatial redundancy and is removed by intra-prediction from the neighboring pixels and spatial transforms. In HEVC, both the motion prediction and transform coding are block-based. The size of motion predicted blocks varies from $8 \times 4$ and $4 \times 8$, to $64 \times 64$ luma samples, while block transforms and intra-predicted block size varies from $4 \times 4$ to $32 \times 32$ samples. These blocks are coded relatively independently from the neighboring blocks and approximate the original signal with some degree of similarity. Since coded blocks only approximate the original signal, the difference between the approximations may cause discontinuities at the prediction and transform block boundaries \[2, 5\]. These discontinuities are attenuated by the deblocking filter. A larger transform can also introduce more ringing artifacts that mainly come from quantization errors of transform coefficients \[22\]. HEVC uses 8-tap fractional luma sample interpolation and 4-tap fractional chroma sample interpolation, while H.264/AVC uses 6-tap and 2-tap for luma and chroma respectively. A higher number of interpolation taps can also lead to more ringing artifacts. These ringing artifacts are corrected by a new filter: Sample Adaptive Offset (SAO).

**Figure 1-1**: Comparison of HEVC and other compression standards [11]. HEVC achieves a higher signal to noise ratio compared to other standards with the same bit-rate.

**Figure 1-2**: Block diagram of the HEVC decoding process [5]. Notice that the SAO filter output populates the decoded picture buffer, which is the step immediately before the display of video.
samples, while H.264 is again limited to 6-tap and 2-tap interpolation respectively [5]. Due to these larger transforms and longer tap interpolations used by the HEVC encoder to reduce bit-rate, undesirable visual artifacts that arise in the decoding process can become more serious compared to previous video compression standards, including H.264. The SAO filter is designed to further reduce artifacts generated by the compression algorithms used by the HEVC encoder.

The SAO filter is added to the HEVC standard to be able to achieve low latency processing while also yielding effective filtering to deal with such encoding artifacts. It is the last step in the reconstruction (decoding) process, coming after the deblocking filter and performing the last filtering operation before the output is generated and can be displayed. This can be seen graphically in Figure 1-2. Specifically, SAO is aimed at reducing the mean sample distortion of a region of the video transmission. Using SAO there is an average reduction in bitrate of 2.3% (that can go up to 23.5% depending on the source video) with only a 2.5% increase in decoding time [5]. Subjective tests have shown that SAO significantly improves the visual quality by suppressing the ringing artifacts' [11], as can be seen in Figure 1-3.

![Figure 1-3: Subjective results of applying SAO filter [5]. Notice several artifacts that appear as 'ghost lines' disappear once the SAO algorithm is applied.](image)

The SAO filter works by applying specific offsets to samples in order to reduce
their distortion relative to other samples in the same video frame. It can do this offset in two different modes of operation, edge offset (EO) and band offset (BO). EO is used to reduce distortion and BO is used to correct for quantization errors and phase shifts.

### 1.2.1 Edge Offset

The Edge Offset mode compares the sample being processed to two neighboring samples, and then applies an offset based on such comparison. In order to comply with a low complexity requirement, SAO defines only four possible 1-D classes for comparison: horizontal, vertical, 45° diagonal and 135° diagonal. These can be seen in Figure 1-4. Once the samples are compared using one of the four classes, the sample is grouped into one of five categories (the categories themselves shown in Figure 1-5). The conditions for the EO categories are shown in Table 1.1. SAO only applies offsets in order to smooth the differences between samples, and thus it applies a positive offset to samples in categories 1 and 2 and a negative offset to samples in categories 3 and 4. Logically, if the samples are the same (category 0), no offset is applied. This preference for smoothing instead of sharpening allows for offsets to be transmitted as unsigned values, thus reducing space requirements.

![Figure 1-4: EO class 1-D patterns (horizontal, vertical, 45° diagonal and 135° diagonal). SAO is limited to only 4 possible patterns in order to reduce complexity of comparisons done.](image-url)

The SAO is designed to be a low latency and low complexity filter, so the calculation of the offsets themselves is left to the encoder, while the classification of the samples is left to the SAO block itself. Four offsets are transmitted by the encoder, each one corresponds to a particular category.
Figure 1-5: EO categories definition. SAO disallows sharpening; thus only positive offsets are applied for categories 1 and 2, and only negative offsets are applied for categories 3 and 4.

Table 1.1: Conditions for EO Category Classification

<table>
<thead>
<tr>
<th>Category</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( c = a = b )</td>
</tr>
<tr>
<td>1</td>
<td>((c &lt; a) \land (c &lt; b))</td>
</tr>
<tr>
<td>2</td>
<td>((c &lt; a) \land (c = b)) \lor ((c = a) \land (c &lt; b))</td>
</tr>
<tr>
<td>3</td>
<td>((c &gt; a) \land (c = b)) \lor ((c = a) \land (c &gt; b))</td>
</tr>
<tr>
<td>4</td>
<td>((c &gt; a) \land (c &gt; b))</td>
</tr>
</tbody>
</table>

1.2.2 Band Offset

The Band Offset mode applies an offset to all samples that fall within some 'band' of values. In this case, no comparison is performed with neighboring samples, instead only the absolute magnitude of the sample being processed is inspected. By default, there are 32 bands defined in SAO for an 8-bit sample, with each band being of size 8. Thus, the \( k \)th band corresponds to an absolute value of a sample of \( 8k \) to \( 8k + 7 \).

As is the case with EO, the calculation of the offsets themselves is left to the encoder. Furthermore, BO is limited to 4 consecutive bands for which offsets can be applied, in order to maintain a low complexity. This leverages on the fact that distortions present on several bands are more likely to be in consecutive bands. The encoder transmits four offsets, as is the case with EO, in order to reduce complexity.
1.2.3 SAO Data Structures

HEVC defines two main data structures, coding tree blocks (CTB’s) and coding tree units (CTU’s), in order to organize the processing of samples, which SAO follows as part of the standard. 24-bit pixel values are divided into a luma (Y) brightness component and chroma (Cr and Cb) color components. SAO processing is done separately (and possibly in parallel) for luma and chroma samples, as discussed in Section 1.4.6.

Within the complete frame, HEVC defines coding tree blocks (CTB’s), which are fixed sized sub-blocks (typically 64x64 pixels) for luma and chroma samples. All three CTB’s put together form a coding tree unit (CTU), as shown in Figure 1-6. SAO information (SAO mode, EO class, EO offsets, BO bands, BO offsets) is transmitted at a CTB level. This means that all samples in a specific CTB share the same SAO parameters. Furthermore, both chroma CTB’s share the same SAO parameters. This is done in order to minimize the amount of information transmitted by the encoder, and relies on the fact that neighboring pixels are likely to have similar distortion patterns. SAO also allows for CTB’s to merge SAO information with neighboring CTB’s, in order to further reduce information transmitted.

![Figure 1-6: A CTU consists of the three color component (Y, Cr and Cb) CTBs put together.](image)
1.3 Processing Algorithm

The challenge of performing SAO in hardware efficiently comes from the fact that current samples being processed depend on future samples in order to be able to decide what offsets to apply. This is due to the fact that CTB’s are processed in a raster scan order, so all the pixel data is not available to the processor at a single specific time. A naive solution of simply delaying the output until the necessary samples are obtained yields long latencies and a significant use of memory which is unsuitable for mobile and low power applications, where low latency and lightweight memory footprint is desired. To solve this, this section describes an algorithm which relies on the use of shifted input sample blocks, in order to be able to process sample blocks with minimal latency and reduced memory use.

In general the most basic processing unit of the SAO block are 4x4 sample blocks
(128 bits at 8 bits per sample), which was chosen to match the overall system memory architecture. To reduce the need to wait for future samples to initiate processing, the algorithm processes a shifted version of the input samples, as detailed in Figure 1-7. This is done to ensure that the data that is needed to process the current input is available at input time - since even within a single CTB the bordering pixels depend on neighbors to apply SAO appropriately and such neighbors are not available until a future time due to the before-mentioned raster scan scheme. Also, the remaining unprocessed samples resulting from the shift are stored and processed at a later time, as part of another input block, as seen in Figure 1-8.

![4x4 input sample block diagram]

Figure 1-8: Detail of samples processed and samples stored in a given 4x4 input sample block. Notice that the samples stored in the small register file are also saved in the SRAM in case we are in the bottom block of the CTU being processed.

Furthermore, the algorithm uses three different raster scan methods to reduce memory storage requirements. At a CTB level (each CTB is generally composed of 256 4x4 sample blocks for the standard 64x64 sample CTB size), the blocks are
processed in a double vertical raster scan. In other words, 4x4 sample blocks are processed in a vertical raster within an intermediate 16x16 sample block and these 16x16 sample blocks are also processed in a vertical raster scan within the complete CTB. At a frame level, each CTB is processed in a horizontal raster scan. This processing pattern can be seen graphically in Figure 1-9. It allows for small storage elements (such as register files) to be reused across CTB’s, without the need to access main memory.

Figure 1-9: Frame is scanned through a horizontal raster scan and each CTB is scanned through a double vertical raster scan. This method allows for hardware storage elements to be reused while processing a complete frame, thus reducing space requirements.

Due to the shifted processing order, the algorithm results in a single sample wide ‘edge’ at the right hand side and bottom side of the frame that has to be processed on its own to maintain data consistency. This is clearly not ideal, since it reduces throughput and creates the necessity for corner cases to deal with these ‘leftover’
samples. The solution is to pad the complete input frame with 'buffer' samples, effectively increasing the size of the frame by 4 pixels on each dimension. This allows for the processing to continue as normal and the 'edges' will only correspond to 'buffer' samples, so there is no necessity to use corner cases. It is only at the last state, when data is being read to be displayed that the corresponding module ignores the 'buffer samples'. This frame size adjustment has no effect on the architecture described in Section 1.4, apart from a negligible increase in SRAM storage size.

This algorithm allows for low latency processing and low memory storage requirements, at the cost of an added computational complexity represented by the logic needed to keep track of all the unprocessed samples and posterior reordering.

1.4 Architecture Details

The main architecture of the SAO processing block is divided into four main parts: the input generator, the pixel producer, the pixel processors and the SRAM module. A block diagram detailing their interconnection is shown in Figure 1-10. It is designed to be highly parallelized, therefore introducing as low latency as possible into the complete decoding process. This high parallelization also leads to high throughput, which can be traded for power savings using voltage scaling [4], which aligns nicely with our low-power design environment.

1.4.1 Input Generator

The Input Generator module serves as the primary interface to receive input from other parts in the HEVC data flow (namely the deblocking filter) and organize data to be processed by the Pixel Producer and Processor modules.

In particular, the Input Generator Module receives input in 16x16 sample blocks (2048 bit-wide bus) and organizes it into 4x4 sub-blocks to be given as input to the subsequent modules in the SAO block while also making sure that the timing requirements of such modules are met.
Figure 1-10: Block diagram for the SAO block. Notice the 16 parallel pixel processors, key feature in guaranteeing a low latency in the processing of samples.

### 1.4.2 Pixel Producer

The Pixel Producer module is designed as a control module that manages samples for the SAO block, performing three critical functions:

i. Interface with the Input Generator that supplies incoming stream of samples

ii. Store samples in order to perform processing algorithm

iii. Provide pixel processor modules with samples to process

The Pixel Producer module interfaces with the Input Generator modules using a simple FIFO scheme. The pixel producer signals when it can process new samples, and stalls the block if there are no new samples available for processing. In order to maintain the integrity of the data, it also stalls processing when the SRAM module is unavailable.

The Pixel Producer module uses a combination of the SRAM module and two register files to deal with the storage of samples necessary for correct processing.
This combination of storage elements is used in order to achieve a high throughput while maintaining area use as low as possible. One register file (small, 100 bits) is used to store samples and corresponding offsets for the bottom eight samples of the input block being processed. This register only needs to store one set of samples per sampled block due to the vertical raster scan scheme employed by the processing algorithm. Another register file (large, 198 bytes) is used to store the left eight samples of the input block being processed. (This can be seen graphically in Figure 1-7 and Figure 1-8). This large register needs to store samples corresponding to all 16 blocks in a CTU, again due to the vertical raster scan scheme employed by the processing algorithm. However, this larger register carries samples over CTU blocks, reducing the need to interface with memory and allowing for increased throughput, due to the horizontal raster scan scheme used at a CTU level by the processing algorithm. We also employ several other small register files that save some samples for a longer time to deal with special cases, such as the corner delay seen in Figure 1-7.

Finally, the Pixel Producer Module interfaces with the SRAM module to store the top samples for the offset block being processed across CTU blocks. This means that the interface is only active when the blocks at the top of a new CTU blocks are being processed.

If samples are available for processing, the pixel producer module is able to route new samples for processing to the pixel processor module with a maximum latency of one clock cycle (while also storing the necessary information in the described register files). The only exception to this scenario occurs when memory access is required (when processing a block at the top of a new CTU), in which case the latency would rise to a maximum of $\text{max\_latency} = \text{cycle\_time} + \text{memory\_access\_delay}$ on this proposed architecture. During testing, this latency usually resulted in 2 full clock cycles, so overall the process remains low latency.

### 1.4.3 Pixel Processor

The Pixel Processor module is dedicated to carrying out the SAO algorithm itself, as described in Section 1.2, using the samples provided by the Pixel Producer module.
In the case that edge offset is being used, the SAO classification is done efficiently in a combinatorial manner (as described in [5]).

First define $\text{category\_array} = \{1, 2, 0, 3, 4\}$ and $\text{sign}(x) = (x > 0) \ ? 1 \ : ((x == 0) \ ? 0 \ : -1)$. Furthermore, $c$ is the sample being processed and $a$ and $b$ are the neighboring samples. Then:

- $\text{sign\_left} = \text{sign}(c - a)$
- $\text{sign\_right} = \text{sign}(c - b)$
- $\text{edge\_id} = 2 + \text{sign\_left} + \text{sign\_right}$

Using these values, the category is given by $\text{category} = \text{category\_array}[\text{edge\_id}]$.

In the case that band offset is being used, the block checks whether samples are set in the specified bands in order to determine whether to apply an offset or not. This is also done with combinatorial logic by checking the five most significant bits of each sample.

Using both techniques allows the Pixel Processor module to have a constant latency of one clock cycle. In this implementation, 16 pixel processors are placed in parallel, in order to be able to process a complete sample block (16 samples) in one clock cycle. However, notice that due to the independence of each processor from each other, they can be easily reconfigured and used in other settings.

### 1.4.4 SRAM Module

As described above, the SRAM module is used to store the bottom samples needed for processing when changing CTU blocks. In this implementation, designed for 1080p video (1920 x 1080 pixels), the SRAM is sized at 6KBytes. In more general terms, the size of the SRAM is the only part of the architecture of the SAO processing block that depends on the target frame size for processing, which allows for high configurability of the design. However, in the current implementation, the size of the SRAM has to be specified at synthesis, and thus should be set to correspond to the maximum frame size allowed for processing.
Table 1.2 presents a summary of the space requirement for a single SAO processing ‘core’. More specifically, it presents the space requirement for a luma processing core, since chroma samples are downsized by half (i.e. 4 bits per sample as opposed to 8), which reduces the space requirement to roughly 4.4 KBytes.

Table 1.2: Storage required for one SAO processing core

<table>
<thead>
<tr>
<th>Structure</th>
<th>Space (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>6000</td>
</tr>
<tr>
<td>Big Register File</td>
<td>198</td>
</tr>
<tr>
<td>Small Register File</td>
<td>12.5</td>
</tr>
<tr>
<td>Misc. Other Registers</td>
<td>28.75</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>6239.25</strong></td>
</tr>
</tbody>
</table>

1.4.5 Verification

One of the biggest challenges of implementing the SAO block is to correctly verify its behavior. In order to do this, several steps were taken.

First, a custom Python software model of the SAO filter was developed, and this one was verified against the reference software provided by the HEVC development task force (the JCT-VC). This is done by modifying the reference code to extract the input and output data to the SAO module inside it. This data is then used as input data to run the created software model and generate output data comparable to the one generated by the reference code. By comparing these two results, the correctness of the implementation of the SAO filter can be determined. This process is shown in Figure 1-11.

Once the software model is appropriately verified, a similar procedure is used to verify the hardware model. The software model is used to generate appropriate input and reference data to run the hardware simulations. Notice we use the software model to generate the test vectors and not the reference code, due to the fact that the software model allows us to customize the test vectors themselves (namely their processing pattern and size), whereas the reference code is much more rigid in this respect. These tests vectors are guaranteed to be correct thanks to the software
verification. In a similar light to the process described for the software simulations, the hardware model is run using the new test vectors, and the output is compared to the reference output to verify that the hardware is doing the processing as expected.

Figure 1-11: SAO software verification flowchart. The input and output of the SAO reference software model is extracted in order to verify the custom implementation of the SAO algorithm described in the paper.

1.4.6 Integration

A significant challenge is the full integration of the SAO processing block into the full HEVC decoder chip under development. This stems from the fact that there are numerous options to be addressed, in particular: the degree of parallelization of the SAO block itself and the rearrangement of the samples processed by the SAO block.

The design detailed in the previous sections allows for a high degree of customization with regards to the integration into the complete HEVC Chip pipeline. The SAO block can be implemented to process the luma and chroma samples sequentially in the complete chip pipeline, reducing the degree of parallelization and throughput but saving area by only having one SAO 'core'. Another option is to process the luma
and chroma samples in parallel, by having three SAO ‘cores’ and thus increasing throughput. This method is desirable because it allows for a significant reduction in power consumption through voltage scaling [4], at the expense of area use. In particular, high parallelism leads to high throughput, which allows for source voltages to be reduced and by extension power consumption is also reduced. The trade-off in this choice is the added amount of area consumed, but this seems like a secondary concern due to the small size of the SAO processing block itself, as will be seen in Section 1.5.

The high throughput achieved by each SAO processing core guarantees that both methods would allow for real time decoding and thus remain realistic options.

Going one step further, the modular design of the SAO block also open up possibilities with respect to the size of input blocks and the degree of parallelization in the design. In particular, as mentioned before, the individual Pixel Processor modules can operate individually from each other, and could potentially be even integrated into a separate stage in a HEVC pipeline. Also, the module could accept 4x4 sample input blocks directly as opposed to 16x16 sample input blocks, if that was required.

The rearrangement of the output samples generated by the SAO block (which are themselves offset due to the offset used in the processing algorithm) is resolved by using the ‘buffer’ areas in the input frame as described in Section 1.3.

1.5 Results and Analysis

Broadly speaking, results support the design choices made in order to achieve low latency, high parallelization, high customizability and reasonable area use for the implementation of the SAO block.

Area wise, a complete SAO processing core is estimated to be made of 36.5 kgates (where a gate is a unit of area that equals the area of a standard 2-input NAND gate). The majority of this is composed by the Pixel Producer Module (32.6%, or roughly 11.9 kgates) and the Input Generator Module (32%, or roughly 11.7 kgates), while each Pixel Processor module contributes 0.80 kgates (2.1%). Also, the SRAM,
as mentioned before, is sized at 6 KBytes. These numbers represent a reduced gate count from similar implementations [18]. Also, compared to a full implementation of an HEVC decoder chip (albeit one without an SAO block) [14], the SAO block would represent merely 3% of the total gate count for the chip, and 9% of the total SRAM storage available.

Performance wise, the SAO block can process 16 samples per clock cycle in steady state (that is, in the case where no memory access is required). This yields a best case latency estimate of 10 ns per 16 processed samples, using a standard 100Mhz clock. The worst case scenario occurs in cases where memory access is required, in which case the latency is bounded by $max\_latency = cycle\_time + memory\_access\_delay$ as described above.

With regards to throughput, using the best case latency estimate (assuming continuous availability of samples to process, no memory access and a 100Mhz clock) yields a steady state throughput of 200 MBytes per second processing luma and chroma in parallel or 133 MBytes per second processing luma and chroma sequentially, both of which are enough to supply 4K video at 120 frames per second in real time (which requires roughly 8 MBytes per second of constant throughput) and more than enough to supply our target 1080p HD video at 60 frames per second. This results are comparable to similar implementations [18]. Notice that this performance is independent of the source data, since all samples are processed in the same manner. Since memory access is only required for blocks that are at the top of a new CTB being processed, the memory interface is only active for roughly 6% of blocks processed each frame. This helps reduce power and guarantee a low latency in most cases. Furthermore, such high throughput can increase the amount the idle time the SAO block will find itself in, which coupled with techniques such as powergating in the overall HEVC chip (and voltage scaling as already discussed before) would result in even more power savings.

Finally, as described above, the design and architecture of the SAO block allow for it to be integrated into a full HEVC with a high degree of customizability and portability. The modular design presented allows for variations in not only data input
patterns and frame size, but also even in the degree of processing parallelization.
Chapter 2

Chapter 2: HEVC Chip FPGA Demo Interface Implementation

2.1 Introduction

As mentioned in Chapter 1, there is a significant amount of testing and verification to be done both relating to the HEVC standard and the HEVC chip under development. Another step in this process is the creation of a complete FPGA demo for the HEVC Chip under development itself - a demo which aims to provide a real-life verification test, by decoding HEVC encoded video and displaying it on an HD screen.

This demo clearly entails the complete synthesis of the HEVC Chip onto an FPGA but, also critically, necessitates the creation of interfaces to permit the input of data to the ported HEVC Chip and the ability to output pixel data to drive an HD display. A block diagram of the demo is presented in Figure 2-1.

This chapter presents the design and implementation of such interfaces for a Xilinx VC707 Evaluation Board, which can be seen physically on Figure 2-2. First, an output HDMI interface is described, which can drive a display at 1080p resolution with a 60Hz refresh rate. Second, a data input system based on an SD Card is also detailed. This system is responsible for the supply of bitstream data to the HEVC chip itself. Together, these systems allow the HEVC Chip to acquire the bitstream data necessary to decode HEVC video and display it in an HD monitor.
Figure 2-1: Block Diagram for the HEVC Chip FPGA Demo. Notice we include the SAO filter presented in Chapter 1, while the SD Card and HDMI Interfaces are highlighted as well.

Figure 2-2: Physical layout of the VC707 Evaluation Board [17]. Note the HDMI output port marked by number 18, the SD Card port marker by number 5 and the DDR3 RAM marked by number 20.
In this chapter, Section 2.2 introduces FPGA’s and the Xilinx VC707 Evaluation Board while Section 2.3 describes the implementation details of the HDMI and SD Card interfaces. Finally, Section 2.4 describes the data input system architecture, functionality and verification technique.

2.2 FPGA’s and Xilinx VC707 Evaluation Board Overview

A Field-Programmable Gate Array (FPGA) is an integrated circuit that can be re-programmed on an arbitrary basis. It contains a large number of configurable logic blocks, which via the use of lookup tables and flip-flops, among other elements, can be configured to perform an arbitrary logic function.

FPGA’s are useful because they present a powerful and configurable interface that can also be reconfigured on an on-demand basis (as opposed to a custom made IC that has to be manufactured and is then unmodifiable). For the purposes of the HEVC Chip FPGA Demo, using an FPGA allows for rapid iterations for testing while making minimal compromises in performance.

In this section, we describe the characteristics of the FPGA used for the HEVC Chip FPGA Demo.

2.2.1 Xilinx VC707 Evaluation Board and Xilinx Virtex 7 FPGA

The FPGA used in this demo is the Xilinx Virtex 7 which is part of the Xilinx VC707 Evaluation Board. The Virtex 7 FPGA can be seen physically in Figure 2-2 marked by number 1. The VC707 Evaluation Board is particularly well suited for the HEVC Chip demo for several reasons.

First, the VC707 board has a wide array of available interfaces for communication, in particular an HDMI driver chip and an SD Card port - critical aspects for the interfaces described in this Chapter. The implemented HDMI and SD Card interfaces themselves are described in more detail in Section 2.3. These interfaces are marked
by numbers 18 and 5 in Figure 2-2.

Second, the VC707 board also has a DDR3 RAM interface (up to 1GB of storage by default [17]), which is a convenient way to provide the HEVC with large-scale storage. In particular, the HEVC chip could substitute its chip-specific storage SRAM and eDRAM modules with access to DDR3 RAM. Another use of the DDR3 RAM interface is for the storage of intermediate data that can be used for verification purposes, as is described in Section 2.4.3. The DDR3 RAM can be seen physically in Figure 2-2 marked by number 20.

Third, the Virtex 7 FPGA present in the VC707 board has enough space to support a fully synthesized version of the chip while also allowing for the possibility of using block RAM’s (BRAM’s) to emulate the before-mentioned chip-specific storage structures.

Fourth, the VC707 board allows for both high speed operation (with a maximum clock frequency of 200Mhz [17]) and a wide degree of clock domain variability. In other words, through PLL’s and user-defined clocks, the VC707 board permits a wide range of clock domains to operate, which adapts nicely to the variable clock domain requirements of the demo, as is described in more detail in Section 2.3.

### 2.3 HEVC Chip Interfaces

As mentioned before, the real-life FPGA demo consists of the use of an SD Card to provide an HEVC encoded bitstream to the HEVC Chip, which in turn decodes the bitstream to generate HD video, which is finally displayed in an HD monitor. An overview of the demo is presented in Figure 2-1.

In this section, the HDMI and SD Card interfaces that are needed for the flow of data to and from the HEVC Chip are presented.

#### 2.3.1 HDMI Interface

High-Definition Multimedia Interface (HDMI) is digital video interface that is designed to transmit uncompressed HD video data to a device capable of displaying it.
Since its creation, HDMI has served as the replacement for older analog video transmission protocols. Its HD video display capabilities and compatibility and portability make HDMI the ideal protocol to use for the HEVC Chip FPGA Demo.

The HDMI interface makes use of the Analog Devices ADV7511 chip present in the VC707 Evaluation Board. After initialization, the ADV7511 chip converts standard VGA video signals into HDMI control signals. For our application, this means that the HDMI Interface module has to first initialize the ADV7511 chip and then for further operation has to provide the ADV7511 chip with several VGA control signals. In order to initialize the ADV7511 chip in the VC707 Evaluation Board, we use code provided by the Energy-Efficient Multimedia Systems Group at MIT [7]. This initialization process activates the HDMI output and sets flags in the hardware registers of the ADV7511 chip (setting numerous things such as aspect ratio and input color space, among others). After initialization, the HDMI Interface module generates VGA control signals to actively drive the chip, as is described next.

The VGA protocol works by using a pixel clock, which on every cycle presents a new set of pixel color data to be displayed. It also uses to synchronization signals ($h_{sync}$ for horizontal sync and $v_{sync}$ for vertical sync) that dictate the end of a horizontal line and a vertical line on the display, respectively.

The ADV7511 chip itself requires the pixel clock, $v_{sync}$, $h_{sync}$, data enable ($de$) and the pixel color data as control signals. These signals are highlighted in the block diagram for the ADV7511 chip presented in Figure 2-3.

Furthermore, the ADV7511 chip can handle multiple input color spaces. In our current implementation, we use the RGB 4:4:4 color space. This means that of the 36 bit wide input pixel data signal, 12 bits are assigned per color value - that is 12 bits for the red component, 12 bits for the blue component and 12 bits for the green component of the pixel. Another common option available is the YCrCb 4:2:2 space, which assigns 12 bits to the luma component (Y) and 6 bits each to the chroma components (Cr and Cb).

The VGA timing constants used to drive the ADV7511 chip are presented in Table 2.1. The $de$ signal is generated using both the horizontal and vertical blank signals,
Figure 2-3: Block diagram for the ADV7511 chip [2]. Highlighted are the signals provided by the HDMI interface module.
using the following expression: $de = !h\_blank && !v\_blank$, where $h\_blank$ is true when the horizontal pixel count is between $h\_sync\ count$ and $(h\_sync\ count) - (h\_sync\ width)$ while $v\_blank$ is true when the vertical pixel count is between $v\_sync\ count$ and $(v\_sync\ count) - (v\_sync\ width)$. The HDMI Interface is based upon the VGA interface developed in MIT’s Digital Systems Laboratory course - heavily modified to accommodate the target HD resolution and incorporate the blank and de signals that the ADV7511 chip requires.

Table 2.1: Timing parameters for 1080p 60Hz video output [6]

<table>
<thead>
<tr>
<th>Pixel clock frequency</th>
<th>148.5Mhz</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_sync\ count$</td>
<td>2200 pixel clocks</td>
</tr>
<tr>
<td>$h_sync\ width$</td>
<td>44 pixel clocks</td>
</tr>
<tr>
<td>$v_sync\ count$</td>
<td>1125 pixel clocks</td>
</tr>
<tr>
<td>$v_sync\ width$</td>
<td>5 pixel clocks</td>
</tr>
</tbody>
</table>

We generate the pixel clock using a Xilinx PLL primitive, a process which is described in more detail in Section 2.3.3.

2.3.2 SD Card Interface

The SD Card standard is a standard for the use of small, non-volatile, flash-based, removable memory storage units (the SD Cards themselves). SD Cards are very common in multi-media applications, due to their relatively high throughput, low cost, high capacity and low power consumption. Standard SD Cards can hold up to 32GB of data (with expanded capacity cards (SDXC) going even further), while typically drawing no more than 100mA of current during routine operation [8]. All these factors make the SD Card an ideal medium for the transfer of encoded bitstream data to the HEVC Chip in the FPGA Demo.

The SD Card protocol describes 3 modes of operation for an SD Card: SD 1-bit mode, SD 4-bit mode and SPI mode. For the purposes of the demo, we use the third mode. As its name denotes, the third mode uses the standard Serial Peripheral Interface (SPI) serial bus protocol, which allows for portability and compatibility. While the SPI protocol does not allow for the use of all the features available from
the SD Card protocol (namely 4 bit parallel CRC, among others), it has enough functionality for our purposes. The SPI protocol utilizes 4 one-bit signals for data transfer and control: CLK, MOSI (Master Out Slave In), MISO (Master In Slave Out) and CS (Chip Select). The MOSI signal is used to send commands to the SD Card while the MISO signal provides the responses from the SD Card. The CS signal is using during initialization.

We used a VHDL open source module [15] to implement the SPI mode of the SD Card standard and created a custom-made Verilog wrapper module to integrate it to the VC707 Evaluation Board and the other modules described in this chapter. Figure 2-4 presents the input/output characteristics of the implemented SD Card Interface. Of particular interest are the handshake signals (\textit{hndShk\_i} and \textit{hndShk\_o}), which allow for fine-tuned one-byte-at-a-time control while reading from the SD Card. In particular, \textit{hndShk\_o} goes high when new a new byte is available for 'consumption'. If the application wishes to read that byte, it sets \textit{hndShk\_i} high, at which point the data becomes available in the \textit{data\_out} port and \textit{hndShk\_o} goes low until a new byte is available for 'consumption'.

Other control signals of interest are \textit{rd}, \textit{address} and \textit{continue}. \textit{rd} is a flag that is set to 1 when the application desires to read content from the SD Card. \textit{address} specifies the address of the \textit{block} that the application desires to read. Notice that SD Cards use \textit{block addressing} as opposed to absolute addressing, so when presented with the address to the start of a block, the SD Card interface will read the complete block. Since blocks are sized at 512 bytes, addresses are, in effect, 512 byte offsets. Along the same lines, when the \textit{continue} flag is set high, the SD Card Interface automatically increments the block address when it has finished reading a block, in order to read a continuous stream of blocks without having to wait for the application to set the address. This behavior is utilized by the data input system described in Section 2.4.

Finally, we drive the SD Card Interface with a 100Mhz clock, but the maximum clock frequency allowed to drive an SD Card is capped at 12.5Mhz. This means that the maximum steady state throughput given by the SPI protocol is given by $\text{max\_throughput} = 12.5 \text{ Mbits/s} \approx 1.5 \text{ Mbytes/s}$. The generation of the clocks
needed to drive the SD Card Interface is discussed in Section 2.3.3.

![Diagram of SD Card Interface](image)

Figure 2-4: Input/Output diagram for the SD Card Interface. Signals on the right correspond directly to a pin on the SD Card, while signals on the left are used by the applications.

### 2.3.3 Clock Generation and Domains

The VC707 Evaluation Board supplies a 200Mhz clock differential pair as its system clock. Based on this signal, we generate a number of clocks to drive the numerous modules and interfaces needed in the demo.

First, we drive the DDR3 RAM interface using the system 200Mhz clock differential pair, which in turn outputs a single 200Mhz clock via the use of a "Differential Signaling Dedicated Input Clock Buffer" (available as a Xilinx primitive as "BUFGDS"). This 200Mhz clock is used as input for various "Mixed Mode Clock Managers" (also available as a Xilinx primitive as "MMCM"). These MMCM’s use phase locked loops (PLL’s) to allow for the generation of specific clock frequencies, determined by the following relation:

\[ f_{out} = \left( \frac{M}{D} \right) \times f_{in} \]
The use of MMCM’s and thus PLL’s effectively slows down the base 200Mhz clock to generate a high quality and low jitter video clock source. In fact, the use of MMCM’s helps compensate for the clock network delay present in the FPGA [1][16].

For our application, we use two MMCM’s. One is used to generate the HDMI pixel clock with a frequency of 148.5Mhz, by setting \( M = 5.94 \) and \( D = 8 \). The other is used to generate the clock used for the SD Card Interface and other modules with a frequency of 100Mhz, by setting \( M = 3.0 \) and \( D = 6 \).

The 100Mhz clock is used by the vast majority of modules present in the demo, including the SD Card interface described in Section 2.3.2, the SD-Cabac interface and the CABAC Engine, both described in Section 2.4. Finally, the SD Card interface itself generates a 12.5Mhz clock via a simple divider for direct interaction with the SD Card. This clock is used exclusively for the interaction with the physical SD Card and has no interaction with exterior modules, thus is not generated via an MMCM. These clock domains are summarized in Table 2.2.

<table>
<thead>
<tr>
<th>Clock Frequency</th>
<th>Dependent Modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>200Mhz</td>
<td>Top-level system, DDR3 RAM, UART</td>
</tr>
<tr>
<td>148.5Mhz</td>
<td>HDMI Interface</td>
</tr>
<tr>
<td>100MHz</td>
<td>SD-CABAC Interface, SD Card Interface, CABAC Engine</td>
</tr>
<tr>
<td>12.5Mhz</td>
<td>SD Card</td>
</tr>
</tbody>
</table>

The clock domains describe in this section use 29% of the Virtex 7 MMCM resources, 7% of its PLL resources, as well as 19% of its buffering resources.

2.4 Input Generation and SD-Cabac Interface

This section details the characteristics of the bitstream data provided to the HEVC Chip and the characteristics of the SD - CABAC Interface, which effectively provides the link between the bitstream data stored in the SD Card and the HEVC Chip and processing pipeline itself.

The CABAC Engine (context-adaptive binary arithmetic coding) is a unit within the HEVC chip. It is responsible for interpreting compressed input data (in this
case the bitstream data coming from the SD Card) and decompressing it into an appropriate pattern in order to be decoded by the HEVC chip into HD video. It uses a form of lossless variable length compression, which allows for the efficient transmission of encoded HEVC video. In particular, it achieves a 9% improvement in compression versus the encoding used in H.264 [12]. For the purposes of the SD-CABAC Interface and the HEVC Chip FPGA Demo itself, the CABAC Engine is the point of contact where data is supplied and processing is initiated. This interaction is explored in more detail Section 2.4.2.

### 2.4.1 Bitstream Details

The bitstream data used to drive the HEVC Chip consists of HEVC encoded video generated by the reference HEVC software provided by the JCT-VC. This bitstream data is read by the CABAC module of the HEVC Chip which then starts the complete decoding process.

The bitstream data consists of two parts: the state data and then the bitstream data itself.

The state data is composed of 35 syntax elements [3], which are presented in Table 2.3. In general, the syntax elements describe some features of the bitstream being decoded, such as the resolution of the video encoded. They are used to set up the CABAC Engine and customize its behavior with relation to the incoming bitstream. Notice we have included a custom variable, `total_bytes_bitstream`, that is not used by the CABAC Engine itself but by the SD-CABAC interface to determine how far to read along the SD Card. This is explored in more detail in Section 2.4.2.

The bitstream data is an arbitrarily long list of 32 bit values, since it depends on the length and specific details of the scene that has been encoded. To this end, there is no limitation (other than the physical storage capability of the SD Card itself) to how big of a bitstream can be provided to the CABAC Engine and thus the HEVC Chip itself.

In general, the state data and bitstream data are loaded directly onto a custom memory-mapped SD Card using a purpose-made Python program. Figure 2-5 shows
Table 2.3: State variables for CABAC Engine

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Bits</th>
<th>Syntax Element</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>SignDataHidingEnabledFlag</td>
<td>1</td>
<td>SliceType</td>
<td>2</td>
</tr>
<tr>
<td>CabacInitFlag</td>
<td>1</td>
<td>Qp</td>
<td>7</td>
</tr>
<tr>
<td>SliceSaoLumaFlag</td>
<td>1</td>
<td>SliceSaoChromaFlag</td>
<td>1</td>
</tr>
<tr>
<td>MinCbLog2SizeY</td>
<td>3</td>
<td>PicWidthInLumaSamples</td>
<td>14</td>
</tr>
<tr>
<td>PicHeightInLumaSamples</td>
<td>13</td>
<td>TransquantBypassEnabledFlag</td>
<td>1</td>
</tr>
<tr>
<td>PcmEnabledFlag</td>
<td>1</td>
<td>Log2MinIpcmCbSizeY</td>
<td>3</td>
</tr>
<tr>
<td>Log2MaxIpcmCbSizeY</td>
<td>3</td>
<td>AmpEnabledFlag</td>
<td>1</td>
</tr>
<tr>
<td>MaxNumMergeCand</td>
<td>3</td>
<td>NumRefIdxL0ActiveMinus1</td>
<td>4</td>
</tr>
<tr>
<td>NumRefIdxL1ActiveMinus1</td>
<td>4</td>
<td>MvdL1ZeroFlag</td>
<td>1</td>
</tr>
<tr>
<td>Log2MaxTrafoSize</td>
<td>3</td>
<td>Log2MinTrafoSize</td>
<td>3</td>
</tr>
<tr>
<td>CuQpDeltaEnabledFlag</td>
<td>1</td>
<td>TransformSkipEnabledFlag</td>
<td>1</td>
</tr>
<tr>
<td>DependentSliceSegmentFlag</td>
<td>1</td>
<td>SliceSegmentAddress</td>
<td>18</td>
</tr>
<tr>
<td>SliceAddrRs</td>
<td>18</td>
<td>CtbLog2SizeY</td>
<td>3</td>
</tr>
<tr>
<td>PicWidthInCtbsY</td>
<td>14</td>
<td>PicHeightInCtbsY</td>
<td>13</td>
</tr>
<tr>
<td>BitDepthY</td>
<td>4</td>
<td>BitDepthC</td>
<td>4</td>
</tr>
</tbody>
</table>
| a
| 3 | b
| c
| 3 | 1 |
| Log2MinCuQpDeltaSize               | 3    | c
| TilesEnabledFlag                   | 1    | total_bytes_bitstream              | 32   |

how the data is mapped to specific blocks and addresses in the SD Card. The state data is mapped to a single block, with each syntax element allocated 16 bits of space (except total_bytes_bitstream, which is allocated 32 bits), while the bitstream data occupies a subsequent arbitrary amount of blocks, with each value allocated 32 bits of space. In practice, this mapping allows several bitstreams and their corresponding state to be loaded into a single SD Card, provided that the SD - CABAC Interface adjusts the address it fetches the information from (and, of course, assuming the SD Card has enough physical space). During testing, as many as 4 different sets of bitstreams were put together on a single 2GB SD Card.

2.4.2 SD - CABAC Interface

The SD - CABAC interface is built in order to extract the data from the SD Card and feed it in a processing order compatible with the HEVC decoder chip - directly

\(^{1a}\): MaxTransformHierarchyDepthInter, \(^{b}\): MaxTransformHierarchyDepthIntra, \(^{c}\): EntropyCodingSyncEnabledFlag
Figure 2-5: Example memory mapping for 2 different bitstreams loaded into an SD Card. The hexadecimal numbers on the left represent the address of the blocks that contain the specified data.
into the CABAC engine as outlined before.

In particular, the SD - CABAC interface communicates with the SD Card using the handshake protocol implemented in the SD Card interface described in Section 2.3.2 to read the state and bitstream data. Figure 2-6 presents a block diagram of the SD Card - CABAC interaction.

![Block Diagram](image)

Figure 2-6: Block diagram for the SD - CABAC Interface. DecodedBin and DecodedBin2 are the outputs of the CABAC Engine that correspond to the decompressed bitstream data.

Of particular interest is the SD-CABAC Linker module, which regulates the interactions between the CABAC Engine and the SD Card Interface itself. It is responsible for reading the state variables from the SD Card, reorganizing the data obtained from the SD Card in 8 bit groups into the 64 bit format the CABAC Engine expects and finally for controlling the initialization and reset sequence for the CABAC engine. Figures 2-7 and 2-8 present the state machine diagrams for the supply of bitstream data to the CABAC engine and for the reset and initialization sequence.
Figure 2-7: State Machine diagram corresponding to the reading of bitstream data from the SD Card interface and supplying it to the CABAC Engine.

Figure 2-8: State Machine diagram corresponding to the initialization and reset sequence for the CABAC Engine.
The CABAC Engine expects bitstream data to come as 64 bit wide signals (we will refer to this signal as \textit{memData}, as is shown in Figure 2-6). We use a handshake scheme similar to the one described for the SD Card Interface to control the flow of data between the SD - CABAC linker and the CABAC Engine itself. The \textit{memReady} signal goes high when there is new data available in \textit{memData}, and the CABAC engine sets \textit{memDataConsumed} high when it has used the data available. At that point \textit{memReady} goes low until there is a new 64 bit piece of data available. Since the SD Card serves data only one byte at a time, we use a simple state machine to control the reading from the SD Card, shown in Figure 2-7. The SD Card is allowed to rest idle until the CABAC reset signal is asserted, at which point the SD - CABAC linker prompts the SD Card for 8 bytes of data. When the data is acquired, \textit{memReady} is asserted and the module waits until the CABAC engine signals it has used the data, by setting \textit{memDataConsumed} high. At this point, the SD - CABAC linker goes back to requesting data from the SD Card. This process is repeated until we have read all the bytes that correspond to the bitstream, which we infer by checking the \textit{total\_bytes\_bitstream} custom variable we inserted into the state data, as mentioned in Section 2.4.1.

The reset and initialization sequence for the CABAC engine is two-tiered. First, the \textit{cabac\_reset} signal must be asserted to reset the state of the whole CABAC engine, and after some time the \textit{cabac\_init} and \textit{cabac\_start} signals must asserted to actually start the processing of data. They key aspect in this process is that before the CABAC Engine is started, it requires us to 'prefetch' the first 4 64-bit bitstream data samples in order to ensure correct operation. Thus, as denoted by the state machine in Figure 2-8, we wait until we read 32 bytes from the SD Card before asserting \textit{cabac\_init} and \textit{cabac\_start}. After this, the reset signals are held low until the processing of the complete bitstream is complete and a restart is signaled by a reset to the SD - CABAC Linker itself.

Outside of these two features, the SD-CABAC linker initially reads the syntax elements from the SD Card and sets them. Since these values do not change, there is no need to revise them during the operation of the CABAC Engine.
2.4.3 Verification

As described in Chapter 1, verification is often one of the most significant challenges in the design process for a module. The modules described in this chapter present an extra level in verification complexity, in the sense that they involve a real world component in the use of an SD Card.

We used a two step process to verify the functionality of the SD Card interface and the SD-Cabac Linker. The first step was through simulation, while the second step was done on the actual VC707 Board.

The first step done was to verify that the already existing CABAC Engine module is functioning properly. To this end, in a similar manner to the approach used in Chapter 1, we generated several 'control' data streams using the JCT-VC reference software, which correspond to the expected output from the CABAC after decoding specific bitstreams. Once these 'control' bitstreams are available, we verified the functionality of the CABAC engine in simulation by feeding in the bitstream data and comparing the output to the expected correct 'control' bitstream.

The second step is to verify that the behavior exhibited in simulation is replicated on a real-life FPGA scenario, in particular since there is really no way of replicating in simulation the use of a physical medium (SD Card) to provide the bitstream data. Another complication arises due to the fact that the bitstream provided to the CABAC engine is compressed and naturally the 'control' bitstream mimicking the output is of a much larger size. Table 2.4 presents the sizes for several control sequences. Due to this, it is unfeasible to use the exact same verification approach used in simulation, since verification would require either reading the bitstream and the control sequence from the SD Card at the same time (not possible) or 'prefetching' and storing the control sequence in the FPGA itself, which is inconvenient due to the size of the control sequence.

To resolve this situation, we use a two tiered approach to verification on the FPGA. First, as a 'first order approximation', we don’t actually use the control sequence to verify correctness on the FPGA. Instead, we count the number of 1’s that result
Table 2.4: CABAC Test Bitstreams Parameters

<table>
<thead>
<tr>
<th>Bitstream</th>
<th>Bitstream Size</th>
<th>Control Size</th>
<th>Total Decoded Bins</th>
<th>Number of 1’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>005</td>
<td>10072 bytes</td>
<td>13120.125 bytes</td>
<td>104961</td>
<td>46499</td>
</tr>
<tr>
<td>045</td>
<td>17332 bytes</td>
<td>24645.625 bytes</td>
<td>197165</td>
<td>81281</td>
</tr>
<tr>
<td>120</td>
<td>8784 bytes</td>
<td>12905 bytes</td>
<td>103240</td>
<td>41000</td>
</tr>
<tr>
<td>231</td>
<td>60304 bytes</td>
<td>89702 bytes</td>
<td>717616</td>
<td>258853</td>
</tr>
</tbody>
</table>

when running the CABAC Engine in simulation while decompressing a bitstream (this number is also present in Table 2.4) and verified whether this number of 1’s is the same when running the system on the FPGA. Clearly, this test offers less guarantees than a full comparison to the expected output, although it is statistically highly unlikely to get a false positive result in this test. However, this approach worked well as a debugging tool and in practice always agreed with the second approach described next.

Figure 2-9: Block diagram corresponding to the verification system using the DDR3 RAM interface and UART. Notice we use a FIFO to ‘pack’ individual bit bins decoded from the CABAC Engine into 64-byte groups for improved performance.

The second approach required the use of the DDR3 RAM interface and UART
to perform a full comparison of the CABAC output to the expected 'control output'. In particular, we write the output of the CABAC Engine to the DDR3 RAM in the VC707 Evaluation Board. After the processing is done, we read the contents on the DDR3 RAM and send them to the host computer using UART. Once all the data is in the host computer, we compared it to the expected 'control' bitstream there. Figure 2-9 shows a block diagram for this verification system. This approach introduced more overhead and complexity, but is more formal and robust. In practice, both methods agreed.

2.5 Results

In general, the interfaces and architectures described in this Chapter achieve their specified goals - namely providing the HEVC Chip the ability to obtain data and display decoded pixels on an HD Display within the context of the VC707 Evaluation Board.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF</td>
<td>2%</td>
</tr>
<tr>
<td>LUT</td>
<td>7%</td>
</tr>
<tr>
<td>Memory LUT</td>
<td>1%</td>
</tr>
<tr>
<td>I/O</td>
<td>26%</td>
</tr>
<tr>
<td>BRAM</td>
<td>2%</td>
</tr>
<tr>
<td>DSP48</td>
<td>1%</td>
</tr>
<tr>
<td>BUFG</td>
<td>16%</td>
</tr>
<tr>
<td>MMCM</td>
<td>29%</td>
</tr>
<tr>
<td>PLL</td>
<td>7%</td>
</tr>
</tbody>
</table>

After synthesis, these interfaces achieve a relatively low utilization of the resources available in the VC707 Evaluation Board, which is necessary to allow for the synthesis and integration of the complete HEVC Chip to the Demo itself. The few resources that do have higher utilization (namely MMCM, BUFG and I/O), deal mostly with clock generation and data input, so the addition of the HEVC Chip will not drive the utilization of these resources even higher. The utilization numbers are summarized
in Table 2.5
Chapter 3

Chapter 3: Conclusion

3.1 Contribution

In Chapter 1, details of the SAO loop filter, a novel processing algorithm designed for parallelization of the SAO filter and the chip architecture of a dedicated SAO processor were described.

The processing algorithm using offset input blocks introduces a novel solution to the problem of performing the SAO filtering algorithm efficiently in a parallel scheme. Not only does it allow for the parallel processing of the SAO filtering algorithm but, together with the double vertical raster scan coupled with a horizontal raster scan scheme presented, it also manages to significantly reduce storage space requirements. This processing algorithm is also scalable, not limited to a certain frame size or degree of parallelization. Altogether, the processing algorithm is the key insight in permitting the architectural characteristics presented later in the thesis.

The SAO processor architecture presented demonstrates the ability to perform SAO processing efficiently in hardware - achieving high throughput rates and a modest area use. Moreover, of significant value is the degree of customizability achieved with the chip architecture presented, which can adapt to different degrees of parallelization, processing orders and area requirements.

Overall, these proposed methods support the idea that SAO filter can be implemented to be an effective, low complexity, low latency, high throughput and low area
component of an HEVC chip.

In Chapter 2, details for the implementation of several interfaces for a real-life FPGA test for the HEVC decoder chip were described, including an HDMI Interface, an SD Card Interface, and the SD-CABAC Interface. Together, these interfaces allow for the transfer of bitstream data stored in an SD Card to the CABAC Engine at a rate of up to 1.5Mbytes per second, which initiates the decoding process of the HEVC Chip. The interfaces also allow for the display of HD video of the decoded bitstream at a 1080p resolution and 60fps refresh rate.

3.2 Future Work

Currently, the degree of parallelization of the SAO processing core (in terms of samples processed concurrently) is set by the overarching system architecture. Loosening this constraint would yield valuable information as to what is the optimal degree of parallelization for the SAO processing algorithm itself. Furthermore, the current implementation of the SRAM module does not allow for energy scaling of memory - it is allocated a discrete block of SRAM in the original synthesis. This is another area for investigation that can yield power savings, by dynamically allocating SRAM storage depending on the input being processed. Finally, an interesting evolution of the SAO processing core would be the ability for it to be merged with other stages in the complete HEVC chip pipeline, something that the Energy Efficient Integrated Circuits and Systems Group at MIT is already investigating.

The real-life FPGA demo is currently limited to pre-generated static inputs, in the sense that input bitstreams have to be mapped to an SD Card and cannot be modified in real-time. An interesting addition would entail the real-time input of live video from an HEVC encoder source.
Bibliography


