Universidade do Minho
Escola de Engenharia

Ricardo Vaz Moreira

Reconfigurable hardware for the new generation IoT video-cards

Novembro de 2019

Universidade do Minho
Escola de Engenharia

Ricardo Vaz Moreira

Reconfigurable hardware for the new generation IoT video-cards

Dissertação de Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do

Professor Doutor Sandro Pinto

Novembro de 2019

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Agradecimentos

Aos meus pais, Almeno Moreira e Maria da Luz Vaz, dedico as minhas primeiras palavras de agradecimento, por todo o apoio dado, não só ao longo desta dissertação, mas ao longo de todo o meu percurso académico.

À minha irmã, Mónica Moreira, dedico este segundo parágrafo, pelo suporte e amizade que, apesar de nem sempre serem evidentes, está intrínseco na nossa relação.

Ao meu orientador, Sandro Pinto, que com o seu conhecimento, experiência e espírito de liderança soube bem definir o caminho a ser seguido. Um exemplo a seguir.

Ao meu co-orientador, Mestre e futuro PhD Miguel Costa, pela transmissão de conhecimento, perfecionismo e rigor que foram essenciais para concluir este trabalho.

Aos meus amigos de curso, em especial ao grupo ”ESRG Top Students”, por todo o apoio, amizade e momentos de felicidade partilhados ao longo do meu percurso académico.

A todos os meus professores que durante estes longos anos me transmitiram conhecimento.

À minha restante família e e amigos que me ajudaram ao longo do meu percurso académico: o meu sincero Obrigado!

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Abstract

Reconfigurable hardware for the new generation IoT video-cards

Embedded systems became a crucial research and developing area because of the dependence of society on devices and the growing demand for new technology products in our lives. The video industry is an example of remarkable technological advances by exploiting the hardware performance for bringing new video products along with even better video quality and higher resolution. Today is time for Ultra High Definition (UHD) resolution and the next new feature is the 8k. A relevant area that may benefit from 8k is medicine, by improving the detail and image quality in diagnoses. Moreover, Japan is preparing to become the first 8k transmitter at the 2020 Olympics.

In spite of existing already general-purpose solutions for managing efficiently UHD video, the deployment of a customized configurable solution can be useful for a specific system needs. Besides, it may dictate market favorable positioning on meeting new market demands by providing faster upgrades.

For addressing this problem, this MSc thesis proposes a hardware-based deployment of two essential reconfigurable cores for a new generation IoT UHD Video-Card, for managing huge memory accesses as well as for compressing video. The memory management provides a memory direct access for dealing with variable video resolution up to 8k, as well as data error control, frame alignment, configurable memory region, and more. The video compression is performed by a configurable core based on an open-source H.264 encoder. The results presented show it was achieved 8k real-time video streaming along with extra control and status functionalities. Video encoding was achieved for up to 8k.

Keywords: 8k, FPGA, H.264, video.

Resumo

Reconfigurable hardware for the new generation IoT video-cards

Os sistemas embebidos tornaram-se uma área fulcral de pesquisa e desenvolvimento devido à dependência da sociedade em dispositivos e à crescente procura por novidades tecnológicas para o quotidiano. A indústria de vídeo é um exemplo do notável avanço tecnológico ao explorar o desempenho máximo do hardware para trazer maior qualidade de vídeo e maior resolução. A resolução de vídeo UHD já é uma realidade e a próxima novidade é o 8k. Uma área de relevo que pode beneficiar do 8k é a medicina, com maior detalhe e qualidade de imagem em diagnósticos. Além disso, o Japão está preparar-se para se tornar o primeiro transmissor de 8k nas Olimpíadas de 2020.

Apesar de existirem soluções capazes de gerir com eficiência vídeo UHD, uma solução personalizada e configurável pode ser útil para as necessidades específicas de um sistema. Além disso, pode ditar um posicionamento dianteiro no mercado ao atender às novas exigências do mercado fornecendo novidades mais rapidamente.

Como possível solução para os problemas expostos, esta tese propõe o desenvolvimento de dois núcleos de hardware reconfigurável essenciais para uma nova geração de placas IoT de vídeo UHD, para gerir acessos à memória assim como para compactar vídeo. A gestão de memória desenvolvida fornece acesso direto à memória para lidar com resolução de vídeo variável e até 8k, além de controlo de erros de dados, alinhamento de frames, região de memória configurável e muito mais. A compactação de vídeo é realizada por um núcleo de hardware configurável, baseado num Encoder H.264 de código aberto. Os resultados mostram que foi alcançada transmissão de vídeo 8k em tempo real, além de funcionalidades extras de controlo e estado. A codificação de vídeo até 8k foi alcançada.

Palavras-chave: 8k, FPGA, H.264, video.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

AMBA Advanced Microcontroller Bus Architecture.

API Application Programming Interface.

ASI Asynchronous Serial Interface.

AXI Advanced Extensible Interface.

BME Block Motion Estimation.

BNC Bayonet Neill–Concelman.

BRAM Block RAM.

COTS Commercial-Off-The-Shelf.

CPU Central Processing Unit.

DDR Double Data Rate.

DMA Direct Memory Access.

FHD Full High Definition.

FPGA Field Programmable Gate Array.

fps frames per second.

HDL Hardware Description Language.

HDMI High Definition Multimedia Interface.

IA Intel Architecture.

IDE Integrated Development Environment.

ILA Integrated Logic Analyzer.

IoT Internet of Things.

IP Intellectual Property.

LUT Look Up Table.

MB Macroblock.

MPSoC Multi Processor System on Chip.

NAL Network Abstraction Layer.

PCIe Peripheral Component Interconnect Express.

PL Programmable Logic.

PS Processing System.

QP Quantization Parameter.

RAM Random-Access Memory.

RTL Register Transfer Level.

SD Standard Definition.

SDI Serial Digital Interface.

SDK Software Development Kit.

SFP Small Form-factor Pluggable.

SMPTE Society of Motion Picture and Television Engineers.

SoC System on-Chip.

UHD Ultra High Definition.

VDMA Video Direct Memory Access.

VTPG Video Test Pattern Generator.

1. Introduction

The present chapter outlines the problem addressed in this MSc thesis, as well as the goals that must be achieved in order to present a solution for solving that problem. Finally, a brief description of the structure of this document is presented.

1.1 Problem Statement

Due to the strong growth of the electronic industry and the demand for improved devices required for society, embedded systems have been facing even more challenging problems over the years. Currently, there is plenty of domains in which embedded system are applied, from basic consumer electronics [14], Internet of Things (IoT) devices [15, 16], aerospace control systems [17], medical devices, machinery [18, 19, 20], and more.

Over the years, we attended the improvement of image resolution. Now it is time for more and more UHD solutions to appear. Although 4k is already very present in the market for broadcasting solutions [21], 8k is still being prepared, but it is close. 8k has been called the next video generation [22] and it is already being prepared by industry to provide a wide range of UHD options from acquisition to playback. The expected is that it will become the new great video feature by the year of 2020 [23]. Japan plans to pave the way for 8k content broadcasting for the first time in the 2020 Olympics [23]. Raising the actual standard Full High Definition (FHD) to UHD seems to be a challenge accompanied by enthusiasm because it would serve people every day as well as crucial areas such as medical imaging devices and microbial analysis [20, 24]. Medicine seems to be one of the beneficent of such video resolution once it may allows improvements in image detail and quality, which may be useful for diagnoses [20].

Furthermore, along with the current tendency of connecting every device to the internet and raising content broadcasting, there is a demand for providing video compress solutions. 8k demands a device to support data rate transmission from 12 Gbit/s, at its lowest format, up to 143 Gbit/s. The data bandwidth required is tremendous comparing with the current FHD (1920 x 1080) standard. In the case of 12-bit per color component, the 8k (7680 x 4320) is 4 times larger than 4k (3840 x 2160) and 24 times larger than FHD with its 8-bit standard pixel format [24, 25]. Therefore, when it comes to internet publishing and streaming of such amount of data, there is a demand for providing encoded video [26].

Memory bandwidth plays a central role in enhancing the real-time transmission of video content. Over the years, there is a gap in devices between Double Data Rate (DDR) memories and memory controllers due to a faster pace for DDR memories in performance improvements [27].

The project *MOG WALL SCREEN*, in which this thesis work is inserted, arises from this scope. This project is funded by *COMPETE 2020* and started in July 2017 to be completed in July 2020. It is promoted by *MOG Technologies*, S.A., *INOV INESC Innovation* and *Universidade do Minho*. The main goal addressed to this project is to develop a platform capable of linking UHD professional video with information system infrastructures. This platform is based on IoT trends and pretends to allow not only acquire and reproduce audio and UHD video but also ensure its publication in the internet, management, and monitoring, as well as the collection of statistical data. It is in this context that some general goals were defined: (i) design and implement of a hardware acceleration system capable of real-time processing of large amounts of data (UHD video up to 8k); (ii) design and implement video reception and transmission modules and other interfaces in HDL language; (iii) increase performance at UHD video capture, compression and playback (especially 8K video content). Following this project, this MSc thesis proposes the deployment under FPGA technology of a system capable of managing and optimizing huge memory accesses required for video up to 8k as well as to compress video to allow lower bit rate requirement for internet publishing purpose.

1.2 Aim and Scope

The main goal addressed to this MSc thesis is to develop a reconfigurable hardware for the new IoT video-cards, mainly focused on memory management as well as H.264 video encoding. Due to the huge amount of data to be dealt with when treating UHD video, the solution needs to be hardware-based instead of a general-purpose processor, to provide higher throughput, according to the technologies available. The proposed work for this MSc thesis aims to achieve some goals divided in two main cores:

1. Provide a reconfigurable and dedicated memory management:

    (a) Manage memory accesses and optimize it;

(b) Configurable video parameters on-the-fly;

(c) Configurable memory space and rotation;

(d) Provide status and control register and accesses to it;

(e) Mechanism of frame error detection;

(f) Self-sufficient management between writing and reading frames from memory for video streaming;

(g) Define memory region occupied for each frame and guarantee integrity.

2. Encode raw video with up to 8k resolution:

   (a) Configurable video parameters on-the-fly;

   (b) Header production and inclusion in encoded frame information;

   (c) Provide status and control register and accesses to it.

1.3 Document Structure

The present document follows a structure in which Chapter 2 discusses the theoretical concepts and technologies necessary for the implementation of this project, ending with the review of some work of the scientific community and industry. It starts with a background of the technologies around Serial Digital Interface, Memory Management, H.264 Encoding, and Peripheral Interface Component Express. The chapter ends with a State of the Art of some relevant video-cards available in the market. Chapter 3 provides details about the System Specification. It starts by explaining the resources available in the *Xilinx ZCU102* platform, followed by a deep statement about the AXI4 protocol. After that, the General architecture of the IoT video-card is dissected. Lately, the Design phase applied to this MSc thesis is detailed. Chapter 4 aims for System Development, which gives an overview of some decisions taken during this phase. The results were taken to prove system functionalities and they are exposed in chapter 5. Lastly, chapter 6 provides a conclusion on the deployed system as well as future improvements.

2. State of the Art

In this chapter, it will be first presented some fundamental concepts and background around this thesis theme, followed by a brief review of available solutions to tackle the problem introduced in the previous chapter. Consequently, this chapter is divided into two sections. The first focus on providing all the background knowledge essential to review and design a reconfigurable hardware and software stack for the new generation IoT video cards. The second is devoted to the analysis of the current solutions, provided by the scientific community and industry players, to overcome this problem.

2.1 Background

The present section addresses some concepts essential for a critical analysis of the hardware and software stack of IoT video- cards. As a consequence, it is divided into four subsections. The first covers the details of the Serial Digital Interface protocol, being followed by a subsection dedicated to the peculiarities of memory management in UHD video cards. The last two subsections address the H.264 protocol and the PCIe.

2.1.1 Serial Digital Interface

Nowadays, digital video data can be transported using one of two serial interface formats: (i) Serial Digital Interface (SDI), for uncompressed data, and (ii) Asynchronous Serial Interface (ASI), for compressed data. As usual, in the video production environment, video and audio are transported via SDI as defined by the Society of Motion Picture and Television Engineers (SMPTE) [28]. There are a lot of hybrid SDI/IP models used to respond to the necessities of designing hardware video systems in the embedded world. SDI became especially important because it is the only standard for digital video transmission over coaxial cable and it seems that it will continue playing an important role in the design of future UHD video solutions [29].

For a long time, SDI has been used as a solution for broadcasting production and delivery systems, supporting data transfer rates defined since the old Standard Definition (SD), to the recent UHD, in which is included the 4k and 8k video resolutions. It is also characterized to be used by the industry in the professional domain and has evolved during the years to suit application-specific interfaces for standard media digital signals that may range from 270 Mb/s up to 192 Gb/s [30]. To support the increasing amount of data demanded by the higher resolutions and color depth, additional SDI standards have been introduced. In spite of being originally designed to support video in YUV 4:2:2 format, it can also carry other different formats, for instance, YUV 4:4:4, YUY2 (also known as YUYV) 4:2:2, YUV 4:2:0, or even RGB [30]. The bit depth may also vary between 8, 10, 12 or 16 bits. Among those new SDI standards, some may fit the purpose for the video card proposed in this MSc thesis. Such standards are reviewed in the next subsections.

2.1.1.1 3G-SDI - SMPTE 424M

The 3G-SDI is a standardized SDI interface solution for 3-Gbit/sec (nominal) data transfer rates, physically supported on the using of coaxial cables and Bayonet Neill–Concelman (BNC) connectors, whose specification was released in [1].

The genesis of this interface was based on the desire of creating a single link 1080-line transport of video instead of the dual-link used until the release of this technology. The concern of keeping the same physical interfaces and infrastructures as its predecessor was always a priority for the SMPTE because of the problems, policies, new expenses, and some general disorder that it would mean for all video industry on applying completely new technology.

At the 3G-SDI releasing, the motivation was already to allow future implementations with new and higher quality video to be delivered to the final consumer. Another key factor for the genesis of this artifact was the digital cinema industry and the video quality demanded by this particular industry.

The 3G-SDI allowed to deliver 2048x1080p 4:4:4 with 12-bit color depth at rating of 24 frames per second with only one link, instead of the dual-link needed with the 1.5G-SDI solution. Figure 2.1 pretends to show the required bit rate for some video formats, which were the most important until the release date of 3G-SDI. As can be observed, a single-link 3G-SDI can satisfy the bit rate demanded by all of them. For a higher frame rate or higher resolution, we would need multiple links to deal with the data carried on the data bus.

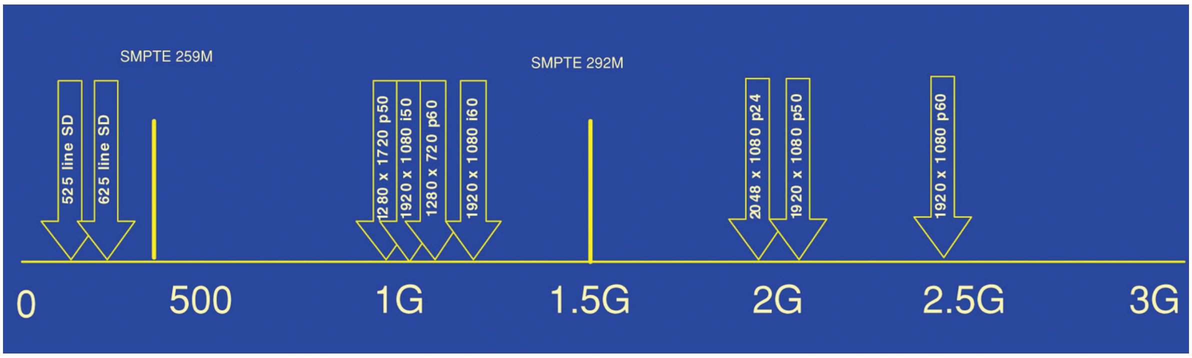


Figure 2.1: Required bit rate to transmit various video formats [1].

2.1.1.2 6G-SDI SMPTE ST 2081

SDI has continued to evolve, maintaining the same interfaces and infrastructures as previous versions. This backward compatibility has proven to be a key-factor for SDI to hold its position as a main digital video data transmitter. SDI-6G was firstly introduced in [31] and it seems to be the first SDI capable of 8k video content delivery. Nevertheless, to accomplish that, a quad-link topology is still needed. In terms of video quality, this solution is still a little tangential, since it can only deliver 7680x4320 video resolution in formats up to YUV 4:2:2, with a maximum of 10-bit color depth and 30 frames per second (fps). This is still insufficient for the cinema industry, which has been assuming a key role in the evolution of SDI standards.

It seems obvious that there is a need for continuously improving the SDI performance to enable higher video quality and resolution since the market is becoming more and more competitive, where each detail may mean higher market share and profit. To respond to those requirements from the industry, it is necessary to keep developing higher capacity technology. The 12G-SDI may have emerged to tackle this issue.

2.1.1.3 12G-SDI SMPTE ST 2082

It is important to emphasize that increasing the video resolution from 4k to 8k requires four times more data processing. Although the immediate perception inclines us to think that comparing both 4k and 8k video resolution would mean only twice capacity, the truth is that it signifies having four images of 4k resolution each. If processing 4k video is already challenging because of the amount of data buffering and throughput required, designing 8k video resolution systems require a far more timing-critical technology.

12G-SDI surged [32], just in time to enable the delivering of 8k video resolution content with honorable quality and higher frame rate. A Dual-link 12G-SDI can transport 8k video resolution up to 30fps with a 10-bit color depth and 4:2:2 signal format sampling. Meanwhile, a Quad-link 12G-SDI is enough to deliver the same video parameters as the Dual-link, but up to 60fps. This improvement results in a more fluid video and consequent improvement of user experience.

Multiple link configurations with a defined SDI bit rate are frequently thought as a solution to achieve higher video quality. However, implementing SDI multi-link solutions (dual, quad or even octal) is usually challenging and too expensive, requiring large board space as well as high power consumption. This may explain why new solutions with improved performance and capacity for single-link SDI keeps appearing.

2.1.2 Memory Management

The requirement for a memory manager, in general, exists due to the demand for managing memory accesses from system modules that may try to read and write concurrently in memory. Once usually there is only one interface for reading and another for writing, all the modules that may try to read or write data may need to wait to get the required access. The demand for a Memory Manager becomes even higher when a large data block is being transferred. A Memory Manager may represent one of the most important and timely critical parts of a system since it will be managing, organizing, prioritizing, and ordering data interchanging in memory.

An implementation based on an independent unit capable to handle memory accesses allows the system to be ready to not only transfer higher data blocks but also free the processor for other complex tasks that may require additional math and processing from the Central Processing Unit (CPU) [33]. It is also usual to see general-purpose solutions such as Direct Memory Access (DMA) [34], as long as when it is enough to satisfy the system requirements. Moreover, there is VDMA [2], which is based on DMA but optimized to video applications. It has specific configurations for video purposes to reduce the engineering effort by abstracting the video color space, data transfers, memory managing, and more.

DMA is often used in implementations that may require a big amount of data to be transferred [33]. It allows the system to be more efficient by enhancing parallelism, not only for processing data but also for higher bus width which permits exploit the memory resources at its maximum capacity. Since usually memory can operate at a higher frequency than the bus protocol can, there may exist no way but increasing parallelism to deliver enough data to take advantage of full memory throughput.

VDMA is a specific application purpose of DMA [2] to treat video. Usually, video content is transmitted via bus protocol with some extra important signals for synchronization and monitoring of incoming data [35]. It may offer higher reliability and allows the system to validate or discard transfers with the extra signals received.

Data buffering is crucial for achieving high-speed communications [27]. It allows the system to be as much as possible timing efficient in data interchanging since it may only deliver content when it is ensured that the system is ready to be efficient in the reading and writing process. There are at least two situations in which buffering seems to help systems increasing its efficiency and performance on delivering higher throughput:

- Data coming from modules in which the data bit rate is lower than the memory capacity;
- Memory busy, so we need to buffer to avoid congesting the system and compromise it.

In the first item, we would be busying the bus interface without taking advantage of its full performance, which may create congestion. The second hypothesis refers that if buffering would not be done, the data source would have to wait for delivering the data. In a worst-case scenario, if the data source cannot wait, data can be lost. Furthermore, buffers may perform an important role in solving problems associated with multiple clock domains in systems [27]. Two different clock domains are allowed for accessing data in different states (read or write). Nevertheless, buffering increases inevitably latency issues [36], and consequently, building a system with this mechanism is a trade-off that a system may pass through, depending on the application purpose.

2.1.2.1 Video Direct Memory Access

AXI VDMA is a soft IP core, designed by *Xilinx* for the *Vivado Design Suite*, which offers a high-bandwidth DMA between a video source, transmitting AXI4-Stream Video data, and a memory [2]. It is a very powerful solution for reading and writing video frames in memory while abstracting the programmer from the peculiarities of the AXI4-Stream Video protocol. Figure 2.2 presents the general architecture block diagram of the AXI VDMA.

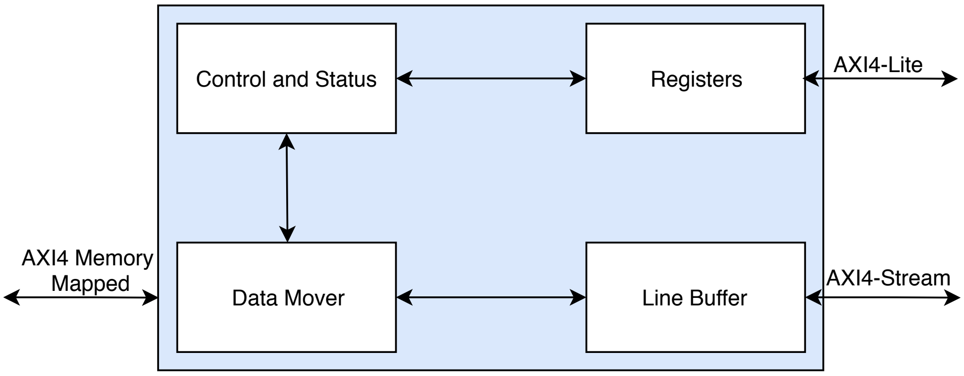


Figure 2.2: AXI VDMA Block Diagram (Adapted from [2]).

This IP core is equipped with a memory-mapped register space containing status and configuration registers. The configuration register can be updated on-the-fly, providing a lot of flexibility for application design. The status register acts as a powerful interface to spot critical errors that may occur during application execution.

There are two paths for data interchanging: (i) AXI VDMA write subsystem and (ii) AXI VDMA read subsystem. In the read path, the AXI4 Memory-Mapped Master interface reads frames from memory and outputs the data on the AXI4-Stream Master interface. Whereas the write path receives frames through the AXI4-Stream Slave interface and writes the content in memory by the AXI4 Memory-Mapped Master interface. Both subsystems operate independently from each other but it is possible to configure the system in order to synchronize the input/output frames. Additionally, there is a line buffer that temporarily holds data to deliver it later by the AXI4 Memory-Mapped Master interface or AXI4-Stream Master interface. Some of the most important features supported by AXI VDMA are enumerated in Table 2.1.

Table 2.1: AXI VDMA Features.

Feature Supported	**Characteristics**
AXI4 Data Width	32, 64, 128, 256, 512, and 1,024 bits
AXI4-Stream Data Width	From 8 bits up to 1,024 bits
Frame Buffers	Up to 32 frames in buffer
Data Realignment Engine	Allows the frame buffer to start at any memory address
Genlock Synchronization	Mechanism to synchronize writing and reading of frames
Asynchronous Channels	Asynchronous clock domains for each AXI4 core interface
Frame Sync Options	*tuser* signals the start of a frame, while *tlast* signals the end of a line
Frame Advance or Repeat on Error	If frame error is detected, VDMA proceeds to the writing of the next frame or can try to write the same frame once again

The width of the data buses inherent to the write (S2MM) and read (MM2S) subsystems are reconfigurable to fit the application. This IP core presents its major flaw in what concerns to frame buffers. In fact it only supports a maximum of 32 buffers, independently of the frame resolution and bit-color depth. This limits the usage of large memory, as part of it would be unused. In contrast, it can start buffering at any part of the memory.

2.1.3 H.264 Encoding

There is an increased demand for video solutions capable to operate higher and higher resolutions as technology is moving on. Video applications are quickly stepping forward from FHD to Quad-FHD, while 8k emerges in the market and starts to define the next-generation video applications [22].

The demand for an encoder emerges from the necessity of compact video content in to reduce the bandwidth required to stream the inherent data via the internet as raw 8k video demands a data rate transmission from 12 Gbit/s up to 143 Gbit/s [12]. The bandwidth required to transmit that amount of bits per second is not yet supported by our web services infrastructures, but even if this were possible we do not need to, as the human eye is not accurate enough to perceive such residual differences [37]. The presence of a H.264 Encoder for video solutions may be useful since the video size can be drastically reduced, enabling the transmission of video content through a web service [38].

Usually, encoders use the YUV color space instead of RGB because of the human eye perception. Even though the RGB color space is much more intuitive to understand, once it is based on the three primary colors (Red, Green, and Blue) which combined can originate any color, scientists had found that our eye is much more sensitive to luma than color variations [37]. Engineers have been using this finding for increasing throughput and save memory resources, while maintaining image quality. The conversion from RGB to YUV can be easily performed only by applying the equations below [37]:

$$Y = 0.299 \times R + 0.587 \times G + 0.114 \times B \qquad (1)$$

$$U = -0.147 \times R - 0.289 \times G + 0.4378 \times B \qquad (2)$$

$$V = 0.615 \times R - 0.515 \times G - 0.1 \times B \qquad (3)$$

In raw video every single pixel is coded, which makes raw video size much larger than encoded video. In the H.264 Encoder, only the different parts need to be codded from a 16x16, 8x8, or 4x4 MB for luma (Y) and also for each chroma (U and V) [39].

In comparison to previous standard MPEG2, the H.264 is designed to achieve up to 59% more bit-rate efficiency in spite of being both designed for streaming purposes [40]. Whereas the standard MPEG2 was developed to be focused on low bit-rate, H.264 can code not only low bit-rate but also better video quality successfully.

Two types of prediction contribute to encoding data: intra-frame prediction and inter-frame prediction. Both predictions are carried out by a set of complex mathematical functions. The frames compression is performed by applying a set of three techniques: prediction, transformation, quantization, and entropy coding [26]. Even though there are two types of frame encoding in H.264, that does not mean that the encoded video needs to be fully performed by intra-frame or inter-frame prediction. Moreover, a video can be entirely encoded in intra-frame prediction, but it can not be only inter-frame coded. While intra-frame prediction is coded independently from consecutive frames, the inter-frame one requires a reference so that the decoder can interpret correctly the information by comparison with an old sample frame.

In spite of the best solution for memory saving and bit-rate being the inter-frame prediction, it may carry some problems for image integrity in the decoding phase. First of all, the very first frame of a video needs to be intra-coded once there is still not a reference for the decoder to interpret the current data frame. The remaining sequenced pictures may be temporally coded (inter-prediction) [40]. There may exist some residual and imperceptible errors in temporal coding due to chaining prediction performed by the encoder. Therefore, to avoid error propagation, it is recommended the use of intra-frame coding. Otherwise, if only spatial coding is used, the eventual encoding/decoding redundancy errors that may occur would result in a significantly adulterated picture [40]. It seems that the best approach may be a hybrid mechanism, contemplating both, spatial and temporal prediction. The first one serves the purpose for higher bit-rate values and the second one to guarantee image integrity.

Another important element of H.264 encoding resides in the header it generates for each encoded frame. Since this encoder is designed to fit several applications as well as variable frame resolution, the information carried in the header is crucial for the decoder to interpret the content. The Network Abstraction Layer (NAL) unit type is specified as well as the payload, which carries the data related to the specific NAL [3].

2.1.3.1 Spatial prediction

In spatial prediction (intra-frame prediction) the estimation is based on MBs presented in the same frame (i.e., there is no reference frame for the current one to be encoded). Since neighbor pixels tend to be similar, it is possible to code efficiently only the differences [40]. This technique may result in a significant reduction of memory occupation for encoded data comparing to raw format for the same image.

A MB is the basis of intra-frame prediction once it is the only source it takes to encode a frame. There is no data passed between different frames to encode the current one. Therefore, according to the H.264 standard, the encoding is fully performed by taking advantage of spatial prediction, where each MB is encoded and temporarily stored in memory so that it may be possible to use this output product to encode the next MBs [40]. Then, only the differences are required to be encoded by comparison because of the resemblance of border pixels.

Intra-prediction is performed within nine different modes for luma samples (if the MB is partitioned into 4x4 blocks), whereas for chroma samples only four modes are allowed [3]. All nine modes shown in Figure 2.3 can be used for luma intra-frame prediction, but only the horizontal, vertical, DC, and plane modes are available for chroma.

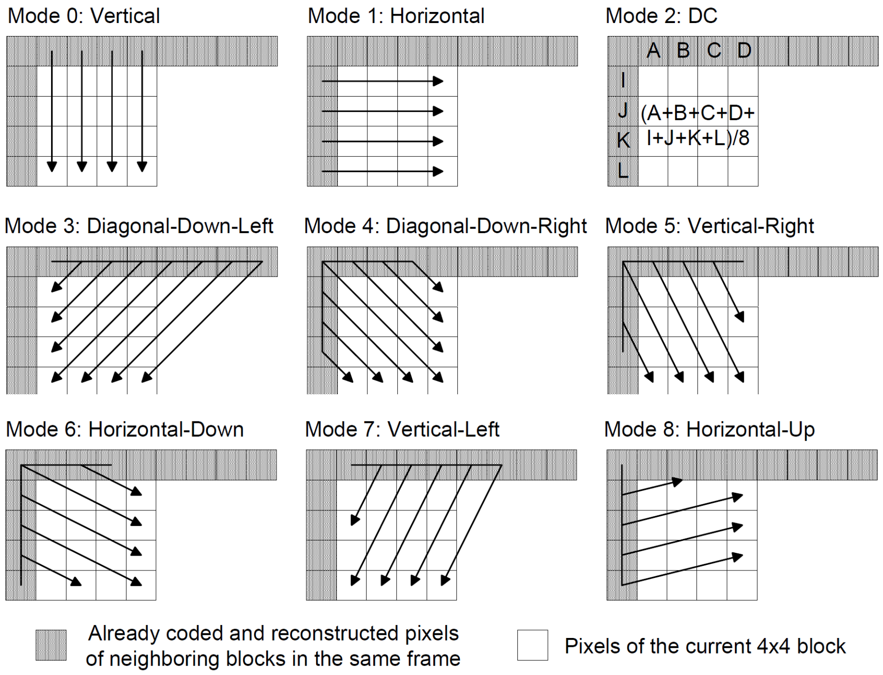

Mode 0: Vertical
Mode 1: Horizontal
Mode 2: DC
A
B
C
D
I
J
K
L
(A+B+C+D+
I+J+K+L)/8
Mode 3: Diagonal-Down-Left
Mode 4: Diagonal-Down-Right
Mode 5: Vertical-Right
Mode 6: Horizontal-Down
Mode 7: Vertical-Left
Mode 8: Horizontal-Up
Already coded and reconstructed pixels of neighboring blocks in the same frame
Pixels of the current 4x4 block

Figure 2.3: Intra-prediction modes for 4x4 MB[3].

It seems clear that encoding a video whose details are so high and different in consecutive pixels may result in less memory saving than a video of a landscape mainly filled by a clear blue sky, for instance. But after all, there is always a better bit-rate than in raw format. It may be clear now that there is not a static percentage in data saving by encoding video with H.264 since it depends on the scenario.

2.1.3.2 Motion prediction

Motion prediction (inter-frame prediction) consists of motion estimation between consecutive frames. Once two consecutively frames are very likely to be similar, there is not a demand for encoding every pixel in the next frame again. This technique reduces memory occupation, being even more compression efficient than the spatial prediction [14]. As can be observed in Figure 2.4, the main goal of this technique is to decrease temporal redundancy.

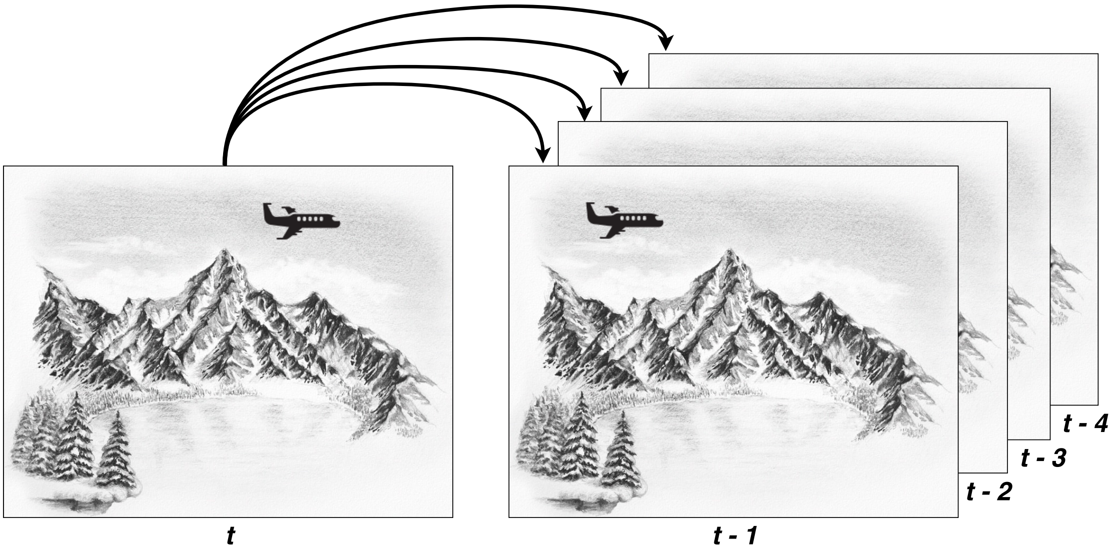


Figure 2.4: Example of Multi Reference Motion Estimation.

Then, for better motion prediction, each 16x16 MB can be split into 8x8, 8x4, 4x8 or 4x4 sub-macroblocks, also known as *sub-mb-type* [3]. The coding of such type of prediction is based on search areas, where the Block Motion Estimation (BME) is applied to predict movement in a rectangular block from the current frame. For this purpose, BME is used to compare neighboring MBs by applying a sum of the absolute difference in the compared areas [14].

2.1.3.3 MacroBlocks, Slices and Quantization Parameter (QP)

A division in MB and slices is a request by the H.264 standard to code the content. Typically, for video in 4:2:0 chroma sampling format, there is a demand for organizing the data in 16x16 matrices for luma (Y) and 8x8 for each chroma sample (U and V). It means that there is a request for unaligned memory accesses since a frame is usually saved in memory according to an interleaved or planar format, which can compromise the throughput of a system by making memory accesses less efficient.

Slices in H.264 serve the purpose of informing the encoder that the next MBs need to be spatially coded, not getting a reference of an earlier frame of the sequence. This type of slices is called *I slices* [40]. In case of absence slice, the next MBs will be temporally coded.

The Quantization Parameter (QP) is applied to residual error data (the difference between prediction and the current MB), where an integer number is given to the encoder to define how much high frequencies data can be discarded. Since the human eye is less sensitive to higher frequency, we can increase the QP to reduce the information coded for high frequencies and thus reduce memory footprint [14].

2.1.3.4 Bitstream and NAL Unit

The bitstream is an important element of the encoder since it has all the relevant information to decode correctly the content. The H.264 standard was designed to support new possible features and higher video resolutions than those used by video systems at the releasing time of H.264. This problem led to the adoption of the Exponential Golomb universal code since it allows the header length to vary instead of being fixed to a given number of bits, difficulting a future upgrade of the encoder. In Table 2.2 is presented how a non-negative integer number can be coded in Exponential Golomb.

Table 2.2: Exponential-Golomb code words.

Integer Number	Binary Number	Exp-Golomb Number
0	0	1
1, 2	1, 10	010, 011
3, 4, 5, 6,	11, 100, 101, 110	00100, 00101, 00110, 00111
7, 8, 9, 10, ...	111, 1000, 1001, 1010	0001000, 0001001, 0001010, 0001011, ...

At the beginning of each frame file, there is always a predefined structure header, whose parameters are used by the decoder to correctly interpret and decode the video data. The SPS, PPS, and IDR are firstly identified by the header, enabling the later decoding of the payload. IDR stands for cleaning the picture references presented in the decoder buffer, consequently, the next frame needs to be spatially coded (intra-frame prediction) [3].

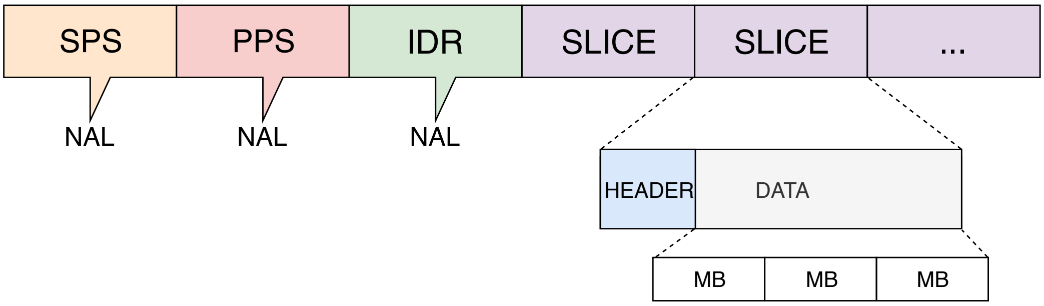


Figure 2.5: H.264 Bitstream Structure.

Figure 2.5 presents the data structure of a bitstream resulting from H.264 encoding. SPS is coded along with the profile and level parameter, which allows the decoder to check if the profile and level are supported, as they may contain some information about the application, maximum bit rate and maximum frame size [40]. If the profile or level is not supported the decoding may be rejected.

2.1.4 Peripheral Component Interconnect Express

PCIe is a high-performance bus interconnection for every platform these days. It allows high data rate communication for transfer applications between a host and a FPGA [41]. With the advancement of technology and the increasing amount of data speed buses, PCIe is becoming widely used and deployed in a large variety of Intel Architecture (IA) and non-IA platforms [11]. Estimations say that PCIe technology doubles data rate every three years [42]. Table 2.3 provides a bandwidth comparative analysis of each available PCIe generation.

Table 2.3: PCIe bandwidth for different generations and lane configurations, based on [11] and [12].

	Raw Bit Rate	BW x1	BW x8	BW x16
PCIe 1.x	2.5 GT/s	∼250 MB/s	∼2 GB/s	∼4 GB/s
PCIe 2.0	5.0 GT/s	∼500 MB/s	∼4 GB/s	∼8 GB/s
PCIe 3.0	8.0 GT/s	∼1 GB/s	∼8 GB/s	∼16 GB/s
PCIe 4.0	16.0 GT/s	∼2 GB/s	∼16 GB/s	∼32GB/s

8k video at 60fps demands a data transfer rate equal to or higher than 15.9 GB/s by PCIe. Therefore, a PCIe technology below of Gen3.0 x16 or Gen4.0 x8 does not fulfill the requirements to equip video cards transmitting video with these characteristics in real-time.

2.2 Related Work

8k is still an emerging technology and, as a consequence, there are a few 8k-ready video cards available in the market. This section pretends to review some of these platforms, focusing on the ones similar to the video-card pretended to be developed during the MOG WALL SCREEN project, where this MSc thesis is inserted.

2.2.1 UHD Demosaicing on Low-Cost FPGA Pavel

The results exposed in [43], show that it is possible to process UHD video in a low-cost FPGA. Although this solution did not achieve a transfer of 8k in real-time, it is still an interesting solution since it was developed based on a low-cost FPGA and it is still able to treat 4k resolution video, contrasting the recommendation by *Intel* and *Xilinx* to choose high-cost FPGA like *Arria*, *Stratix*, *Virtex* and *Kintex*. According to the board specifications available in the market, it usually refers to higher operation frequency than it could achieve by testing in real conditions, which may be disappointed when passing the analysis to the design phase, as this paper describes. In addition, it can lead to money and time lost since sometimes it may be impossible to perform the needed System on-Chip (SoC) to accomplish the requested specifications. Despite this, the article describes this solution was deployed in the low-cost FPGA *Intel Cyclone V*. The broadcast interface is performed by a 12G-SDI video stream, whereas the color space used in the video transferring is RGB. The pixel clock is set at 596 MHz. Due to FPGA limitation cells that could not execute the tasks at an enough frequency to process the needed data, the solution taken ahead to accomplish the system has been by increasing parallelism. This technique may lead to a larger occupation of the board resources and consequently increased energy consumption. Therefore, the maximum frequency set for the FPGA programmable logic was 124.07 MHz.

2.2.2 AJA Kona IP

The *AJA KONA IP* supports only video resolutions up to 2k pixels. It can to communicate with a workstation by using the PCIe technology Gen2x8. The High Definition Multimedia Interface (HDMI) 1.4 is available for video connectivity. Moreover, it has two Small Form-factor Pluggable (SFP)+ connectors for input and output of video content. The broadcast interface is performed by a multi-link 3G SDI [4].

Figure 2.6: AJA Kona IP [4].

2.2.3 DELTA-3G-elp-tico-d 4C

The *DELTA-3G-elp-tico-d 4C* is one of the two models available in the market provided by *DeltaCast*. It has four BNC connectors for the Quad-Link SDI interface. Additionally, it is possible to configure each BNC connector for supporting other video formats. The quad-link SDI interface enables the transmission of 4k video in YUV 4:2:2 with 10-bit color depth. The external communication is performed by a Gen2x8 PCIe data bus. [5].

Figure 2.7: Delta-3G-elp-tico-d 4C [5].

2.2.4 Magewell Pro Capture AIO 4K Plus

The *Magewell Pro Capture AIO 4K Plus* has one BNC connector for broadcast interface only with a Single-Link 6G SDI. It supports video transmission up to 4k at 60 fps by the HDMI 2.0 video interface. Additionally, this platform is equipped with PCIe Gen2 x4 technology, allowing data transfer rates up to 2 GB/s. [6].

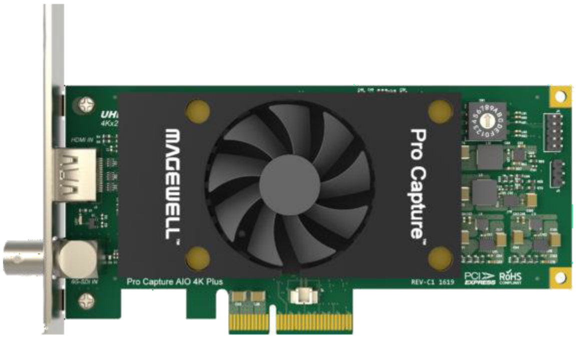


Figure 2.8: Pro Capture AIO 4K Plus [6].

2.2.5 Bluefish444 Kronos Optikos

Even though the *Bluefish444* has two video-Cards available in the market, the *Kronos Optikos* seems to be the most complete solution, when compared with the *Kronos Electron* version. This platform has an octa-link 3G SDI interface available for video transmission, supporting up to 4k video resolutions.

Some 8k video formats may also be supported by this board, in spite of being a little restricted in this aspect. This board is equipped with an HDMI 2.0 interface, already supports the transmission of 4k video. Additionally, it has three broadcast interfaces available, two for SFP+ and one Dual-Link 3G SDI along with BNC connectors [7]. The PCIe technology available on this board is the

Figure 2.9: Bluefish444 Kronos Optikos [7].

2.2.6 Dektec DTA-2179

The *Dektec DTA-2179* presents a higher number of BNC connectors (twelve), where each one can be configured individually for input or output by software. From this set of BNC connectors, only eight support 3G-SDI, which limits the number of 3G-SDI connections following a quad-link topology to two. This board is available with PCIe Gen3x16 technology, allowing data transfer rates capable of handling 8k video in real-time. However, unlike the SDI technology provided, the remaining components that are part of this video-card seem to be short to accomplish 8k video. Nevertheless, it is able for transmitting up to 4k video [8].

Figure 2.10: Dektec DTA-2179 [8].

2.2.7 Discussion

Table 2.4 summarizes the most relevant specifications of the video boards reviewed during this section.

Table 2.4: Characteristics summary of the analyzed Video-Cards.

Name	Video Interfaces	Broadcast Interfaces	Data Bus	4k/ 8k?
UHD Demosaicing on Low-Cost FPGA Pavel	BMC	Single-Link 12G SDI	–	4k
AJA Kona IP	HDMI 1.4	Multi-link 3G SDI	Gen2x8	–
DELTA-3G-elp-tico-d 4C	BMC	Quad-link 3G SDI	Gen2x8	4k
Pro Capture HMDI 4K Plus	HDMI 2.0	Single-link 6G SDI	Gen2x4	4k
Bluefish444 Kronos Optikos	HDMI 2.0	2 SFP+ & Dual-link 3G SDI	Gen3x8	8k/ 4k
DTA-2179	BMC	Octa-link 3G SDI	Gen3x16	4k

From the set of boards reviewed, *Dektec DTA-2179* presents the most powerful PCIe technology. Its PCIe Gen3x8 and x16 are solutions already able to handle the data rate demanded by 8k video transmission. On the other hand, the remaining technologies presented seem to be a little short. The presence of HDMI 2.0 is also a valuable resource since this HDMI version is already able for transmitting up to 4k video in raw format and even 8k in some compressed video formats.

Bluefish444 Optikos seems to be the only option able for treating 8k video in raw format due to the two SFP+ and the Octa-Link 3G SDI available. The bottleneck of this video card resides in the PCIe technology adopted, as PCIe Gen3x8 does not have sufficient throughput to handle 8k video in raw format. From this review, it can be easily spotted the gap in the industry that this MSc thesis pretends to tackle.

3. System Specification

In this chapter is presented the design phase behind the reconfigurable hardware for a new generation IoT video-card. It will be presented and discussed the design of some critical points to achieve as much as possible throughput to guarantee the proposed goal of 8k at 60fps. The design phase will focus only on the modules aimed to this MSc thesis, namely on Memory Manager and H.264 Encoder.

3.1 Development Environment

This chapter presents the tools and platforms used in the development of the work inherent to this MSc thesis. Some tools are requirements of the project, whereas others are used just for convenience. Both tools are summarized in this chapter, being highlighted its main features.

3.1.1 Zynq UltraScale+ MPSoC ZCU102

The Xilinx UltraScale+ architecture allows multi-hundred gigabit-per-second levels of performance for efficient routing design and data processing in an SoC. The ZCU102 board is based on the Zynq UltraScale+ XCZU9EG-2FFVB1156E Multi Processor System on Chip (MPSoC), which combines PL with a powerful PS, as depicted in Figure 3.1.

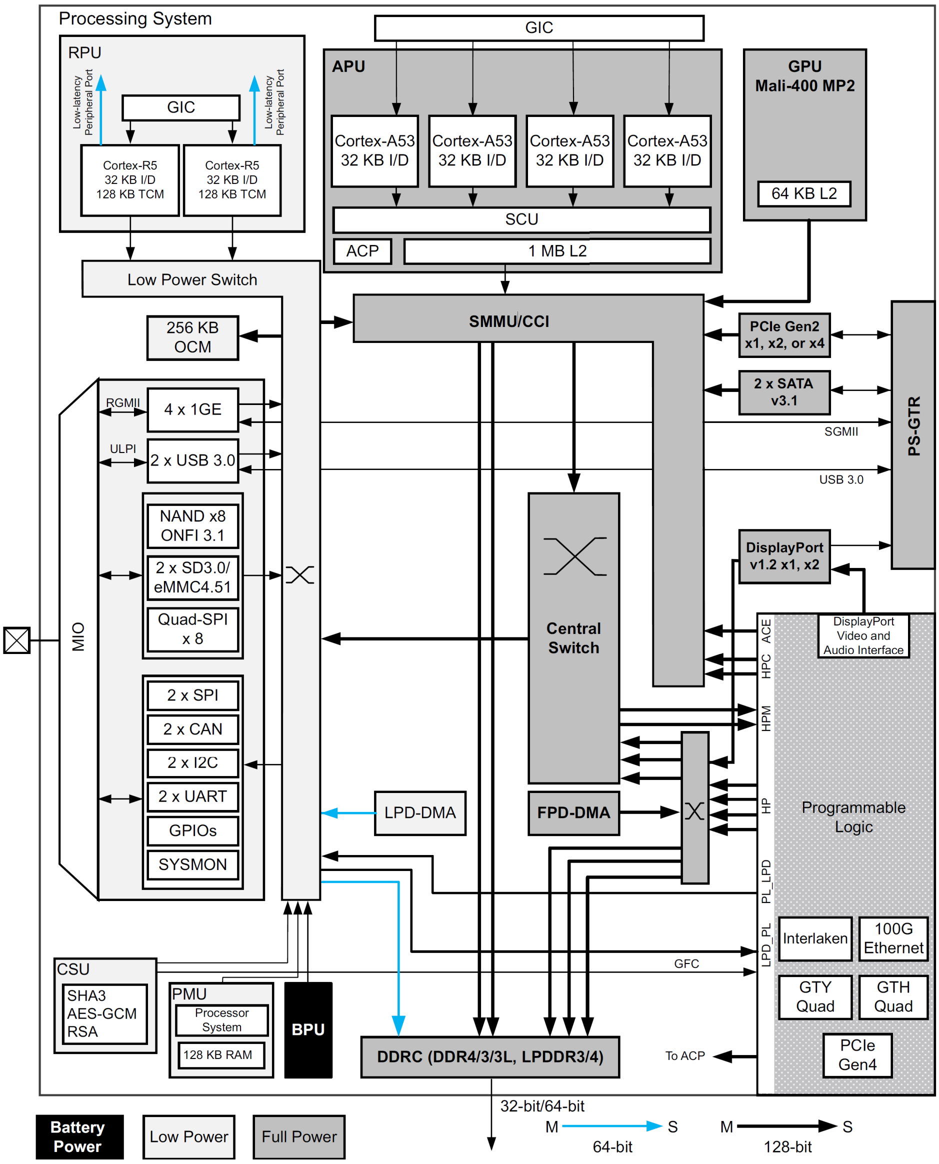


Figure 3.1: Zynq UltraScale+ MPSoC Overview Block Diagram [9].

The Processing System features the Arm flagship Cortex-A53 64-bit quad-core processor and Cortex-R5 dual-core real-time processor, with 256 KB of on-chip memory and 4 GB of Random-Access Memory (RAM) (DDR4). This PS is also equipped with general-purpose I/O and interfaces that establish a connection with the PL part of the board. The connection interfaces provided along with this PS are detailed in Figure 3.2. The resources available on the PL side are summarized in Table 3.1.

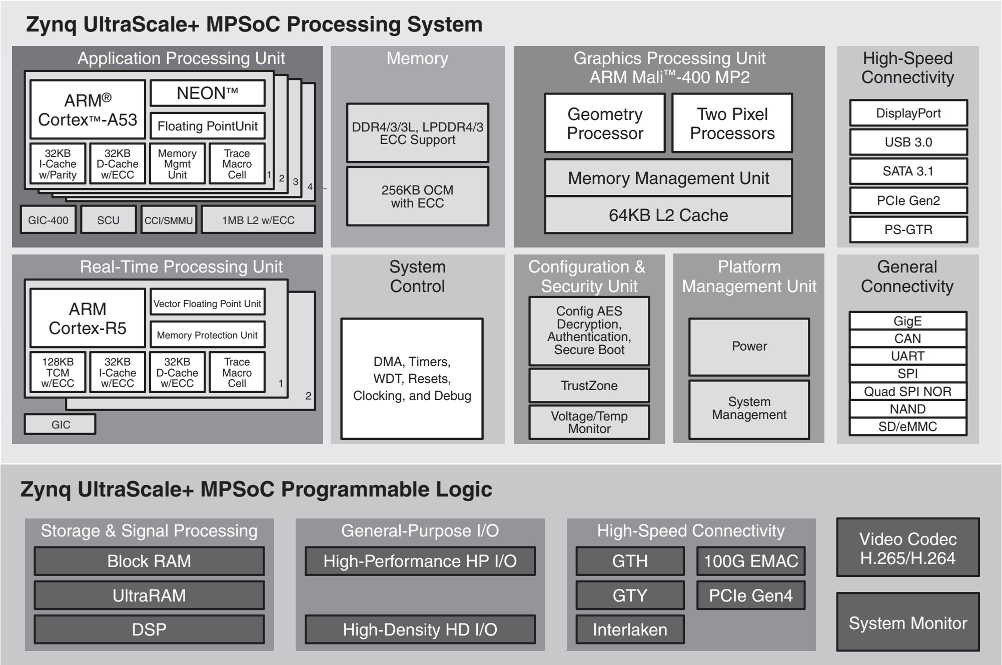


Figure 3.2: Zynq UltraScale+ MPSoC PS and PL Block Diagram [9].

Table 3.1: Zynq UltraScale+ MPSoC ZCU102 Features and Resources [9].

Feature	**Resource Count**
System Logic Cells	599 550
CLB Flip-Flops	548 160
CLB LUTs	274 080
HD banks	5 banks, total of 120 pins
HP banks	4 banks, total of 208 pins
MIO banks	3 banks, total of 78 pins
PS-side GTR 6 Gb/s transceivers	4 PS-GTRs
PL-side GTH 16.3 Gb/s transceivers	24 GTHs
Max. distributed RAM	8.8 Mb
Block RAM Blocks	912
Total block RAM	32.1 Mb
DSP slices	2 520

Additionally, there are a set of tools for deployment assistance. Xilinx Vivado Design Suite and Xilinx Software Development Kit (SDK) are Integrated Development Environments (IDEs) of Xilinx for hardware and software development, respectively. The Xilinx Vivado Design Suite is a software suite for the analysis and synthesis of HDL designs. This IDE even provides an IP library, with a bunch of soft cores which can be mixed with or be part of user-made IP cores. The Xilinx SDK enables the development of embedded applications for the PS part of the board.

Besides the substantial set of resources available shown in the MPSoC diagram, there are a few crucial interfaces for the PS-PL communication, which allows not only data interchanging with memory but also eventual reconfiguration for IP cores.

3.1.1.1 Master Interfaces PS-PL

There are three AXI High Performance Master PS-PL interfaces allowed in the SoC. Each interface can be configured to operate in 32-bit, 64-bit, and 128-bit data widths [44].

- AXI HPM0 FPD and AXI HPM1 FPD - Two high performance master interfaces for full-power domain.

- AXI HPM0 LPD - High performance master for low-power domain.

Each slave interface supports 32-bit, 64-bit, and 128-bit data widths [44].

3.1.1.2 Slave Interfaces PS-PL

- AXI HPC0 FPD, AXI HPC1 FPD - Two high performance coherent slave interfaces for full-power domain.

- AXI HP0 FPD, AXI HP1 FPD, AXI HP2 FPD, AXI HP3 FPD - Four high performance slave interfaces for full-power domain.

- AXI LPD - One slave interface for low-power domain.

- AXI ACP - One Accelerator Coherency Port slave interface that can be connected to a DMA engine or a non-cached coherent master.

- AXI ACE - One AXI Coherency Extension slave interface.

3.1.2 AMBA Advanced eXtensible Interface

Advanced Microcontroller Bus Architecture (AMBA) AXI protocol is an open-standard introduced by ARM and its scope aims the design of high-performance and high-frequency systems for data interchanging [13]. Some key features that makes of AXI a reliable protocol are:

- Support for unaligned data transfers by using byte strobes signal;
- Write and read address validation before the data transaction;
- Allows transmission of data in burst mode, in which only the start address is needed for up to 256 or other power of two (2^n) number of words being transferred in a row;
- Read and write channels are independent [13].

Figure 3.3 represents the data interchanging timing order inherent to AXI4-Memory Mapped read and write operations. As can be observed, both operations occur in independent buses. Both Read and Write channels are independent of each other, they have not only different channels but also independency in the protocol level. Moreover, the Figure 3.3 pretends to represent the data interchanging timing order of each sub-channels of Read and Write channels.

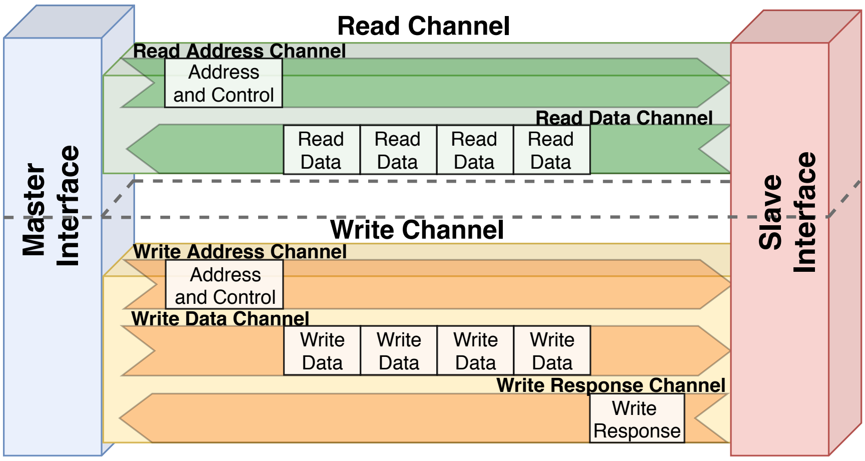


Figure 3.3: Timing diagram for AXI4-Memory Mapped read and write operations.

3.1.2.1 AXI4-Memory Mapped

AXI4 is a data transaction protocol designed for memory mapped interfaces. It allows to burst up to 256 data transfer cycles with a single address phase, both for read and write operations. Each AXI interface uses a single clock signal (*ACLK*). The data is accepted or transferred on the rising edge. The reset signal in the AXI (*ARESETn*) takes effect driven in low. Whereas the reset assertion may happen asynchronously, deassertion must occur synchronously with the rising edge of *ACLK* [13].

3.1.2.1.1 Read Channel

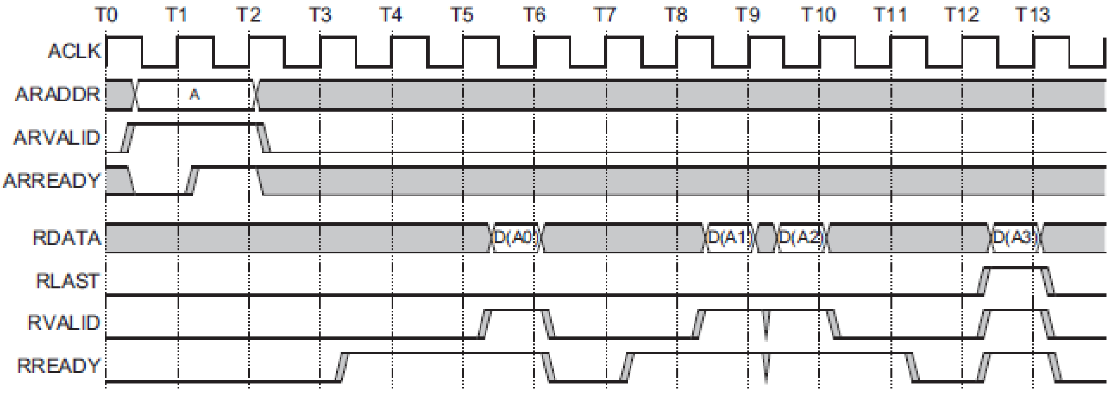


Figure 3.4: AXI4 Read channel timing diagram [10].

According to Figure 3.4, the reading address is accepted by the slave only when *ARVALID* and *ARREADY* are set high. The former signal indicates whether the address issued by the master is valid, while the latter indicates whether the slave is ready to accept a read address. As expected, a read address is only accepted when the slave is ready to accept it and the master issues a valid one. In Table 3.2 are listed all the signals required for address acceptance.

Table 3.2: AXI4 Read address channel signals [13].

Signal	**Description**	**Source**	**Required?**	**AXI4-Lite?**
ARID[3:0]	Read address ID	Master	Optional	No
ARADDR[$a-1$:0]	Read address	Master	Required	Yes
ARLEN[3:0]	Burst length	Master	Optional	No
ARSIZE[2:0]	Burst size	Master	Optional	No
ARLOCK[1:0]	Lock type	Master	Optional	No
ARCACHE[3:0]	Cache type	Master	Optional	Yes
ARVALID	Read address valid	Master	Required	Yes
ARREADY	Read address ready	Slave	Required	Yes

After the address being accepted by the slave, the master starts reading the data issued by the slave. The process moves forward only when *RREADY* and *RVALID* are set high. While *RVALID* is set by the master to inform the slave of the existence of valid data in the channel, *RREADY* is driven by the salve to inform the master it is ready to receive new data. Although *RLAST* is optional, its use is highly recommended for detecting whenever misalignment between master and slave occurs. Table 3.3 shows the signals needed for the read data stage [45].

Table 3.3: AXI4 Read data channel signals [13].

Signal	**Description**	**Source**	**Required?**	**AXI4-Lite?**
RID[3:0]	Read data ID	Slave	Optional	No
RDATA[$n-1$:0]	Read address	Slave	Required	Yes
RRESP[1:0]	Burst length	Slave	Optional	No
RLAST	Read last	Slave	Optional	No
RUSER	Read User	Slave	Optional	No
RVALID	Read valid	Slave	Required	Yes
RREADY	Read ready	Master	Required	Yes

3.1.2.1.2 Write Channel

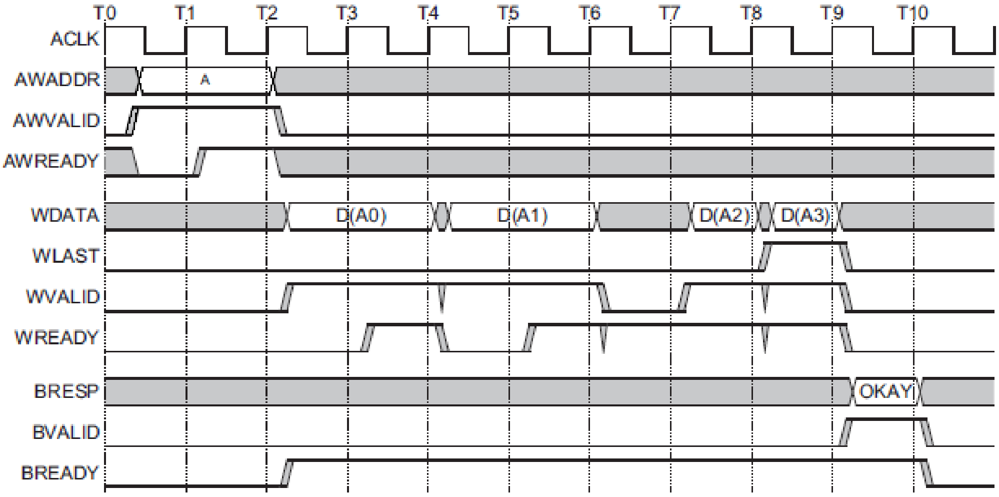


Figure 3.5: AXI4 Write channel timing diagram [10].

The write channel (Figure 3.5) acts similarly to the read channel. The difference resides in the feedback required in the write channel through the write response channel. In this particular stage of the communication, the slave signals the master the success or not of the operation by *BRESP*. The write response is only accepted when both *BVALID* and *BREADY* are set high. The Write Channel is divided in three different sub-channels that communicate in different stages. Similarly to the read address channel, a write address is only accepted by the slave when the *AWVALID* and *AWREADY* signals are driven high. The remaining signals used to validate an address are detailed in Table 3.4.

Table 3.4: AXI4 Write address channel signals [13].

Signal	**Description**	**Source**	**Required?**	**AXI4-Lite**
AWID[3:0]	Write address ID	Master	Optional	No
AWADDR[$a - 1$:0]	Write address	Master	Required	Yes
AWLEN[3:0]	Burst length	Master	Optional	Yes
AWSIZE[2:0]	Burst size	Master	Optional	No
AWLOCK[1:0]	Lock type	Master	Optional	No
AWCACHE[3:0]	Cache type	Master	Optional	Yes
AWVALID	Write address valid	Master	Required	Yes
AWREADY	Write address ready	Slave	Required	Yes

In the write data stage, is where the data is transferred from the master to the slave upon the *AWLEN*. In the last data transaction, analogously to the read data channel, the *WTLAST* is asserted high. In case that *WSTRB* is not set all bits high, only the N least significant bytes shall be passed to the memory in each instant [13]. The signals inherent to the data transaction stage are detailed in Table 3.5.

Table 3.5: AXI4 Write data channel signals [13].

Signal	**Description**	**Source**	**Required?**	**AXI4 Lite?**
WDATA[$n - 1$:0]	Write data	Master	Optional	Yes
WSTRB[3:0]	Write strobes	Master	Required	Yes
WLAST	Write last	Master	Optional	No
WVALID	Write valid	Master	Required	Yes
WREADY	Write ready	Slave	Required	Yes

Finally, the write response provides the master a feedback from the slave of the transaction state. Only the *BRESP* is optional in the protocol. Table 3.6 details the signals present in the write response channel.

Table 3.6: AXI4 Write response channel signals [13].

Signal	**Description**	**Source**	**Required?**	**AXI4-Lite?**
BID[3:0]	Response ID	Slave	Required	No
BRESP[1:0]	Write response	Slave	Optional	Yes
BVALID	Write response valid	Slave	Required	Yes
BREADY	Response ready	Master	Required	Yes

3.1.2.2 AXI4-Stream

AXI4-Stream does not require an address phase and it also allows unlimited data burst size. AXI4-Stream does not have an address since it is not intended for memory-mapped purposes [45]. Even though *TREADY* is optional, it is highly recommended for ensure that the slave is ready for receive data and that it was also accepted [45]. Both *TLAST* and *TUSER* signals, detailed in Table 3.7, are optional but sometimes it may be a useful, depending on the system, to inform the slave the occurrence of an event.

Table 3.7: AXI4-Stream channel signals [13].

Signal	**Description**	**Source**	**Required?**
TVALID	Transfer valid	Master	Required
TREADY	Transfer ready	Slave	Optional
TDATA[$v - 1$:0]	Transfer data	Master	Required
TSTRB[3:0]	Transfer strobes	Master	Optional
TKEEP[3:0]	Valid bytes	Master	Optional
TLAST	Response ready	Master	Optional
TID[$i - 1$:0]	Response ready	Master	Optional
TDEST[$d - 1$:0]	Response ready	Master	Optional
TUSER[$u - 1$:0]	Response ready	Master	Optional

3.1.2.3 AXI4-Stream Video

AXI4-Stream Video is a variant of the AXI4-Stream protocol designed for video transmission purposes. When compared to the basic protocol, AXI4-Stream Video uses two additional signals. Whereas the *TLAST* signal notifies the slave that the current data being transmitted is the end of a line, the *TUSER* serves to inform the slave about the start of a new frame. Both signals give some information that may be important to detect when errors may occur, for instance, when the slave is tracking the number of pixels received from master and it does not match with the pre-configured data.

3.1.2.4 AXI4-Lite

AXI4-Lite is a light-weight, single transaction for memory-mapped purpose. It has a simple interface and a small logic footprint. [45]. In Tables 3.2, 3.3, 3.4, and 3.5 are shown the signals used by AXI4-Lite. The protocol acts the same way as described for the write and read channels of AXI4-Memory Mapped, however, with a limit of one transaction per address phase.

3.2 IoT Video-Card Architecture

The MOG WALL SCREEN project, in which this MSc thesis is inserted, has already a predefined general architecture defined in Figure 3.6. This figure already contemplates some residual changes that were added during the deployment of this work. This IoT Video-Card has been designed to accomplish 8k video acquisition and reproduction in real-time. Besides the demand for a video-card ready for reconfigurable video characteristics (resolution, fps) on-the-fly, there is also a requirement for deployment of reconfigurable hardware to enable and ease the process of future system updates. Next it will be given an overview of the general architecture of the video-card, in which it will be explained the aim for each core. Then, this thesis will be focusing on the aim for this thesis, the Memory Manager and H.264 Encoder.

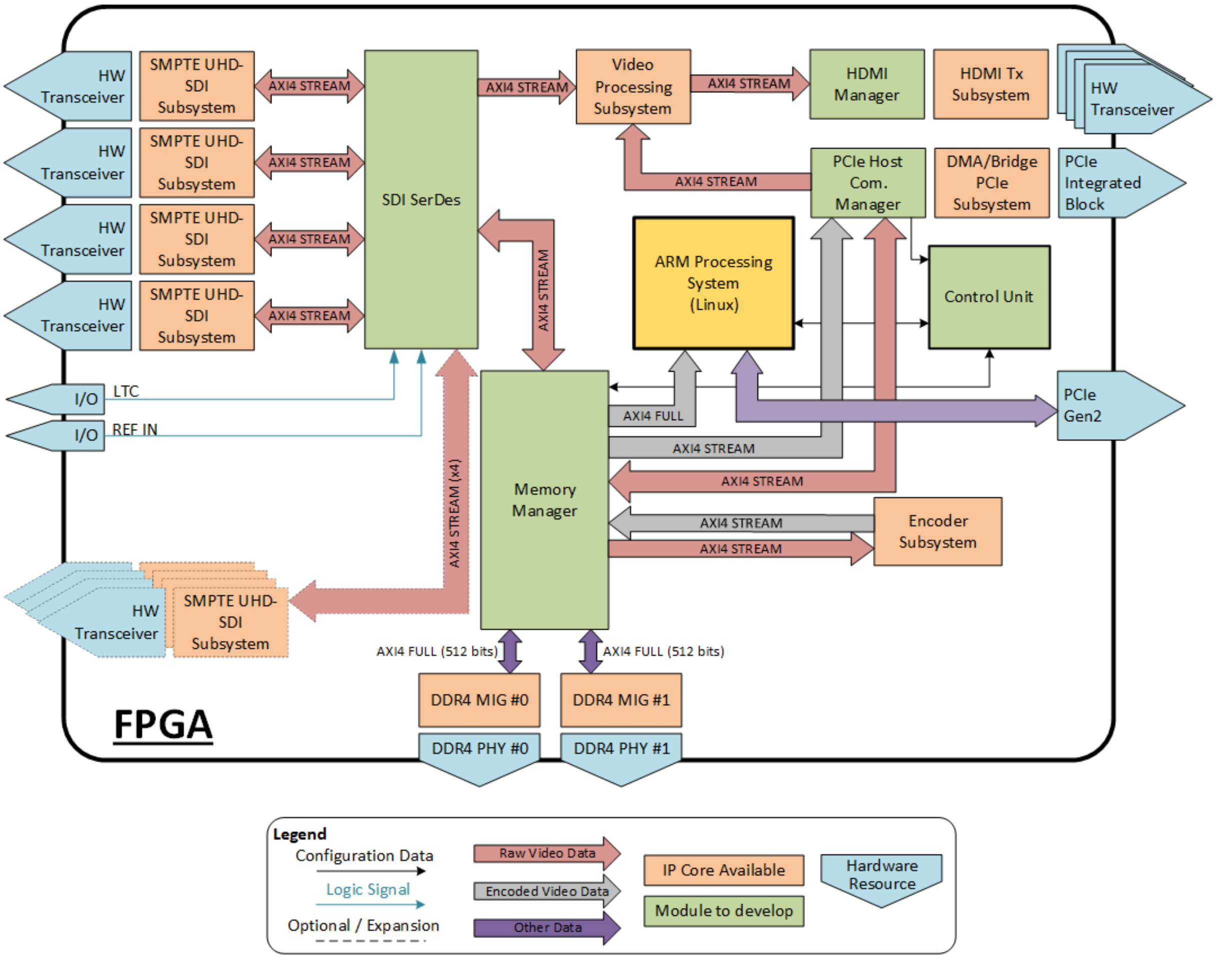


Figure 3.6: HDL General Architecture of the Reconfigurable Video-Card.

SMPTE UHD-SDI Subsystem

This IP core is available from *Xilinx* and is responsible for the interaction between the remaining video-card IP cores and the hardware transceivers, which support the transfer rates required to carry video up to 8K resolution at 30 fps.

SDI SerDes

The SDI Serializer Deserializer is responsible for synchronizing the streams coming from the four IP core SMPTE UHD-SDI instances. The total sum of video streams is 64, 16 per channel. The product of this module is a stream containing the video content from each one of the 16 channels. This stream will be redirected to Memory Manager, which is in charge of performing memory accesses.

Memory Manager

The memory management is performed by a Memory Manager, whose aim is to manage each memory access (read and write) as well as to provide a register for both control and status purpose. It is also responsible for error detection. Additionally, the Memory Manager is in charge of deciding where will be saved each raw and encoded frame, according to a pre-configured user parameter.

DDR4 MIG

Xilinx IP core that allows the interface between a FPGA IP core and the physical layers of a DDR3 or DDR4 memory over the AXI4-Memory Mapped bus.

H.264 Encoder

The H.264 Encoder is in charge of encoding video in YUV 4:2:0 color space 8-bit. Once the video present in memory is YUYV 4:2:2 color space 10-bit, this module is responsible for color space conversion as well. The video is provided by the Memory Manager by AXI4-Stream and it is transferred again to this core in encoded video format.

Control Unit

This element is responsible for the communication and configuration of each core presented in the FPGA, according to the configured user settings, in order to assure the correct operation of the Video-Card. This communication is performed either from PCIe as from the PS. Additionally, this module is also in charge of gather statistics from the video-card and to guarantee that they may be always accessible.

ARM Processing System

This module is intended to run a Linux system responsible for cloud and app communications through the default TCP/IP communication stack already included in this operating system. This module will also be able to access HDL modules through memory-mapped registers.

HDMI Manager

Due to many possibilities of the input video format and the specific requirements of some of the IP cores that will be used, this component is responsible for properly converting the input video to a specific element to interpret the video.

PCIe Host Manager

The demand for a PCIe in this project emerges from the requirement of transferring video content from the FPGA to a common computer for later processing or publishing on the internet. This module is in charge of receiving the configurations to be delivered to the Control Unit, presented in the FPGA, and for the synchronization between a Windows driver and the FPGA, for video content transfer. This module works analogously to a bridge for communication with the driver.

3.3 Memory Manager and H.264 Encoder Dataflow Overview

Before we go deep into the design of each core, it is important to analyze the system overview of this specific project part. Figure 3.7 depicts an architecture overview in which the Memory Manager will be receiving raw video streaming by a source properly configured to transmit video content. The Memory Manager will be receiving both raw video streaming and encoded video data and put it into an external RAM. Additionally, the Memory Manager will be providing the raw video outside of the core by reading a pre-configured memory space through AXI4-Memory Mapped. Whereas the Memory Manager is in charge of managing the memory, the H.264 Encoder will be encoding the data and sending it back to Memory Manager. Both cores need to provide a control/configuration and status register.

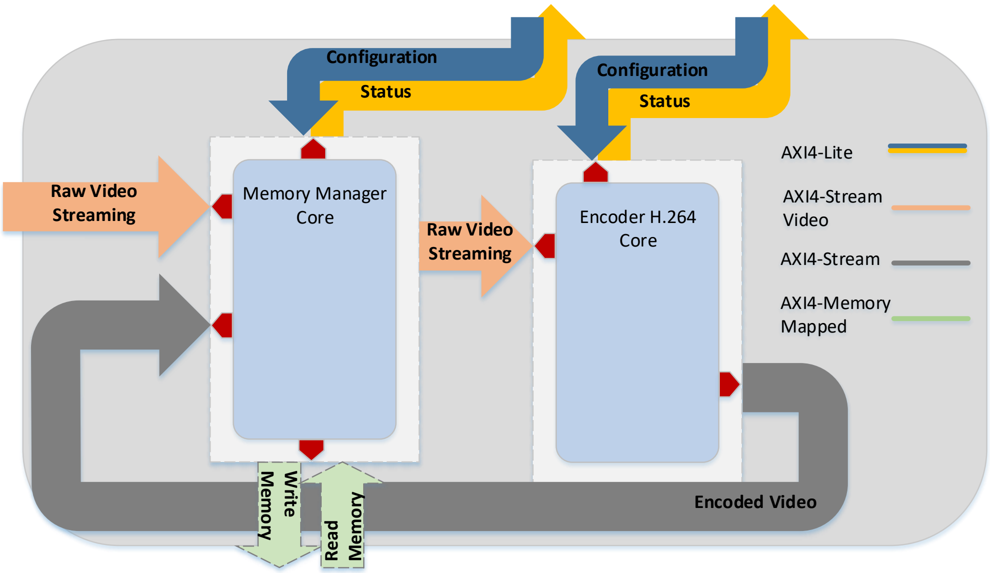


Figure 3.7: Dataflow of the Memory Manager and H.264 Encoder.

As it is represented in Figure 3.7, the system will be using the AXI4: AXI4-Memory Mapped, AXI4-Stream and Stream Video, and AXI4-Lite. AXI4 protocol seems to be the very best option because of the peripherals protocol already available on the board as well as the Commercial-Off-The-Shelf (COTS) provided by *Xilinx* for this protocol. Therefore, the using of AXI4 may reduce the engineering effort for developing the required system.

3.4 Memory Manager

The Memory Manager is the element responsible for managing and organizing the memory accesses as well as to decide where will be saved the data coming from the video source uninterruptible in streaming mode. Since every data interchanging shall be passed through this core, it will be in charge of managing the complexity between different memory accesses. To maintain the integrity of the video present in the memory, it is not supposed the core to lose data. Therefore, there will be a requirement for status register to allow the system to know when data may have been lost.

It was decided by an external scope of this project that the video source will be transmitting video content at 300 MHz in YUV 4:2:2, interleaved format, with 10-bit per color component. The system will be taking advantage of the quad-link 12G SDI and it will be delivering 256-bit video in streaming mode to the memory manager. This core needs to manage the data received and save it in an external RAM. Since the RAM is byte-addressable,the design of this IP core faced two options:

1. Each color component is saved in memory consecutively, in which each color occupies only 10-bit with no padding, as represented in Figure 3.8. It is also called as standard V210;

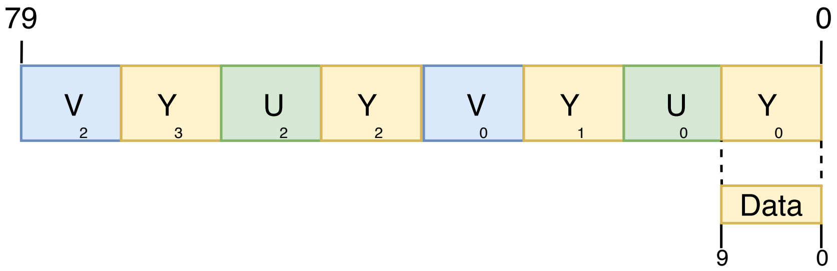


Figure 3.8: Stream with four pixels structured in V210 format.

2. Each color component is saved in memory byte-aligned. This option would require 16-bit (2 bytes) for each color component, as shown in Figure 3.9. This is the standard YUYV.

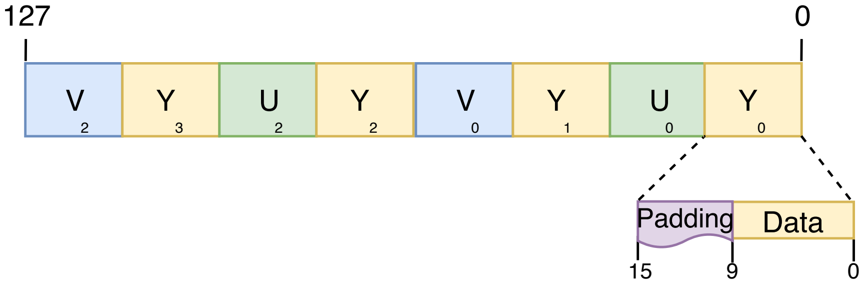


Figure 3.9: Stream with four pixels structured in YUYV format.

Both options have some pros and cons about them. Option 1 would allow a full memory utilization, and consequently, complete throughput exploit. On the other hand, it would require extra processing capacity for the video player.

For option 2, the system would be using 10-bit with the color component information and the remaining 6-bit would be filled up with padding that would be ignored by the video player. This solution does not require the video player for extra processing. However, it means that the system would be facing a loss of 37.5% of useful memory. The throughput would be also affected negatively in the same percentage.

Since adding extra processing for the video player is highly discouraged, it was decided to design a system based on the second option once the throughput is still achievable for delivering 8k at 60fps as will be shown next.

First of all, it should be checked if it is possible to deal with 8k (8192x4320) at 60fps by transmitting the video through SDI 12G quad-link of 64-bit each. This means that the final output from this part will be 256-bit at 300 MHz. Since the system is dealing with YUV 4:2:2 color space, every two pixels would require 4 color components (two for Y, one for U, and 1 for V). A 256-bit length stream implies the transmission of 8 pixels per clock.

- $8192 \times 4320 = 35,389,440$ pixels per frame;
- $35,389,440/8 = 4,423,680$ number of clocks per frame;
- $60 \times 4,423,680 = 265\ 420\ 800$ minimum frequency.

The Memory Manager is in charge of synchronizing the write and read operations and guarantee that a frame is only transmitted when it has already been completely saved in memory. Additionally, it is needed to guarantee that this process is continuous, which means that it is not supposed to transmit a frame that is still being received since this would condition the throughput of the system because we would be wasting a resource without being using it due to a possible delay of data availability.

This core is intended to be configurable due to a variable frame resolution demand from the source. To accomplish that there is a configuration subsystem present in the Memory Manager core that is responsible for both control and status updates. This subsystem is addressed by AXI4-Lite to interface with both elements of the Memory Manager, one responsible for raw video and other for encoded video. A demand for reconfigurable memory space is also required since this resource may be shared by many peripherals, therefore, it is important to prevent the overlapping of memory regions.

The Memory Manager is designed to be as much as possible self-sufficient handling the data coming. This mechanism is necessary to prevent eventual errors that may occur during the system run-time to avoid compromising the system. Therefore, there is a demand for design a system capable of not only detect errors but also discard non-valid data, so the system may not halt due to an error and keep running uninterruptible.

Since the reading of frames from memory is dependent on the write subsystem, it is required the existence of a read/write control upon the content already transferred to memory.

The Memory Manager will be the element responsible to put the video content in memory and it will be using the memory resource massively. Once there is a demand for optimizing as much as possible the memory accesses, there will be a demand for buffering the data so it can only write the content in memory when the Memory Manager has enough data to be sent continuously to external RAM. Since this core will be reading video content from memory and streaming it to another component, there is a demand to provide video signals for end of line and start of frame as explained in [46].

In Figure 3.10 is presented the general architecture for the Memory Manager IP core. Each subsystem as well as some important elements to better understand the solutions behind this core will be explained in the next subsections.

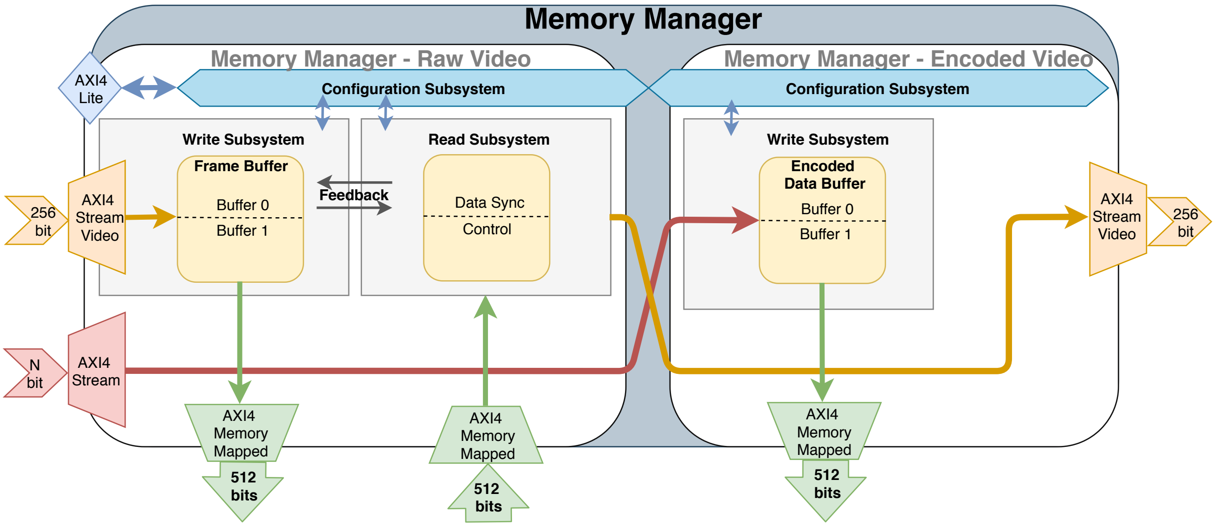


Figure 3.10: Memory Manager IP core General Architecture.

3.4.1 Configuration Subsystem

The configuration subsystem is intended to provide a way for controlling the actions of the Memory Manager as well as to allow the system to access the status register of this core. The AXI4-Lite is the protocol used to communicate between a master, that will be commanding the read/write operations, and a slave, that will be demanded. This protocol is enough since the volume of data transferred in these operations is not significant. The video memory region cannot be accessed by this data bus.

In Table 3.8 is presented the address offset and its respective name of registers available in Memory Manager. These registers allows to change the parameters for configuring the core as well as for checking status. The registers shown in the Table 3.8 have a default value of 0×00000000. Table 3.9 presents in detail the purpose of each register as well as the bit structure of each one.

Table 3.8: Memory Manager register address map.

Register Name	**Address Offset**	**Permission**
Raw video control register	0×00000000	Read/Write
Raw video start memory address	0×00000008	Read/Write
Raw video start memory address (Bytes)	0×00000010	Read/Write
Horizontal resolution (Bytes)	0×00000018	Read/Write
Vertical resolution	0×00000020	Read/Write
Raw video status register	0×00000028	Read Only
Encoded video control register	0×00000030	Read/Write
Encoded video start memory address	0×00000038	Read/Write
Encoded video memory size	0×00000040	Read/Write
Encoded video status register	0×00000048	Read/Write
Encoded video last memory data saved	0×00000050	Read Only

Table 3.9: Raw video control register of Memory Manager (00h Offset).

Bit	**Name/Purpose**	**Function**
0	Restart Memory Manager	0 = No effect 1 = Restarts Memory Manager
1	Write subsystem enable	0 = Write subsystem disabled 1 = Write subsystem enable
2	Read subsystem enable	0 = Read subsystem disabled 1 = Read subsystem enable
3	Halt enable - Raw video	0 = System does not halt with errors 1 = System halts with errors
63:4	Reserved	Reserved

Since this is a system for streaming purposes in which the data usually flows uninterruptible, there may be a demand for configuring the system and receiving data at the same time with no errors. To accomplish that, it was implemented a system that would allow the configuration of the registers to be effected only when there is a restart in the core. This approach avoids the demand for stopping the system during this process in order not to cause errors or unpredictable behavior that may cause unknown issues. Therefore, the new configured parameters to be used by the Memory Manager will only be applied when the *Restart Memory Manager* bit (0) of the *Raw video control register* is set high. This bit can be set by software through AXI4-Lite but it is immediately cleared by hardware, which means that after the writing in the register if we would try to read it, that bit will be already driven low. It works like a pulse in which writing a '1' on it initiates the pulse.

Both *Write subsystem enable* and *Read subsystem enable* only enables or disables each subsystem. This feature may be useful during the running time or even to conciliate the read/write process with other cores. The bit *Halt allow - Raw video* is useful to inform the application when there is an error associated to the writing process the core should stop or only notify it by updating the status register.

Raw video start memory address (0× 08) and *Raw video memory size (0× 10)*

Both registers serve the purpose of configuring the memory region in which the Memory Manager will be writing and reading from external memory. It is intended to perform a circular buffer in which after the last address is reached during the reading/writing process the pointed value for the next transaction is rotated and starts again from the initial address.

Horizontal resolution (0× 18) and *Vertical resolution (0× 20)*

As the name implies, this register is intended to configure the frame size expected to be saved in memory and to the Read subsystem transfer video according to the configured parameters through AXI4-Stream Video. Moreover, both registers are useful to calculate the next frame address offset during the writing process for error controlling as shown in Figure 3.12.

Raw video status register

In Table 3.10 is exhibited in detail the *Raw video status register* of Memory Manager, which permits to access relevant information happening in the core.

Table 3.10: Raw video status register of Memory Manager (28h Offset).

Bit	Name/Purpose	Description
0	Memory Manager halt	0 = Memory Manager running 1 = Memory Manager halted
1	EOL early error	0 = No EOL early error 1 = EOL early error
2	EOL late error	0 = No EOL late error 1 = EOL late Error
3	SOF error	0 = No SOF error 1 = SOF error
4	Unwritten data	0 = No Unwritten data error 1 = Unwritten data error
63:5	Reserved	Reserved

The *halt* bit register allows checking whether the core is running or not. This bit can only be driven high by hardware when the bit 3 (*Halt enable - Raw video*) of *Raw video control register* is set high. Additionally, this bit is set '1' when there are errors (*EOL Early, EOL Late*, or *SOF error*) that may have happened during the video receiving.

EOL early error is set high when the AXI4-Stream Video signals have notified the end of line earlier than expected. This problem may have been caused due to a non-matched configuration between the video source and the Memory Manager. Similarly to *EOL early error*, the *EOL late error* notifies if the end of line signal from the video source is received later than expected. The *SOF error* is driven high when there is a problem in the vertical length of the frame, which is larger or shorter than expected, according to the pre-configured registers. Finally, the *Unwritten data* serves to notify when the system was not able to deal with the volume of data read and written due to a possible congestion in the slave because of many memory accesses. This is a piece of useful information that, for instance, the VDMA provided from *Xilinx* does not support. The VDMA would only be setting the AXI4-Stream Video *ready* signal low, which would cause data lost when the streaming process cannot be stopped.

Table 3.11: Encoded video control register of Memory Manager (30h Offset).

Bit	Name/Purpose	Function
0	Restart Memory Manager	0 = No effect 1 = Restarts Memory Manager
1	Encoding subsystem enable	0 = Encoding subsystem disable 1 = Encoding subsystem enable
63:2	Reserved	Reserved

The *Restart Memory Manager* bit of the control register for encoded video (Table 3.11) enables the restart of the core with the new parameters configured previously. The bit 1 of this register is used for enabling or disabling the encoding of video to be saved back in memory in form of encoded video.

Table 3.12: Encoded video status register of Memory Manager (48h Offset).

Bit	Name/Purpose	Description
0	Unwritten data	0 = No Unwritten data error 1 = Unwritten data error
63:1	Reserved	Reserved

Since this part of the memory manager deals with video derived from frames previously validated in memory, it does not require much status information, as can be observed in Table 3.12. If there would be some frame error, the raw video component of the Memory Manager would already be solved the issue by jumping to the correct address, to keep the frames aligned, or by rewriting the memory in which the error occurred.

Encoded video last memory data saved (0×50)

This register allows the system to know always where is being written the last frame of encoded video.

3.4.2 Write Subsystem

There are three major elements in this subsystem, namely two interfaces, Slave AXI4-Stream Video and Master AXI4-Memory Mapped, and a top module, in which is managed these interfaces. It is responsible for tracking of the data received, error control, buffering, main state machine (represented by a diagram in Figure 3.11), and more.

In the Slave AXI4-Stream Video, there is a demand to ensure the protocol compliance and to interface with a master in which the slave must be always ready to receive the data to guarantee there is no data loss in the process. The data is saved in circular buffers, controlled by the top module.

The top module is responsible for monitoring and initiating every operation happening in both Slave and Master interfaces as well as for tracking both data received and transferred. Frame size error control is performed here by tracking the data and signals received to compare them with a pre-configured parameters of the register space.

A Master AXI4-Memory Mapped is needed for interfacing with a slave that will be receiving the video content through the AXI4 data bus and put it in memory whenever the resource is available. This interface should be as optimized as possible for the system to lose less time possible on validations and responses.

There is a simple main state machine presented in Figure 3.11 whose main goal is to control the reading of the data previously buffered in circular buffers and transfer the data through the Master AXI4-Memory Mapped to save in external memory RAM.

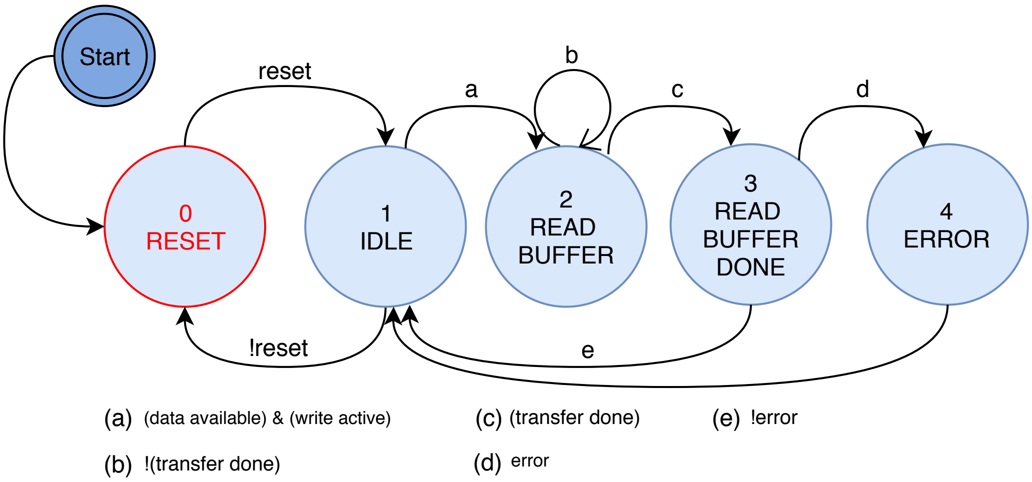


Figure 3.11: Write subsystem of Memory Manager Main State Machine.

- *RESET* - This state, as the name implies, resets the entire hardware of the Memory Manager and takes the system to the initial state as well as the control registers and status. Additionally, both buffers for reading and write are taken to the initial address value.

- *IDLE* - When the subsystem is in this state it is waiting for a buffer to be full to be ready to initiate the transfer.

- *READ BUFFER* - In this state, the subsystem is reading from the currently active buffer until it reaches the last buffer address.
- *READ BUFFER DONE* - When the state machine reaches this state it triggers a pulse that notifies the end of a burst transaction for frame tracking purposes. It evolves immediately to *IDLE* or *ERROR*.
- *ERROR* - If this state is reached, the system may halt, depending on the user configurations, and stays in this state until a restart or a *RESET* of the core is performed.

The buffering mechanism present in the core is actually a double buffering mechanism. This mechanism is designed to avoid race conditions, i.e., the simultaneous read and write of a memory address of a given buffer. If the buffer controller tries to write in a buffer still being read for data transfer by AXI4-Memory Mapped, it may mean the external memory resource is congested, and an error may be released in the status register. If this situation happens, the Memory Manager opts to write the new data coming in the same buffer again, i.e., it will not start writing in the next buffer once it is still being read.

Since the purpose of this core is to be self-sufficient in error recovery, it was designed an algorithm able to deal with the wrong frame size. Then, the system may keep running as if nothing would have happened. Once residual errors may happen during long periods of run time, there is not a demand to stop the system. If the system is able to understand and correct the error, it may keep running. In Figure 3.12 is presented a block diagram for offset control between frames.

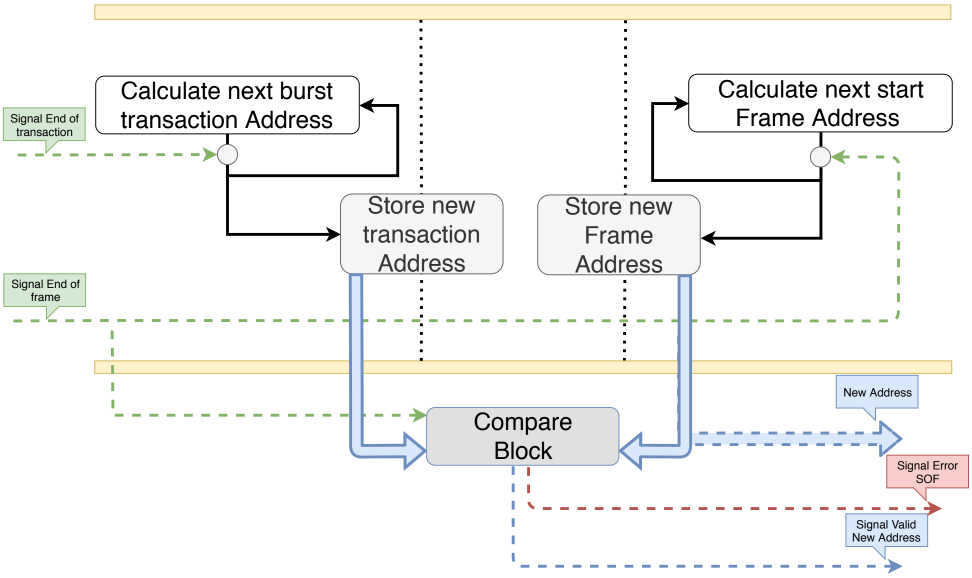

Signal End of transaction
Calculate next burst transaction Address
Store new transaction Address
Store new Frame Address
Calculate next start Frame Address
Signal End of frame
Compare Block
New Address
Signal Error SOF
Signal Valid New Address

Figure 3.12: Algorithm behind frame address offset controlling to prevent misaligned/wrong frames being saved in memory.

By analyzing the Figure 3.12, we can see that this block is receiving two input signals, End of Transaction and End of Frame. In the left part of the Figure, the value is calculated after each burst transaction whereas, in the right, the block is calculating the initial frame address of the next frame. Then, when the End of Frame signal is driven high, the current context of both blocks are compared for checking if the values match. If they match the system continues with its default execution as there is no error. If not, one of two things may happen: (i) The system halts and waits for a restart, or (ii) the offset is corrected and the Memory Manager keeps running. These errors arise due to a fault behavior of the video source and they are supposed to be residual. In such situations, the Memory Manager can correct the error and keep the system running, or it can simply stop. This a trade-off that the Memory Manager delegates to the final user of the video card. For this purpose, the user just needs to configure the *Raw video control register*.

The register to specify the memory size, *Raw video Memory Size*, will be used to calculate the last address of of the external RAM, which is used to reference the moment when memory should be rotated. However, if this value is not divisible by the frame size in bytes it would be a problem because it is not recommendable that a frame should be split in memory. Moreover, due to the burst transaction, if this value would not be carefully calculated, the end of memory can coincide in the middle of a burst transaction. As it is possible to observe in the example of Figure 3.13, there is enough memory space for at least three frames to be saved in memory. Since the configured memory size resulted in an end address not aligned with a complete frame saved in memory, the mechanism will identify that the N+4 frame cannot be fully saved in the remaining memory. Then, it will rotate immediately after the N+3 frame is saved in memory (at the 0×384000, in this case).

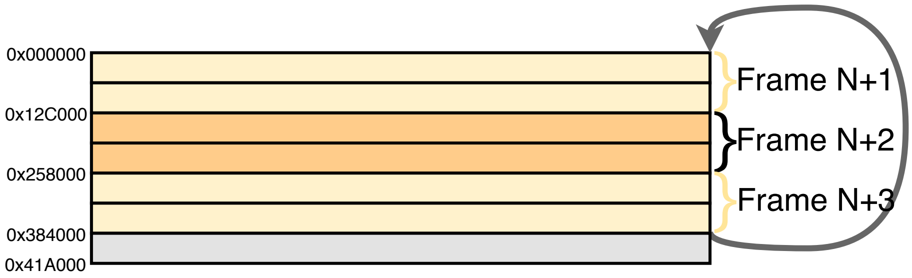


Figure 3.13: Example of buffer rotation when the configured address is not aligned to fill an entire frame YUV 4:2:2 640×480.

This mechanism is also used by the Read subsystem to read video frames from memory.

Encoded Video Write Subsystem

There are only a few differences from the Encoded video write subsystem and the Raw video write subsystem. Now, the reference for the system to decide whenever it should be rotating the address is based on the (frame size (bytes) / 16). According to specification, the H.264 guarantees a minimum of 10× in the compacting rate. Moreover, the original number of bits per color depth is 10, while in the encoder it will be receiving only 8. This saves another 20% of memory size, and also the fact that the YUYV 4:2:2 has 17% more data than the YUV 4:2:0 color format. Then, dividing the frame size by 16 is a secure margin to ensure that each frame may never be parted in memory.

3.4.3 Read Subsystem

This subsystem is in charge of reading the video content previously saved in memory by the Write subsystem. To accomplish that, the Read subsystem is composed by three main elements, two interfaces (Master AXI4-Memory Mapped and a Master AXI4-Stream Video) and a top module, which is in charge of, not only monitoring the data interchange in the interfaces, but also to command the initiation of the reading process from the memory and the immediate streaming of the content saved in memory through AXI4-Stream Video data bus.

There is an evident dependency from this module to the Write subsystem as the Read subsystem needs to track the number of frames saved in memory to know whenever there is enough data to be delivered from memory. Again, as discussed previously in the 3.4.2 Write subsystem subsection, the memory accesses optimization is a requirement since both reading and writing accesses may be delaying the delivering and receiving of data, and as a consequence of that, would be conditioning the delivering of 8k video resolution. Therefore, the same metrics discussed before are applied for this subsystem as well.

In Figure 3.14 is presented a simple state machine for the Memory Manager Read subsystem. Since there is not a demand for buffering mechanism in this subsystem, the state machine is quite simple and the top module should only be tracking the quantity of data saved in memory by the feedback received from the Write subsystem.

- *RESET* - When a reset is performed, the hardware presented in Memory Manager Read subsystem is taken to the initial state, namely, the control registers and status.

- *IDLE* - The subsystem is waiting for the beginning of the reading process of the external RAM through Master AXI4-Memory Mapped.

- *READ MEMORY* - In this state, the subsystem is reading the data from the memory region space specified in the configuration.

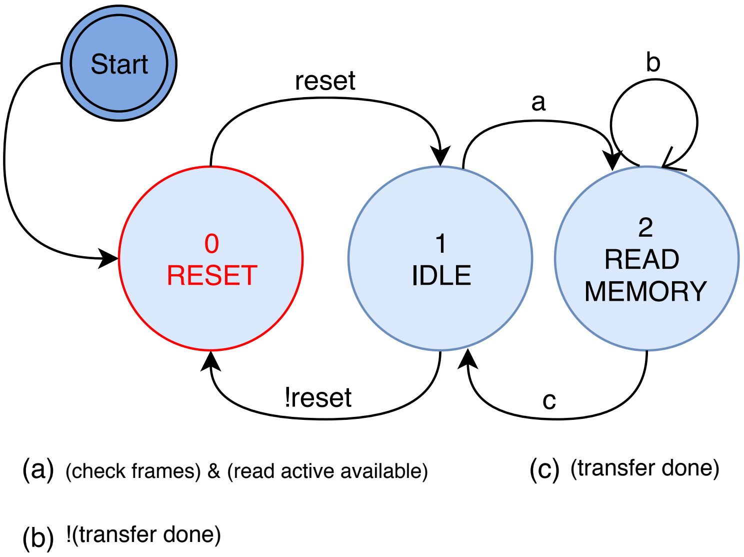


Figure 3.14: Read subsystem of Memory Manager Main state machine.

Whereas in the Master AXI4-Memory Mapped the protocol is reading data, in the Master AXI4-Stream Video it is streaming the video along with the required video signals as approached in the 3.1.2.3 subsection. Additionally, for the Read subsystem to know whenever it is able for streaming video, it is performed a simple hardware block in charge of monitoring the data being written and data being read from memory. It is represented in Figure 3.15. There is a counter responsible for saving the difference number between the quantity of received data and the data already being delivered. With this mechanism, the system can know whenever it is able to start a new transaction from memory. This feedback is performed by both Read and Write subsystems in which whenever each one finishes a data transaction triggers a pulse to update the counter.

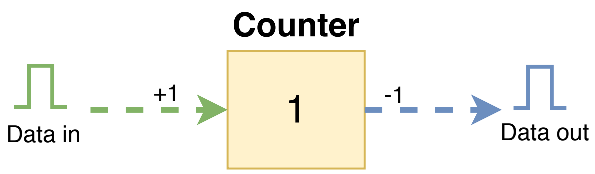


Figure 3.15: Mechanism to count the frames not streamed from memory.

3.5 H.264 Encoder

The H.264 Encoder emerges from the necessity of compacting video. Since there is already an open-source solution available [47], the mission now is to adapt the core to fit in the system as well as to allow an interface to give ordered data to the encoder. Therefore, to maximize the throughput and the memory accesses, there may be a demand for a huge buffering in this core.

As it is already known from the background discussion in section 2.1.3, the encoder shall be receiving the data differently than it is stored from a common streaming video source. Here, the data need to be divided and organized to be delivered in small MB of 16x16, in case of Y color component, and 8x8, in case of UV color component. This encoder is originally designed to receive data in YUV 4:2:0 8-bit per color component, which means that there will be a demand for converting the YUYV 4:2:2 10-bit per color component before the delivering to the H.264 Encoder. Since the memory contains video ordered by each frame whose content has the data organized in pixels from left to right and from top to bottom, the streaming video provided by the Memory Manager is delivered in this way as well. Therefore, there is a necessity to buffer, at least the first sixteen horizontal lines to this subsystem be ready to access the needed data ordered according to the demand from the H.264 Encoder.

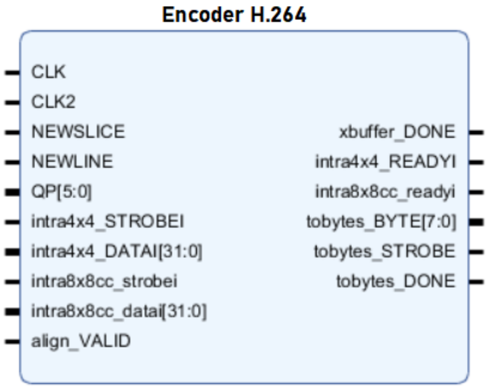


Figure 3.16: H.264 Encoder input and output signals.

In Figure 3.16 is presented the open-source core along with the input and output signals.

- *CLK* (input) - The frequency of this clock should be the same as the frequency of the video-card. Besides being used by the main state machine presented in Figure 3.17, it is also used by the following signals: (i) *NEWSLICE*, (ii) *NEWLINE*, and (iii) *align_VALID*.

- *CLK2* (input) - The frequency of this clock should be twice the frequency of the first clock. It is used for data delivering and for prediction purposes.
- *NEWSLICE* (input) - Works analogously to a reset, in which some prediction data stored in Encoder registers is cleaned.
- *NEWLINE* (input) - This signal serves to inform the Encoder that a new 16x line is starting to be transferred.
- *QP[5:0]* (input) - Quantization parameter, from 0 to 51.
- *intra_4x4_STROBEI* (input) - Validates the data in intra4x4_DATAI.
- *intra4x4_DATAI[31:0]* (input) - Lumen data (Y), 4x8-bit.
- *intra8x8cc_strobei* (input) - Validates the data in intra8x8cc_datai.
- *intra8x8cc_datai[31:0]* (input) - Chroma data (UV), 4x8-bit.
- *align_VALID* (input) - Serves to notify the end of a frame transaction.
- *xbuffer_DONE* (output) - Intra signal, is used as a ready statement for a new line. After some time of not receiving any data this signal notifies that the *xbuffer* has completed all the work buffered. The system need to wait this signal to be set high before *NEWLINE* can be driven high again.
- *intra4x4_READYI* (output) - Signals that the Encoder is ready to receive new lumen data (Y).
- *intra8x8cc_readyi* (output) - Signals that the Encoder is ready to receive new chroma data (UV).
- *tobytes_BYTE[7:0]* (output) - Data codded.
- *tobytes_STROBE* (output) - Validates the tobytes_BYTE.
- *tobytes_DONE* (output) - Signals that the Encoder finished a encoding of an entire frame.

Again, as specified in the previous design of the Memory Manager (subsection 3.4) it will be used a dual circular buffering system for storing temporary data. Since this component is intended to be encoding video with different resolutions, there is a demand for design a system able to be addressing MBs whose resolution frame has variable size. In this case, we need to find an efficient algorithm for addressing the buffered data efficiently and dynamically to be delivered as specified in the background, section 2.1.3. This algorithm is shown and explained in subsection 3.5.2.

One particular characteristic of this encoder is the fact that it requires two different frequency clock to work. To overcome this design issue, it was performed three different state machines with dependency from each other. They are represented in Figures 3.17 (main state machine), 3.18 and 3.19 (two secondary state machines for Y color component and UV color component).

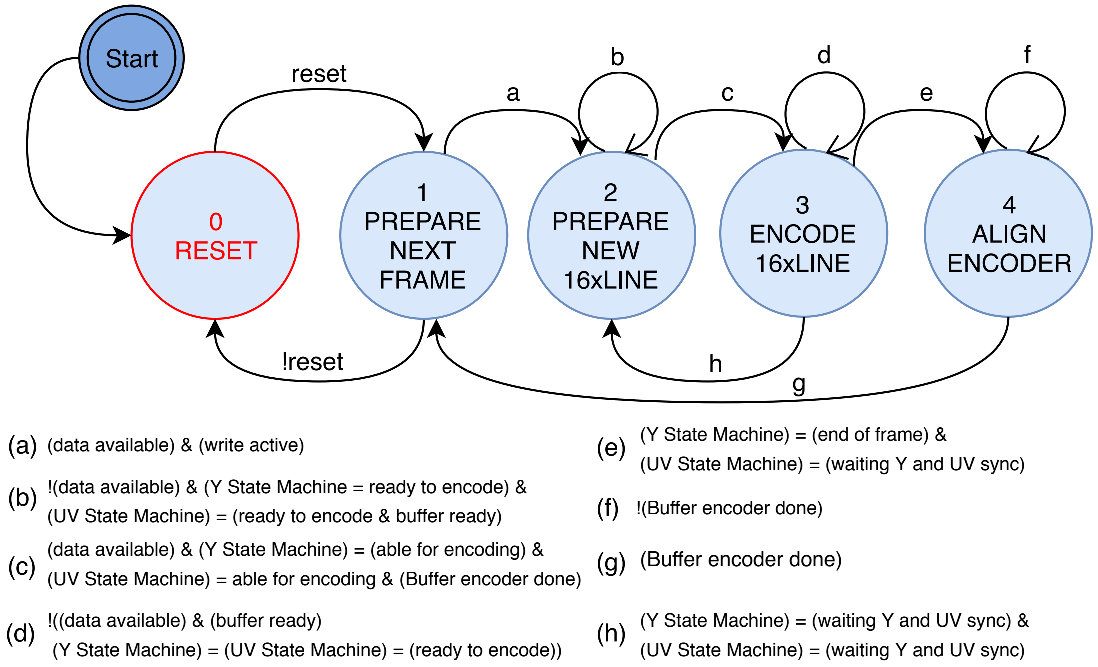

Start
reset
a
b
c
d
e
f
0
RESET
1
PREPARE
NEXT
FRAME
2
PREPARE
NEW
16xLINE
3
ENCODE
16xLINE
4
ALIGN
ENCODER
!reset
h
g
(a) (data available) & (write active)
(b) !(data available) & (Y State Machine = ready to encode) & (UV State Machine) = (ready to encode & buffer ready)
(c) (data available) & (Y State Machine) = (able for encoding) & (UV State Machine) = able for encoding & (Buffer encoder done)
(d) !((data available) & (buffer ready) (Y State Machine) = (UV State Machine) = (ready to encode))
(e) (Y State Machine) = (end of frame) & (UV State Machine) = (waiting Y and UV sync)
(f) !(Buffer encoder done)
(g) (Buffer encoder done)
(h) (Y State Machine) = (waiting Y and UV sync) & (UV State Machine) = (waiting Y and UV sync)

Figure 3.17: H.264 Encoder - Main state machine.

- *RESET* - When there is a reset, the hardware associated to this state machine is set to the initial value.

- *PREPARE NEXT FRAME* - In this state it is prepared the signals for the encoder to signalize the start of frame. The next state is reached as soon as there is data available in the buffer to be delivered to the H.264 Encoder.

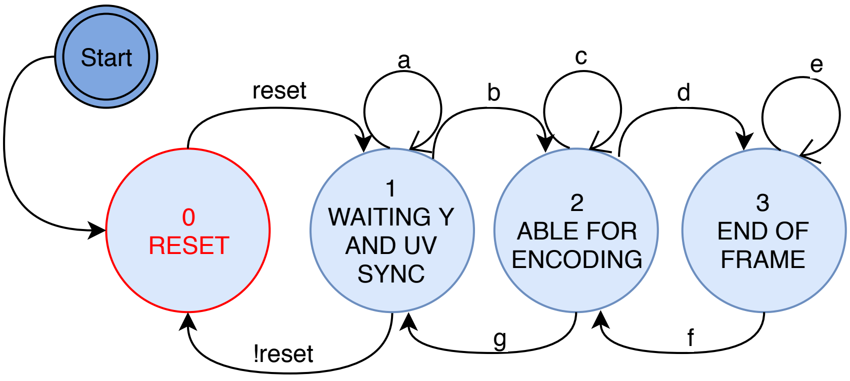
- *PREPARE NEW 16× LINE* - In this state it is prepared the beginning of a new line. Moreover, a new line can only start being encoded after *xbuffer_DONE* is set high and when there are already data available in buffers.

- *ENCODE 16× LINE* - When the system is in this state it is being performed the data addressing algorithm presented in subsection 3.5.2, i.e., it is being transferred the data reordered from the buffers to be delivered to the H.264 core. After this state, the system can evolve to the end of frame codding, or, if there is still data from a frame to be encoded, jump to *PREPARE NEW 16× LINE* again.

- *ALIGN ENCODER* - In this state, the frame was already encoded and we are now waiting for the confirmation that it was successfully encoded.

A triple co-dependent state machine was the solution found to synchronize the signals dependent from the clock and the data being delivered to the encoder in twice the frequency. Since the H.264 needs to keep the coherency between signals dependent from different clock domains, there are two possible methods. (i) Set all the state machines working in the lower clock frequency (it would decrease the performance of the whole encoder system), or (ii) design a solution dependent from three synchronized state machines, working in different clock domains. Some states depend on another state machine with different clock domain. The (ii) guarantees not only the synchronism between different clock domain signals but also a some performance improvement in the whole system.

Once the Y and UV color will be delivered to different channels, and for avoiding non-synchronization between both color components, it was designed two similar co-dependent state machines which are controlling and synchronizing different color components, Y and UV. Both state machines are presented in Figures 3.18 and 3.19.



(a) !((UV State Machine) = (waiting Y and UV sync) & (Main State Machine) = (encode 16xline))

(b) (UV State Machine) = (waiting Y and UV sync) & (Main State Machine) != (encode 16xline)

(c) !(read Y buffer done)

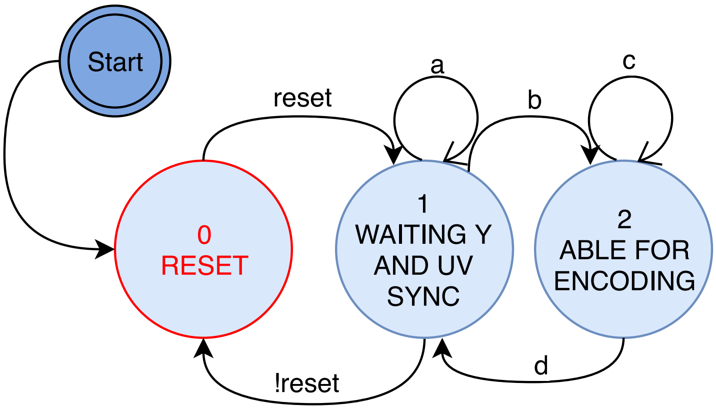
(d) (read Y buffer done) & (end of frame)

(e) (UV State Machine) = (waiting Y/UV sync) & (Main State Machine)!= encode 16xline)

(f) (UV State Machine) = (waiting Y/UV sync) & (Main State Machine) != (encode 16xline)

(g) read Y buffer done &!(end of frame)

Figure 3.18: H.264 Encoder interface Y color component State Machine.



(a) !(UV State Machine = (waiting Y and UV sync)) & (Main State Machine) = (encode 16xline)

(b) (UV State Machine) = (waiting Y/UV sync) & (Main State Machine) != (encode 16xline)

(c) !(read UV buffer done)

(d) (read buffer UV done) & !(end of frame)

Figure 3.19: H.264 Encoder interface UV color component State Machine.

- *RESET* - Again, in this state the hardware is restarted to the initial values.
- *WAITING Y AND UV SYNC* - In this state, the state machine is waiting for synchronization between the other Y/UV state machine to reach this state. When the state machine reaches this state it means the data was already dispatched and it is ready for the next state.
- *ABLE FOR ENCODING* - In this state machine, the system is synchronized and is able to encode video. After each Y/UV color component data being delivered, it may evolve for the next state to wait for synchronization. The state machine for the Y color component may evolve to the *END OF FRAME* once all data have been delivered.
- *END OF FRAME* - This state is only available in the state machine for Y color component. Besides signalizing the end of the frame, it also serves the purpose of synchronization with the main state machine.

3.5.1 Configuration Subsystem

This subsystem exists due to the demand for controlling some parameters for the H.264 Encoder to properly encode the data. In Table 3.13 is presented the address offset as well as the name to access the H.264 Encoder core registers. The default value of each register presented in Table 3.13 is 0×00000000.

Table 3.13: H.264 Encoder register address map.

Register Name	Address Offset	Permission
Control register	0×00000000	Read/Write
Horizontal resolution	0×00000008	Read/Write
Vertical resolution	0×00000010	Read/Write
Frames per second	0×00000018	Read/Write

Figure 3.14 presents the control register for H.264 Encoder. Similarly to the Memory Manager control register, this core have the *Restart bit in the H.264 Encoder control register*. The Restart bit enables the restarting of the system whenever it is required. This bit is cleaned by hardware immediately after being driven high. The second bit of this control registers allows the enable or disable of this core.

Table 3.14: H.264 Encoder control register (0×00).

Bit	Name/Purpose	Function
0	Restart H.264 Encoder core	0 = No effect 1 = Restarts H.264 Encoder core
1	H.264 Encoder core enable	0 = H.264 Encoder core disable 1 = H.264 Encoder core enable
63:2	Reserved	Reserved

Horizontal resolution (0× 08)

Contrary of the Memory Manager that need to receive the horizontal frame length in bytes, in the Encoder, this number is a decimal integer.

Vertical resolution (0× 10)

This register allows the configuration of the vertical frame length for the core to work properly.

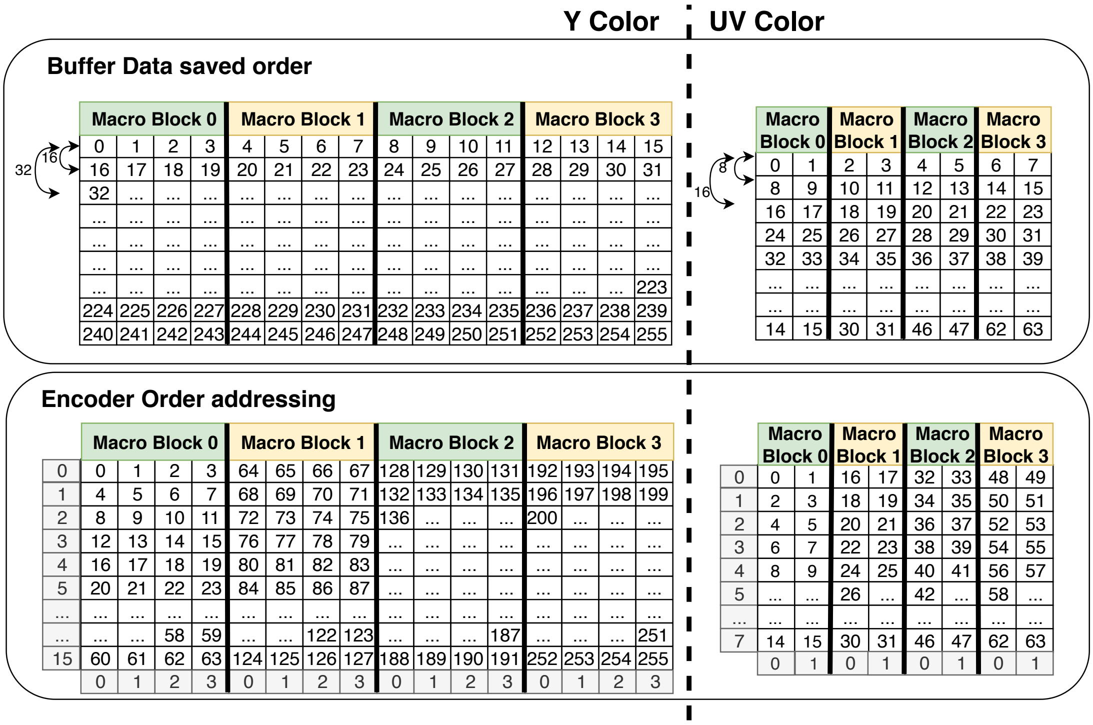
Frames per second (0× 18)

The demand for this register emerges from the necessity for coding the header of the H.264 Encoder as well to make this core self-sufficient on the encoding video content.

3.5.2 Data Acquisition Subsystem

This subsystem is responsible for receiving data through AXI4-Stream Video, which is buffered to be later transmitted according to H.264 Encoder demand. Once the video format stored in memory is YUYV 4:2:2 10-bit per color component, this subsystem is in charge of converting from this format to the YUV 4:2:0 8-bit per color component. Therefore, the data from the AXI4 bus shall be parsed. Then, the data is directed to the correspondent color component buffer, Y or UV. Additionally, the increment of buffers addresses shall be performed by this subsystem as well.

Mechanism of data delivering to H.264 Encoder



Macro Block 0				Macro Block 1				Macro Block 2				Macro Block 3			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Macro Block 0		Macro Block 1		Macro Block 2		Macro Block 3	
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...
14	15	30	31	46	47	62	63

	Macro Block 0				Macro Block 1				Macro Block 2				Macro Block 3			
0	0	1	2	3	64	65	66	67	128	129	130	131	192	193	194	195
1	4	5	6	7	68	69	70	71	132	133	134	135	196	197	198	199
2	8	9	10	11	72	73	74	75	136	...	...	...	200	...	...	...
3	12	13	14	15	76	77	78	79	...	...	...	...	...	...	...	...
4	16	17	18	19	80	81	82	83	...	...	...	...	...	...	...	...
5	20	21	22	23	84	85	86	87	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	58	59	...	...	122	123	...	...	...	187	...	...	...	251
15	60	61	62	63	124	125	126	127	188	189	190	191	252	253	254	255
	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

	Macro Block 0		Macro Block 1		Macro Block 2		Macro Block 3	
0	0	1	16	17	32	33	48	49
1	2	3	18	19	34	35	50	51
2	4	5	20	21	36	37	52	53
3	6	7	22	23	38	39	54	55
4	8	9	24	25	40	41	56	57
5	...	...	26	...	42	...	58	...
...	...	...	...	...	...	...	...	...
7	14	15	30	31	46	47	62	63
	0	1	0	1	0	1	0	1

Figure 3.20: H.264 Encoder algorithm scheme to determine a equation for addressing Y/UV buffers. Example for 64x16 frame resolution.

In Figure 3.20 is presented an algorithm developed for delivering the data synchronously and ordered from buffers, as demanded by the H.264 Encoder. More specifically, it represents an example, applied for a 64x16 frame size, of how the decision of what pixel should be delivered next to the H.264 Encoder is calculated. Nevertheless, the algorithm works for every frame size and the addressing varies with it at each clock. Once both Y and UV color components vary in terms of data quantity in the YUV 4:2:0, they also vary in the equation needed to address each color component. Therefore, two examples are shown in Figure 3.20 for the Y color (left) and UV color (right). Equations 3.1-3.3 are used to address the Y color component buffers, while Equations 3.5-3.7 address the UV component. The word lenght of each buffer is 32-bit, which allows storing 4 pixels in each word. It explains why each Macro Block is 4x16, instead of 16x16, as shown in Figure 3.20. Summarizing, each square has information for 4 pixels.

$$mb\_c\_width\_y = y\_counter\%4 \tag{3.1}$$

$$mb\_c\_height\_y = (y\_counter\%64)/4 \tag{3.2}$$

$$mb\_c\_n\_y = y\_counter/64 \tag{3.3}$$

Each equation presented aims to perform a specific assignment in the general equation (Equation 3.4 for Y and Equation 3.8 for both UV color components). These equations can be split in three main purposes: (i) calculate the width position of every MB, whose equation is presented in 3.1; (ii) Equation 3.2 calculates the height position of every MB; (iii) the calculus of the MB number is presented in Equation 3.3. In (i) it is calculated the remainder of the division by 4, which will give a number between 0-3 to address each MB width. The *y_counter* is in charge of counting the number of pixels delivered to Encoder. The calculus of the height position is addressed in (ii). It is divided by 4 to eliminate the width component of the calculated number, which will result in a number between 0-16. Lately, once each MB has 64 words, dividing the *y_counter* by 64, the resulted number represents the MB number, as stated in (iii). The variables discussed until now are independent of the frame size. The width frame size component is finally taken into account in the equations that calculate the address of the color components of a given pixel (Equation 3.4 for Y and Equation 3.8 for both UV color components).

$$\begin{gathered} buffer\_out\_addr\_y = mb\_c\_width\_y + \\ mb\_c\_height\_y \times y\_frame\_width + mb\_c\_n\_y \times 4 \end{gathered} \tag{3.4}$$

Equations 3.5, 3.6 and 3.7 follows the same logic as the 3.1, Equations 3.2 and 3.3, respectively. Equation 3.8 calculates the address of UV component from a given pixel.

$$mb\_c\_width\_uv = uv\_counter\%2 \tag{3.5}$$

$$mb\_c\_height\_uv = (uv\_counter\%16)/2 \tag{3.6}$$

$$mb\_c\_n\_uv = uv\_counter/16 \tag{3.7}$$

$$\begin{aligned} buffer\_out\_addr\_uv = mb\_c\_width\_uv\%2+ \\ mb\_c\_height\_uv \times uv\_frame\_width + mb\_c\_n\_uv \times 2 \end{aligned} \tag{3.8}$$

An example for addressing the 70th word of Y buffers to be delivered to encoder is presented in 3.9 equations. Therefore, number 70 in the left bottom table of Figure 3.20 need to correspond to number 22 in the left top table:

$$\begin{aligned} mb\_c\_width\_y = 70\%4 = 2 \\ mb\_c\_height\_y = (70\%64)/4 = 1 \\ mb\_c\_n\_y = 70/64 = 1 \end{aligned} \tag{3.9}$$

Then,

$$buffer\_out\_addr\_y = 2 + 1 \times 16 + 1 \times 4 = 22 \tag{3.10}$$

In this case, the *y_frame_width* is equal to 16 because 64/4 = 16. The number of input data pixels to be delivered each time to the Encoder is 4 since each addressable word has four data pixels. The final equation for both Y and UV color components is given in *buffer_out_addr_y* and *buffer_out_addr_uv*, respectively. It is used for real-time addressing the correspondent data from each buffer.

3.5.3 Data Transmission Subsystem

The demand for this subsystem emerges from the necessity of delivering the data in AXI4-Stream. It will be added a video signal to the AXI4-Stream interface for signalizing an end of a frame transaction. Moreover, this system is in charge of producing and delivering the header for the H.264 Encoder. This set of operations is controlled by the state machine depicted in Figure 3.21.

Part of the header is static and dependent on H.264 Encoder specificities in the prediction types applied, among others. Therefore it will not be much explored in the designing phase. The header carries some problems in the hardware designing phase due to a variable header depending on the frame resolution, as explained in section 2.1.3. To solve this problem it was set a fixed maximum size to be delivered in the header in which the useful data is variable. This solution adds some overhead in the header, however it solves the hardware designing problem. The detailed approach is presented in the subsection 4.2. Due to the fact that the encoder will be coding spatially every frame, the IDR parameter presented in the Figure 2.5 shall not be coded along with the bitstream of each frame.

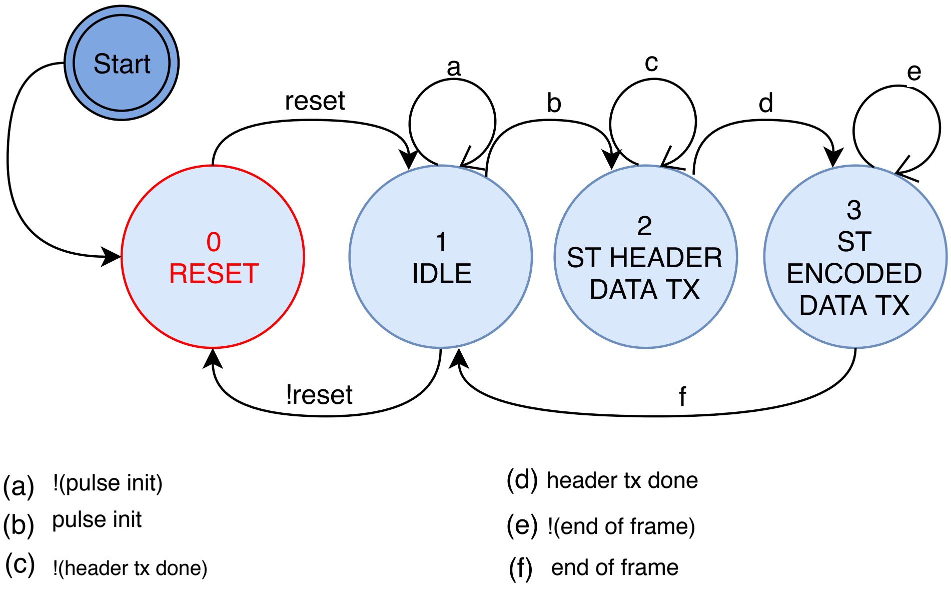


Figure 3.21: H.264 Encoder header delivering State Machine.

- *RESET* - This state serves for restarting the subsystem for the default initial values.
- *IDLE* - The state machine keeps waiting in this state after a frame is completely delivered to Memory Manager or after a reset. After a pulse is triggered for initiation of a new frame, it can run to the next state, *ST HEADER DATA TX*.
- *ST HEADER DATA TX* - In this state this subsystem is coding the encoder header and delivering it by AXI4-Stream.
- *ST ENCODED DATA TX* - After the header is finally delivered, the state machine remains in this state until the frame is completely encoded.

4. System Development

This chapter describes the implementation and the engineering effort on the development of the Memory Manager and Encoder H.264 cores specified in the previous chapter. To better expose some decisions taken in this phase, it will be listed some code snippets and the reasoning behind it will be explained as well.

4.1 Memory Manager

For the Memory Manager to work properly, it was implemented a hierarchy in which every sub-module, already presented in the design phase, communicates directly with the top-level module, where is implemented the main state machine to control the whole system. In short, each sub-module works as a slave led by the master top-level module.

Despite this core was developed aimed to support run-time configurable frame size up to 8k, the hardware was also designed to be configurable on the hardware level, i.e., the user can choose the same variables before the synthesis generation for the hardware be designed fitting the required specifications. For instance, the data bus width for each slave or master interface, address width and frames delay. More specifically, both AXI4 Stream Video and AXI4 Memory Mapped were designed to provide a configurable data bus width. The AXI4 Stream Video data width can be 64, 128, 256 or 512-bit, whereas the AXI4 Memory Mapped allows 32, 64, 128 or 256-bit. The burst transaction was set to 256 to improve the system throughput, by minmizing the disruptive latencies between the AXI4-Stream and AXI4-Full protocols. Additionally, it was implemented a configurable delay between the reading of frames in memory and the writing, up to five frames delay. It may be a responsibility of the user to assure that the parameters are set to fill the required system characteristics. The configurable parameters referred are applied in the elaboration of the hardware phase of the FPGA.

The buffers required for this core to work could be implemented either in FPGAs Block RAMs (BRAMs) or in Look Up Tables (LUTs), by simply adding vectors of registers in the HDL code once this is the default provided by IDEs. In case of a big amount of data to be required, it is highly recommended using BRAMs since the LUTs are a reource much more limited resource in any FPGA. Moreover, implementing big data storage in LUTs requires big hardware areas, which can result in severe time delays, leading the system to not meet the time constraints.

Since this core will be getting a large use of FPGA embedded memory, it was decided to be implemented in BRAMs by instantiating IP cores provided by *Xilinx*. This solution raises the complexity of the system because the data can only be accessed by address and has one clock delay. An address mechanism was implemented for anticipating the data required due to the delay imposed by this resource.

4.1.1 Configuration Subsystem

The next listings presented belong to the Configuration subsystem. It will be displayed some important code parcels to complement the information given in the design phase as well as some implementation decisions.

Memory Mapped registers updating

```verilog
input wire [4 : 0] i_REG_STATUS,
{...}
reg [C_S_AXI_DATA_WIDTH-1:0] status_reg;
{...}
always @(posedge S_AXI_ACLK)
begin
  if(S_AXI_ARESETN == 1'b0)
    {...}
  else
    begin
      status_reg <= {59'd0, i_REG_STATUS};
    end
end
```

Listing 4.1: Error register update.

Listing 4.1 shows the update of the status register of the system, in which at each clock the Configuration Subsystem is receiving the value updated from the top-level module. Currently, the system is providing 5-bit for status register and it may be easily upgraded in the future. The missing bits are filled up with zeros. This example pretends to show how the register interchanging between the Configuration Subsystem and the top-level module is done. The *status_reg* can be accessed at any time for core status checking as well as every register present in the core, as specified in 3.8.

Hardware bit cleaning for Core restart

```
// Signal to start/restart the memory manager
output wire o_PULSE_RESTART_MM,
reg [C_S_AXI_DATA_WIDTH-1:0] control_reg;
{...}
always @(posedge S_AXI_ACLK)
begin
  if(S_AXI_ARESETN == 1'b0)
    {...}
  else if(!slv_reg_wren && control_reg[0])
    begin
      control_reg <= {control_reg[C_S_AXI_DATA_WIDTH-1:1], 1'b0};
      {...}
end
// Generates a pulse after updating all the registers
assign o_PULSE_RESTART_MM = (!slv_reg_wren && control_reg[0])? 1 : 0;
```

Listing 4.2: Pulse generation and bit clear by hardware.

As referred previously in the design phase, the first bit of the Raw video control register (0h offset) was implemented to allow a master for restarting the Memory Manager with no need for a complete hardware restart. To make the task for the user/control unit easier and to decrease the necessity for configuring again this register to stop the restarting effect, it was implemented a bit register that is cleaned automatically by hardware.

Listing 4.2, although quite summarized, pretends to show how the bit cleaning is done for restarting the core. The *control_reg[0]* is set low as soon as possible after it is set high since we are tracking the *slv_reg_wren* signal. This signal enables the writing process of registers in the Configuration Subsystem, which means that when this bit is set low it is safe to clean the bit. This approach prevents the risk of a short circuit since there are not two elements trying to write in the same register at the same time.

This procedure is completed in the assignment of *o_PULSE_RESTART_MM* to generate only a single pulse for restarting the core. The missing bits remain untouched.

4.1.2 Write Subsystem

A few listings from the Write Subsystem will be shown next to complement the explanation of the system.

Data flow control counter tracking

```
always @(posedge S_AXI_ACLK)
begin
  if(m_axi_s2mm_aresetn == 1'b0 || i_PULSE_RESTART_MM == 1)
    {...}
  else
    begin
      // If AXI-Stream filled up a buffer
      if(pulse_axi_rxn_done && !pulse_axi_txn_done)
        begin
          check_counter <= check_counter + 1;
        end
      // If AXI-Full have dispatched the data stored in buffer
      else if(pulse_axi_txn_done && !pulse_axi_rxn_done)
        begin
          check_counter <= check_counter - 1;
        end
      {...}
    end
end
```

Listing 4.3: Counter updating for tracking input and output data from buffers.

To monitor the congestion during the write of a frame in memory, it was developed a counter to track the congestion in the double-buffering mechanism used by the Memory Manager. For this purpose, it counts how many times the Raw video write subsystem filled up a buffer. Once the number of buffers is limited, when this value rises above a given threshold, an error is issued because data is not being dispatched on time.

When the *pulse_axi_rxn_done* signal shown in Listing 4.3 emits a pulse, it means the slave finished the reception of new data from video source, whereas the *pulse_axi_txn_done* signals the end of data dispatching from buffers. It allows controlling both data reception and dispatching in buffers.

In the Verilog code presented in 4.3, there are three possible situations: (i) new data received and filled up the current buffer; (ii) data previously stored in buffers was dispatched from buffers by a Master AXI4 Memory Mapped; or (iii) new data arrived and filled up a buffer at the same time Master AXI4 Memory Mapped has ended up the transaction, i.e., both (i) and (ii) happened at the same time. When there is already new data in buffers to be dispatched, the slave in charge of receiving the video from external source notifies the top-level module by generating a pulse, and the *check_counter* is incremented (i). If the data is dispatched to external memory the counter is decremented (ii). In case both have completed each task at the same time, the register remains untouched, with the same number (iii). This counter is not supposed to reach the number 3 because the Memory Manager is provided by a double buffering mechanism, and the number three would mean that one buffer was filled up two times before data has been dispatched to external RAM, and consequently data has been lost. The status register (Table 3.10) should be updated if this error occurs.

Additionally, once the receiving data cannot stop, the system may receive the *pulse_axi_rxn_done* signal uninterruptedly, whereas the *pulse_axi_txn_done* signal depends on memory availability. This means that *pulse_axi_txn_done* may only decrement the counter after a previous increment because the main state machine, which is leading the starting of data transfer process to external memory, is allowing the start of the transfer only when there is a number different from zero in the *check_counter* register.

Frame address starting control

```
always @(posedge S_AXI_ACLK)
begin
if(m_axi_s2mm_aresetn == 1'b0 || i_PULSE_RESTART_MM == 1)
      {...}
      else if(pulse_init_axi_txn && flag_SOF)
        begin
          pulse_new_addr_valid <= 1;
          if(next_frame_addr + single_frame_size_bytes > i_MEM_ADDR_END)
            begin
              next_frame_addr <= i_MEM_ADDR_START;
            end
          else
            begin
              next_frame_addr <= next_frame_addr;
            end
        end
      else if(pulse_new_addr_valid)
        begin
          pulse_new_addr_valid <= 0;
          next_frame_addr <= next_frame_addr + single_frame_size_bytes;
      {...}
    end
end
```

Listing 4.4: Next address updating in the top Module.

There is a mechanism to, not only control the next frame address, but also update the status register in case the value being calculated in the top-level module does not match with the one calculated in the interface level of Master AXI4-Memory Mapped. In Listing 4.4 is shown a brief piece of code developed to lead and control the address increment presented in the top-level module.

After a new transaction has been initiated (*pulse_init_axi_txn*), in case the system has notified that a start of frame has already occurred (*flag_SOF*), the hardware will perform actions to guarantee that every frame is saved in the correct address, to prevent a possible misalignment of frames being saved in memory. This approach allows the system to know always where is the start of every frame saved in memory, simply by calculating an offset based on the size of each frame. The *flag_SOF* is set every time that the slave in charge of receiving the video receives a start of frame signal by AXI4-Stream Video. The demand for this flag emerges due to the delay in the system since the moment we receive the data and save it in buffers until the moment we finally read the data again to be delivered by the Master AXI4 Memory Mapped. Then, the flag is set high for informing the system that the very first word of the next transaction is the beginning of a new frame. A memory rotation is immediately performed if the remaining pre-configured memory space has not enough space to save a complete frame.

The *pulse_new_addr_valid* is designed to inform the Master in charge of delivering the video whenever a new address is set. Then, the block in charge of increment the address after each end of burst transfer is updated.

Transaction address increment

```verilog
input wire [C_M_AXI_ADDR_WIDTH - 1 : 0] i_NEW_ADDR,
{...}
localparam integer C_ADDR_INCREMENT = C_M_AXI_BURST_LEN * (C_M_AXI_DATA_WIDTH/ 8);
{...}
always @(posedge M_AXI_ACLK)
begin
  {...}
  // New addr coming from top module to assure each frame starts in the right addr
  else if(i_PULSE_NEW_ADDR_VALID)
    begin
      axi_awaddr <= i_NEW_ADDR;
    end
  else if(M_AXI_AWREADY && axi_awvalid)
    begin
      axi_awaddr <= axi_awaddr + C_ADDR_INCREMENT;
    end
  {...}
end
```

Listing 4.5: Next burst transaction updating Address register.

Listing 4.5 pretends to show how the address is updated before each transaction has been started. The top-level module is always notifying when a new start of frame has occurred. In this case, the *i_PULSE_NEW_ADDR_VALID* is signalizing it. The current address to be used in the next burst transaction is always changed as prevention. We could choose a different approach by comparing if the values do not match before applying the update, but it would cause the same effect and would also add unnecessary hardware in FPGA. However, it is supposed that the new value coming in the *i_NEW_ADDR* match with the *axi_awaddr* value, which is updated in each data burst. Additionally, there is another block in charge of comparing both values. In the case of non-matching of the values compared, an error is set in the status register (Table 3.10). After the burst transaction being accepted (*M_AXI_AWREADY && axi_awvalid*) by the Slave, it is immediately calculated the next address to be used.

4.1.3 Read Subsystem

The Read subsystem implementation is explained in this subsection. The explanation will be accompanied by some listings that highlight some key-functionalities.

Reading start control

```
always @(posedge m_axi_mm2s_aclk)
begin
  if(m_axi_mm2s_aresetn == 1'b0 || pulse_restart_mm == 1)
    {...}
  else
    begin
      case(mst_read_subsys_exec_state) IDLE_READ_MEMORY:
        begin
          if((check_frames_counter >= FRAMES_DELAY) && read_subsys_active)
            begin
              pulse_init_axi_rxn <= 1;
              mst_read_subsys_exec_state <= READ_MEMORY;
          {...}
    end
```

Listing 4.6: Reading start control in the Main Read State Machine.

The read of a frame from the memory is controlled by the state machine described in detail in Section 3.4.3. Part of the HDL that implements this state machine is shown in Listing 4.6. It pretends to control when the Read subsystem is able or not to initiate a reading of frame once it may be started only when there is already at least one frame completely saved in memory. The *FRAMES_DELAY* presented in the code is a fixed parameter compiled in the hardware implementation phase to impose a frame delay between the reading and writing of video content. It is the responsibility of the user to assure that the memory space is large enough to support the *FRAMES_DELAY* set in the IP Core configuration. In case of this value being wrongly configured it would result in frames lost at the beginning of the video streaming due to frames overwrite before the Read Subsystem could finally start reading video.

Both top-level module and Read subsystem are in charge of a continuous update of the *check_frames_counter* register. The top-level module emits a pulse whenever a new frame has been completely saved in memory, whereas the Read Subsystem is informing when a frame was completely read from external memory RAM and delivered by AXI4-Stream Video.

Transaction address increment

```
always @(posedge m_axi_mm2s_aclk)
  begin
    if(m_axi_mm2s_aresetn == 1'b0 || pulse_restart_mm == 1)
     {...}
    else if(pulse_axis_tx_frame_done && !pulse_new_addr_valid && ((m_axi_mm2s_araddr
    ↪   + single_frame_size_bytes) > mem_addr_end))
      begin
        pulse_new_addr_valid <= 1;
        new_read_addr <= i_MEM_ADDR_START;
        single_frame_size_bytes <= single_frame_size_bytes;
      end
    else
      {...}
end
```

Listing 4.7: Calculates the new frame address to keep frames in the expected addresses or to rotate memory.

The code presented in Listing 4.7 is part of the starting frame address control mechanism discussed in the design subsection 3.4.2. This piece of code is aimed to define when the rotation of memory is performed. The *pulse_new_addr_valid* informs the master module in charge of increment the address at each burst transaction that there is a new address for the next transaction to be applied. Then, the master replaces the current value by the new value transferred in *new_read_addr*.

Tracking the number of pixels transmitted

```
localparam N_PIXELS_PER_TRANSACTION = C_M_AXIS_TDATA_WIDTH/32;
{...}
always@(posedge M_AXIS_ACLK)
begin
  if(!M_AXIS_ARESETN || i_PULSE_RESTART_MM)
    {...}
  else if(tnext && (counter_width == (frame_width - N_PIXEIS_PER_TRANSACTION)))
    begin
      frame_width <= frame_width;
      counter_width <= 0;
    end
  else if(tnext && (counter_width != (frame_width - N_PIXEIS_PER_TRANSACTION)))
    begin
      frame_width <= frame_width;
      counter_width <= counter_width + N_PIXEIS_PER_TRANSACTION;
    end
  else
    {...}
end
```

Listing 4.8: Counter for frame width tracking, according to pre-configured parameters.

There is a demand for tracking the video content transferred by the Master AXI4-Stream Video, led by the Read subsystem, to perform correctly the protocol according to specification provided in [45]. The tracking allows knowing not only whenever each frame has finished transmitting, but also to provide the video signals *TUSER* and *TLAST*.

As detailed in Listing 4.8, once the video is transferred at each clock, it is necessary to know how many pixels are delivered in each word. This value vary due to a different IP Core user hardware configuration in *C_M_AXIS_TDATA_WIDTH* parameter, which can be 128, 256 or 512-bits. Therefore, the *N_PIXELS_PER_TRANSACTION* parameter calculated is used to increment the number of pixels transferred each time *tnext* is driven high. The *tnext* is set high whenever both Slave and Master are ready for data transferring. After transmitting the expected number of pixels, the value of this counter is reset and the *tlast* signal of the AXI4-Stream Video bus is driven high during a single clock.

Specific video signals assignment

```
input wire i_TVALID_FROM_TOP,
{...}
assign M_AXIS_TVALID = i_TVALID_FROM_TOP;
assign axis_tlast = (tnext && (counter_width == (frame_width -
    ↪ N_PIXEIS_PER_TRANSACTION)) && (mst_exec_state == FRAME_TRANSMISSION))? 1 : 0;
assign axis_tuser = (!counter_height && !counter_width && (mst_exec_state ==
    ↪ FRAME_TRANSMISSION) && !txn_done)? 1 : 0;
```

Listing 4.9: Combinational logic for AXI Stream Video Signals.

Both *axis_tlast* and *axis_tuser* signals presented in the Listing 4.9 are demanded to accomplish the AXI4-Stream Video protocol. The *axis_tlast* is controlled by a block in charge of tracking the progress of pixels transferred in each line. Whereas *axis_tuser* is driven high when both *axis_tlast* signal and a block in charge of height counter achieve the configured values. Additionally, the top-level module is in charge of control if the data transferred is valid (*i_TVALID_FROM_TOP*) once the system is streaming immediately from the Slave, i.e., the Slave is signalizing if the data is valid or not.

4.2 H.264 Encoder

This subsection introduces the implementation of the core parts of the H.264 Encoder. It could be also exposed the configuration subsystem for this core, but since there are only some residual differences from the Configuration Subsystem of Memory Manager, it was decided to focus most on other pieces of this hardware module.

Frame width and Odd line control for Raw video color space conversion

```verilog
always@(posedge S_AXIS_ACLK)
begin
  if(!S_AXIS_ARESETN || i_PULSE_RESTART_ENCODER)
      {...}
  else if((count_width >= (i_FRAME_WIDTH - N_PIXELS_RECEIVED)) && data_ready)
    begin
      count_width <= 0;
      odd_line <= odd_line + 1;
    end
  else if(data_ready && (count_width != (i_FRAME_WIDTH - N_PIXELS_RECEIVED)))
    begin
      count_width <= count_width + N_PIXELS_RECEIVED;
      odd_line <= odd_line;
    end
  else
  {...}
end
```

Listing 4.10: Updates the width counter and odd line for conversion from YUYV 4:2:2 to YUV 4:2:0.

Due to the demand for video conversion from YUYV 4:2:2 10-bit per color component to YUV 4:2:0 8-bit, the H.264 Encoder core is in charge to perform this conversion. This conversion, whose part of the code is presented in Listing 4.10, depend on the *odd_line*, which serves to control when this data is accepted or not to be buffered. The color component data is accepted in odd lines, while in even lines it is discarded. Additionally, the aim of the code developed is to work as a parsing block of each color component to be saved in the respective color component buffer. Both *odd_line* and *count_width* are used in another block to assist the color space conversion and buffering.

Reconfigurable hardware combinational logic for conversion from YUYV 4:2:2 10-bit to YUV 4:2:0 8-bit

```
genvar i;
generate
  for(i = 0; i < N_PIXELS_RECEIVED; i = i + 1)
  begin
    assign o_Y_BUFFER_INPUT_DATA[(i*8) +: 8] = S_AXIS_TDATA[2 + i*32 +: 8];
  end
  for(i = 0; i < N_PIXELS_RECEIVED/2; i = i + 1)
  begin
    assign o_UV_BUFFER_INPUT_DATA[(i*16) +: 8] = !odd_line ?  S_AXIS_TDATA[2 + (i*64 + 16) +: 8] : 0;          //  U assign
    assign o_UV_BUFFER_INPUT_DATA[((i*16) + 8) +: 8] = !odd_line ? S_AXIS_TDATA[2 + (i*64 + 48) +: 8] : 0;       //  V assign
  end
endgenerate
```

Listing 4.11: Reconfigurable input data and conversion from YUYV 4:2:2 10-bit to YUV 4:2:0 8-bit by combinational logic.

The code presented in Listing 4.11 aims for performing the parsing of video content from the data bus. This code was developed to be reconfigurable in the hardware assignment level during the Register Transfer Level (RTL) generation phase, depending on the user configuration parameters set in the instantiation of the core. This approach avoids the demand for the user to understand the coding context to change the code. The *S_AXIS_TDATA* width may vary on the supported values 128, 256 or 512-bits. Therefore, depending on the user parameters set, the code presented will generate an auto-assignment and adjust the parsing of each color component. Both Y and UV buffers are filled automatically and aligned at each clock and valid data. Moreover, the color component data are only buffered during odd lines.

Addressing MacroBlocks assignment

```
localparam N_PIXELS_PER_OUTPUT_WORD_Y_BUFFER = (Y_BUFFERS_OUTPUT_DATA_WIDTH/8);
{...}
// Y component
assign mb_width_y = reg_y_frame_width /N_PIXELS_PER_OUTPUT_WORD_Y_BUFFER;
assign mb_counter_width_y = ((counter_y_16x_width /4) % 4);
assign mb_counter_height_y = (((counter_y_16x_width /4) % 64) /4);
assign mb_counter_n_y = ((counter_y_16x_width /4) / 64);
assign y_buffer_output_addr = mb_counter_width_y + mb_counter_height_y * mb_width_y
    ↪ + mb_counter_n_y * 4;
{...}
```

Listing 4.12: Algorithm for addressing Y MBs.

As specified previously in the design phase, it was developed the algorithm presented in Figure 3.20 to address dynamically the data presented in buffers at each clock. The Listing 4.12 pretends to show a part of the code developed in charge of performing the calculus to address the buffers to deliver the data, according to the H.264 Encoder demand. In this case, it is presented the addressing for the Y color component. The local parameter *N_PIXELS_PER_OUTPUT_WORD_Y_BUFFER* serves to calculate the number of words each frame line requires to fill an image whose resolution was previously configured. The *counter_y_16x_width* is a register being updated each time the data flow has moved forward. Finally, the *y_buffer_output_addr* is the register interfacing with the buffer to address the data to be delivered to the encoder.

Frame width header coding in Exponential Golomb

```
input wire  [9 : 0] i_FRAME_WIDTH_DIV_BY_16,
input wire  [8 : 0] i_FRAME_HEIGHT_DIV_BY_16
{...}
assign exp_golomb_frame_width = i_FRAME_WIDTH_DIV_BY_16;
assign exp_golomb_frame_height = i_FRAME_HEIGHT_DIV_BY_16;
assign exp_golomb_frame_width_height = (exp_golomb_count_bits_frame_height_lenght ==
    ↪  17)? {exp_golomb_frame_width[18:0], exp_golomb_frame_height[16:0]}:
```

```
(exp_golomb_count_bits_frame_height_lenght == 15)? {align_header[1:0],
    ↪ exp_golomb_frame_width[18:0], exp_golomb_frame_height[14:0]}:
(exp_golomb_count_bits_frame_height_lenght == 13)? {align_header[3:0],
    ↪ exp_golomb_frame_width[18:0], exp_golomb_frame_height[12:0]}:
(exp_golomb_count_bits_frame_height_lenght == 11)? {align_header[5:0],
    ↪ exp_golomb_frame_width[18:0], exp_golomb_frame_height[10:0]}:
(exp_golomb_count_bits_frame_height_lenght == 9)? {align_header[7:0],
    ↪ exp_golomb_frame_width[18:0], exp_golomb_frame_height[8:0]}
{...}
: 0; // ERROR
```

Listing 4.13: Algorithm determination of Exponential Golomb for frame width header coding in Encoder H.264 header.

The code presented in the Listing 4.13 pretends to show how the frame resolution is coded in the header. This is a particular and tricky part since the header may vary depending on the frame width and height. Usually, hardware coding is based on predictable and static data length, contrary to the H.264 Encoder header, whose frame resolution is coded in the header in the Exponential-Golomb format. This is complicated to perform because, depending on the frame resolution, the frame width may occupy at least 11-bit (in case the frame width is 640 px) and up to 19-bit (in case the frame width is 8192 px). Consequently, the frame height may start early or later in the header bit sequence and it may occupy a variable number of bits in the header since 9 (in case the frame height is 480 px) up to 17-bit (in case the frame height is 4320 px). The missing parameters for the header that comes after frame resolution coding may start early or later in the header due to the same reasons. Whereas in software it could be easily performed, in HDL it is a little more complicated due to data parallelism characterized by the hardware. Moreover, the header is supposed to be calculated faster than the first output data encoded by the H.264 Encoder, and since in this case the header may be coded in 16 different possibilities, the hardware needs to calculate some chained data fast and also be prepared for each different possibility. To accomplish that, it has been developed a mechanism to detect previously how many bits it will be required to code the frame resolution in Exponential-Golomb format. Two different block are in charge of calculate how many bits each width (*exp_golomb_count_bits_frame_width_lenght*) and height (*exp_golomb_count_bits_frame_height_lenght*) will be required to occupy in the header. This is important to calculate where will be placed the remaining parameters. Then, another block is summing both values to determine the offset to code the missing parameters also presented in the H.264 header.

The code shown in Listing 4.13 is assigning each bit in the correct position. Moreover, the remaining parameters need to be shifted as well.

Header Sequence Parameter Set bit assignment

```
assign sp_parameters = ((exp_golomb_count_bits_frame_width_lenght +
    ↪ exp_golomb_count_bits_frame_height_lenght) == 36)? {DEFAULT_SP_PARAMETERS_1
    ↪ [40:0], exp_golomb_frame_width_height[35:0], DEFAULT_SP_PARAMETERS_2[4:0],
    ↪ align_header[13:0]}:
((exp_golomb_count_bits_frame_width_lenght +
    ↪ exp_golomb_count_bits_frame_height_lenght) == 34)? {DEFAULT_SP_PARAMETERS_1
    ↪ [40:0], exp_golomb_frame_width_height[33:0], DEFAULT_SP_PARAMETERS_2[4:0],
    ↪ align_header[15:0]}:
((exp_golomb_count_bits_frame_width_lenght +
    ↪ exp_golomb_count_bits_frame_height_lenght) == 32)? {DEFAULT_SP_PARAMETERS_1
    ↪ [40:0], exp_golomb_frame_width_height[31:0], DEFAULT_SP_PARAMETERS_2[4:0],
    ↪ align_header[17:0]}:
{...}
((exp_golomb_count_bits_frame_width_lenght +
    ↪ exp_golomb_count_bits_frame_height_lenght) == 20)? {DEFAULT_SP_PARAMETERS_1
    ↪ [40:0], exp_golomb_frame_width_height[19:0], DEFAULT_SP_PARAMETERS_2[4:0],
    ↪ align_header[29:0]}
: 0; // ERROR
{...}
```

Listing 4.14: H.264 Encoder header SPS.

H.264 Encoder header also requires the coding of the SPS, which is a fixed parameter coded in the HDL once it is dependent on the data encoding laws and techniques performed inside the open-source H.264 Encoder [47]. In the Listing 4.14 is presented how the SPS may be assigned in the final header, by shifting left or right, depending on the Exponential-Golomb coding of frame resolution. Additionally, the header needs to be byte-aligned, which means that if the total header length is not an integer byte number, the remaining bits may be filled up with zeros. The decoder may know the added zeros need to be ignored during the decoding phase.

5. Evaluation and Results

The present chapter evaluates the implementation of both Memory Manager and H.264 Encoder IP cores. In each one was applied some tests to validate the system. In the end, it is given a brief conclusion about the results. Once the project in which this MSc thesis is inserted is not finished yet, the only way to test the read and write of both raw and encoded video is by transferring the content of the external RAM to an SD card, after running the system. In the case of raw video, the content of the SD card is compared byte by byte with the content of the expected frames, using *VBinDiff* [48]. In the case of encoded video, the content of the SD card was reproduced using the *FFMPEG* Application Programming Interface (API), being the output compared with the inherent raw video.

Two different environments were used to test and validate the Memory Manager: (i) Memory Manager IP core instantiated along with a Video Test Pattern Generator (VTPG) IP core, provided by *Xilinx*, to simulate the video source; (ii) Memory Manager IP core instantiated with 12G-SDI IP core in a Quad-Link topology, for video streaming to be validated in the real application. Initially, it was used the (i) approach to approve the system and test some particular situations to be applied in (ii). Still in the (i) testing environment, it was also instantiated a VDMA IP core, provided by *Xilinx*, to analyze the conformity of the frames streamed by the Memory Manager Read Subsystem and the AXI4-Stream Video protocol by checking the VDMA register status. After verified and validated, the next phase was performing the (ii) situation, in which the video is streamed in continuous mode by a 12G-SDI in a Quad-Link topology to deliver up to 8k at 60 fps. The clock frequency was set at 300 MHz. Once some parts of the global project are still not finished yet, it was used the PS for reading frames by the *FFMPEG* API [49]. The boards used for testing was the *Xilinx ZCU102* and *Zybo*. The demand for this second board emerges for testing the case in which the Memory Manager is losing data due to congestion in memory accesses. Once this board has lower memory throughput, impossible to deal with 8k, it is supposed that it will be given an error of data loss.

For testing and validation of H.264 Encoder IP core, it was used two different approaches: (i) Code and behavior simulation deployed for better understanding how data should be provided, and (ii) H.264

Encoder IP core instantiated along with the Memory Manager for providing the AXI4-Stream Video by reading video content through memory. After each one has completed the encoding process, it is read in a common computer by taking advantage on the *FFMPEG* open-source API [49].

5.1 Memory Manager Core Functionalities

Two different environments were used to test and validate the Memory Manager: (i) Memory Manager IP core instantiated along with a VTPG IP core, provided by *Xilinx*, to simulate the video source; (ii) Memory Manager IP core instantiated with 12G-SDI IP core in a Quad-Link topology, for video streaming to be validated in a real application. Initially, it was used the approach (i) to approve the system and test some particular situations to be applied in (ii). In the testing environment (i), it was also instantiated the *Xilinx* VDMA core, to attest to the integrity of the frames written in memory. In the testing environment (ii) the video source was set to deliver video with up to 8k resolution at 60 fps. The clock frequency was set at 300 MHz. Once some parts of the video card are still not finished yet, it was used the PS for reading frames by the *FFMPEG* API [49]. The boards used for testing was the *Xilinx ZCU102* and *Zybo*. The demand for this second board emerges for testing the case in which the Memory Manager is losing data due to congestion in memory accesses. Once this board has lower memory throughput, which can not handle 8k video, the Memory Manager is supposed to raise the data loss errors.

5.1.1 Double Buffering mechanism

In Figure 5.1 is presented the double buffering mechanism, in which each buffer is only being read or written at each time. In spite of both address channels being connected to the same register, it is not being affected in both buffers. The read enable and write enable signals are in charge of enabling the write/read action in each buffer. The most important is that the system is not losing any data due to buffer changing. As the waveform shows, data is being dispatched faster than received. Indeed, when buffer 1 is filled up (Write Enable (*WE*) signal goes from 1 to 0), buffer 2 has already the Read Enable (*RE*) signal driven low.

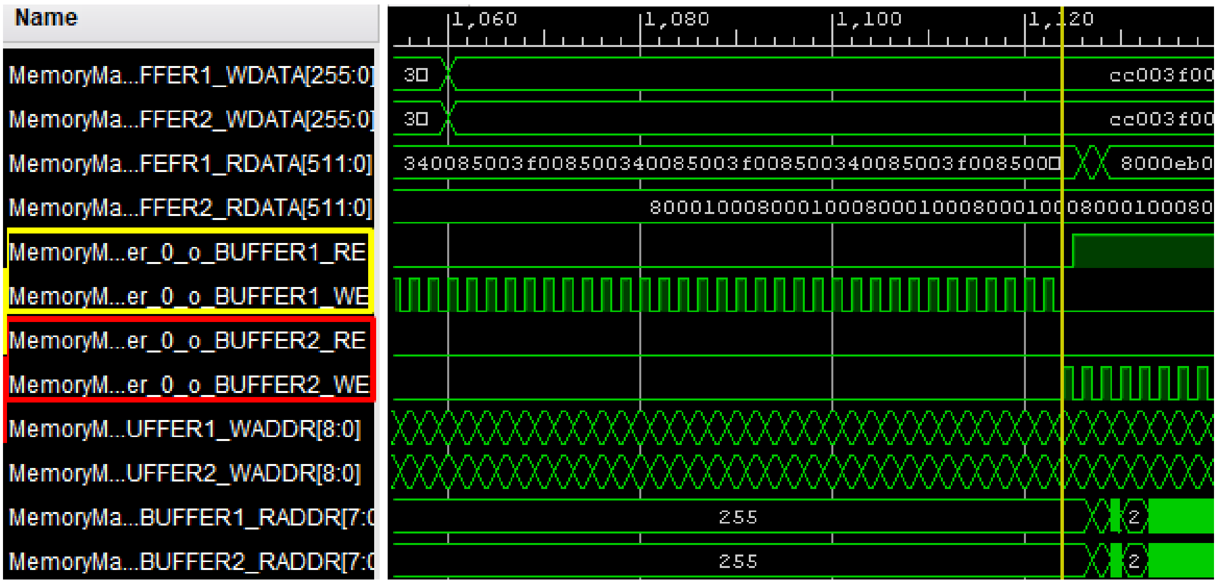

Name
1,060
1,080
1,100
1,120
MemoryMa...FFER1_WDATA[255:0]
MemoryMa...FFER2_WDATA[255:0]
MemoryMa...FEFR1_RDATA[511:0]
MemoryMa...FFER2_RDATA[511:0]
MemoryM...er_0_o_BUFFER1_RE
MemoryM...er_0_o_BUFFER1_WE
MemoryM...er_0_o_BUFFER2_RE
MemoryM...er_0_o_BUFFER2_WE
MemoryM...UFFER1_WADDR[8:0]
MemoryM...UFFER2_WADDR[8:0]
MemoryMa...BUFFER1_RADDR[7:0
MemoryMa...BUFFER2_RADDR[7:0
30
cc003f00
30
cc003f00
340085003f008500340085003f008500340085003f008500
8000eb0
800010008000100080001000800010008000100080
255
2
255
2

Figure 5.1: Double-buffering mechanism test - ILA.

5.1.2 Memory Rotation

Memory rotation shown in Figure 5.2 is the result of a test applied by software programming, in which it is configured the start address and memory size for Memory Manager IP core. A brief part of the software program is shown in Listing 5.1. As can be observed in the waveform, after reaching the 0x10384000 address, the system is ready for delivering a new frame. Then, the system perceived the remaining memory region, previously configured for video, is not enough to save a complete frame. The Memory Manager applied immediately an address memory rotation. Finally, the address is accepted by the slave and the new frame is written at the beginning of the memory region. There is no fragmentation of frames saved in memory.

```
u64 startAddr_MM = 0x10000000;
u64 mem_size_MM =  0x10390000;
mm_encoder_configFrame(HOR_SIZE_BYTES, VER_SIZE);
mm_encoder_setMemorySize(startAddr_MM_Encoder, mem_size_MM_Encoder);
{...}
mm_Start(RESTART_MM, WRITE_SUBSYS_ACTIVE, READ_SUBSYS_ACTIVE);
```

Listing 5.1: Example Software Core configuration for Memory writting Rotation.

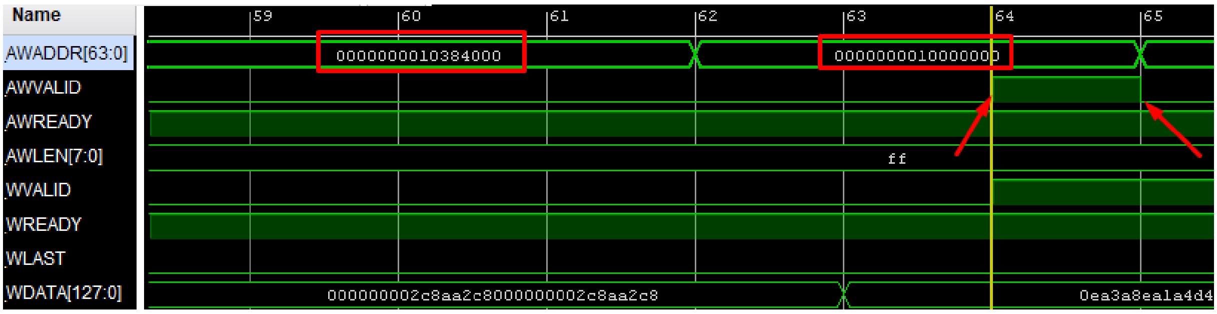


Figure 5.2: Memory rotation test - ILA.

5.1.3 Frame Error Detection Mechanisms

An early end of line error occurs whenever a video source has informed the Memory Manager an end of a line transmission earlier than expected by pre-configured parameters. For testing this functionality, the Memory Manager and the video source were configured with different resolutions. The Memory Manager was configured for accepting 8192x4320 video size. The video source was configured to deliver 8092 frame width. The height was kept in 4320. This configuration originated, as expected, the error shown in Figure 5.3.

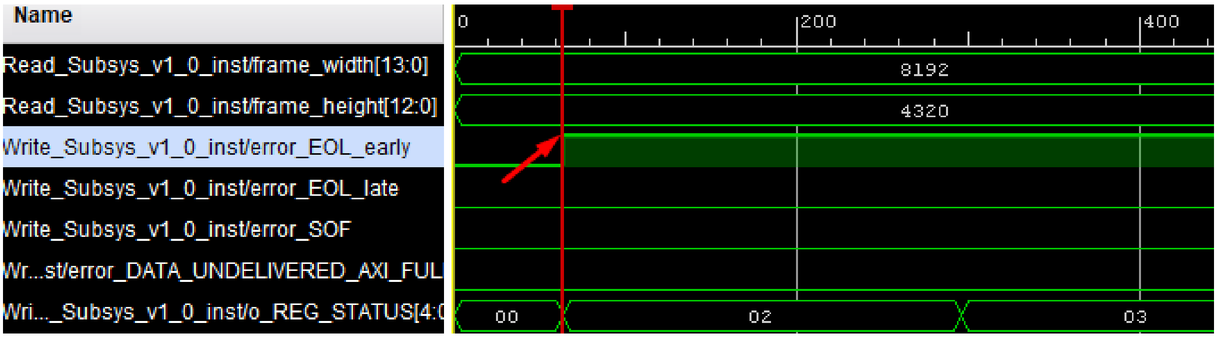

Name
Read_Subsys_v1_0_inst/frame_width[13:0]
Read_Subsys_v1_0_inst/frame_height[12:0]
Write_Subsys_v1_0_inst/error_EOL_early
Write_Subsys_v1_0_inst/error_EOL_late
Write_Subsys_v1_0_inst/error_SOF
Wr...st/error_DATA_UNDELIVERED_AXI_FUL
Wri..._Subsys_v1_0_inst/o_REG_STATUS[4:0
0
200
400
8192
4320
00
02
03

Figure 5.3: Early end of line error test - ILA.

Contrary to the early end of line error, the end of line late error occurs if the video source delivers a larger width than expected. For testing this situation, the Memory Manager was configured with an 8192x4320 frame size, and the video source was delivering an 8292 frame width. The error occurred as expected, as shown in Figure 5.4.

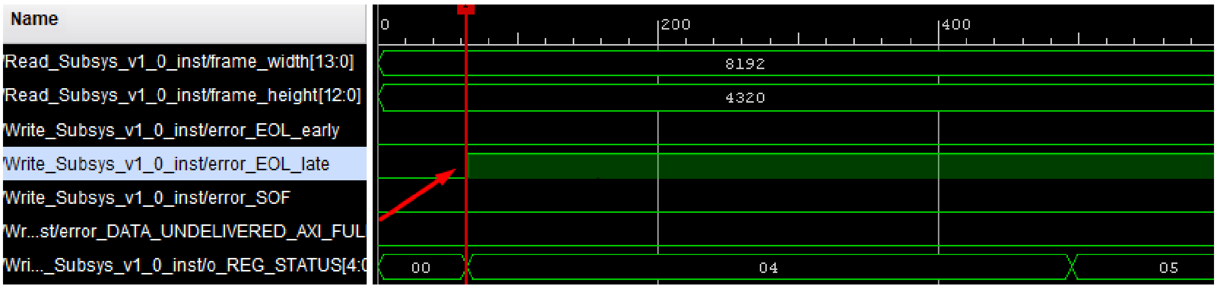

Name
Read_Subsys_v1_0_inst/frame_width[13:0]
Read_Subsys_v1_0_inst/frame_height[12:0]
Write_Subsys_v1_0_inst/error_EOL_early
Write_Subsys_v1_0_inst/error_EOL_late
Write_Subsys_v1_0_inst/error_SOF
Wr...st/error_DATA_UNDELIVERED_AXI_FUL
Wri..._Subsys_v1_0_inst/o_REG_STATUS[4:0
0
200
400
8192
4320
00
04
05

Figure 5.4: Late end of line error test - ILA.

For testing a start of frame error, the video source was configured to deliver 8192x4420 frame size. Whereas the Memory Manager is configured to receive each frame with 8192x4320 pixels. This configuration generated the predicted error, as shown in Figure 5.5. If the video source delivers a smaller frame height (4120, for instance) than the expected from Memory Manager, the error is detected as well.

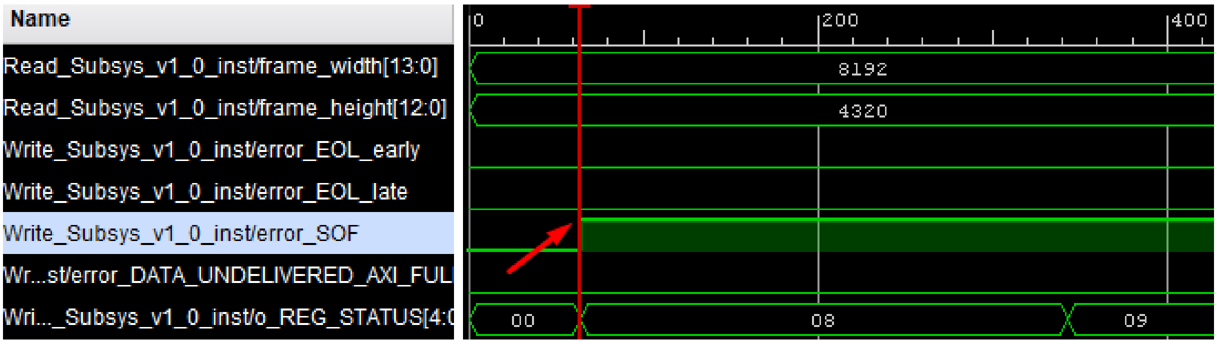

Name
Read_Subsys_v1_0_inst/frame_width[13:0]
Read_Subsys_v1_0_inst/frame_height[12:0]
Write_Subsys_v1_0_inst/error_EOL_early
Write_Subsys_v1_0_inst/error_EOL_late
Write_Subsys_v1_0_inst/error_SOF
Wr...st/error_DATA_UNDELIVERED_AXI_FUL
Wri..._Subsys_v1_0_inst/o_REG_STATUS[4:0
0
200
400
8192
4320
00
08
09

Figure 5.5: Start of frame error test - ILA.

5.1.4 Frame start address rectification

Each error on video delivering to Memory Manager may produce a misalignment of frames saved in memory. The system is prepared to not only detect but also to take actions in case an error occurs to guarantee that every frame is aligned in memory and saved as supposed. In Figure 5.6 is shown the Memory Manager performing an addressing frame rectification before the address is accepted, and data starts being written in memory.

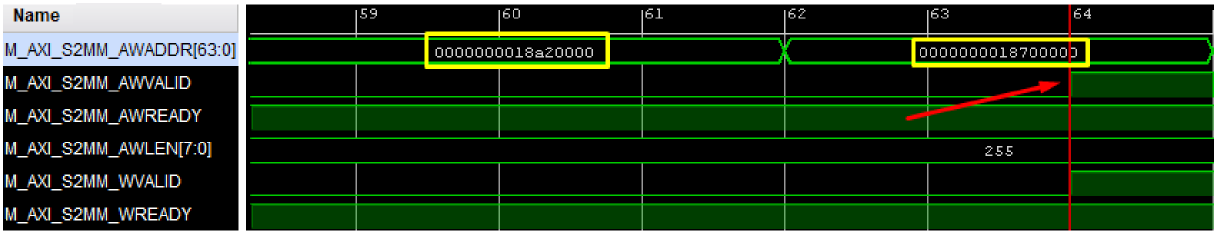


Figure 5.6: Frame start address rectification test - ILA.

5.1.5 Data loss detection

In Figure 5.7 is presented a test performed with the *ZYBO* board to test the undelivered data error, whose aim is tracking if there is data lost in the process of writing to external memory. As can be observed, this mechanism detected the rate the system was delivering the data was not enough to deal with the data coming.

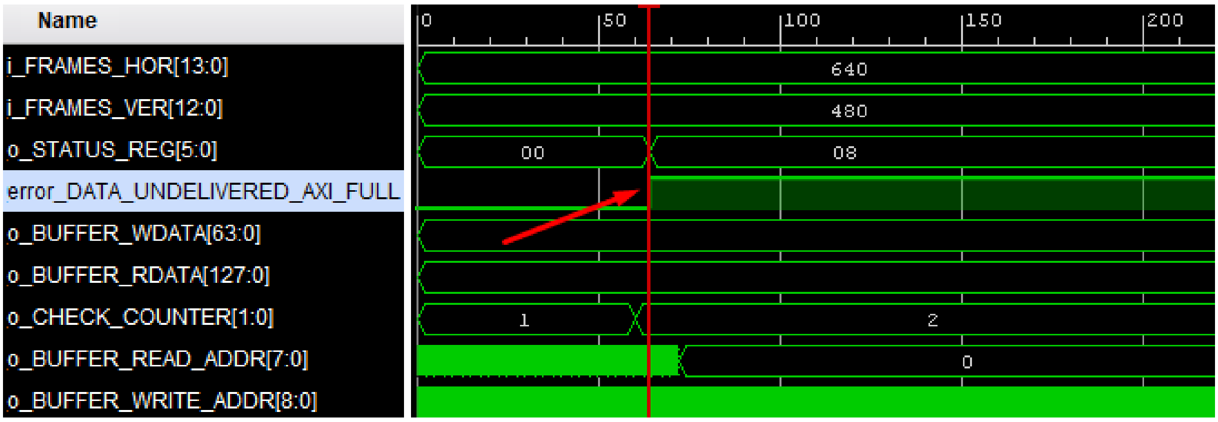


Figure 5.7: Data loss detection test -ILA.

5.2 H.264 Encoder Core Functionalities

For testing and validating the H.264 Encoder, it was used two different approaches: (i) code and behavior simulation, to understand how data must be delivered to the encoding mechanism, and (ii) H.264 Encoder IP core instantiated in a block design for FPGA, along with the Memory Manager for providing the AXI4-Stream Video by reading video content through memory. The output of each testing scenario is verified through the open-source *FFMPEG* API [49].

5.2.1 Working time in encoding

The ILA waveform presented in Figure 5.8 pretends to show a part of the encoding process being performed. The system was configured for 8k video resolution. The first three waveforms are the synchronized state machines discussed in the design and implementation phases (3 and 4). During the system test, it was confirmed that the video was being delivered faster than the encoder could process it. As proof, in Figure 5.8 are presented four colors of signals responsible for hand-shaking between the buffers and the encoder. The waves marked in red and dark blue are signals provided by the H.264

Encoder to signalize it is ready to process new data. Whereas the orange and cyan blue are the signals for indicating the availability for delivering new data to be encoded. Admittedly, if this encoder processed 8k video at a real-time rate, we would be delivering the data without delay. This ensures that if the H.264 Encoder is updated in the future, this interface can undoubtedly provide real-time data. At the bottom, the wave colored in purple presents the first bytes of encoded video provided by the H.264 Encoder, whereas the white signal is validating the data encoded.

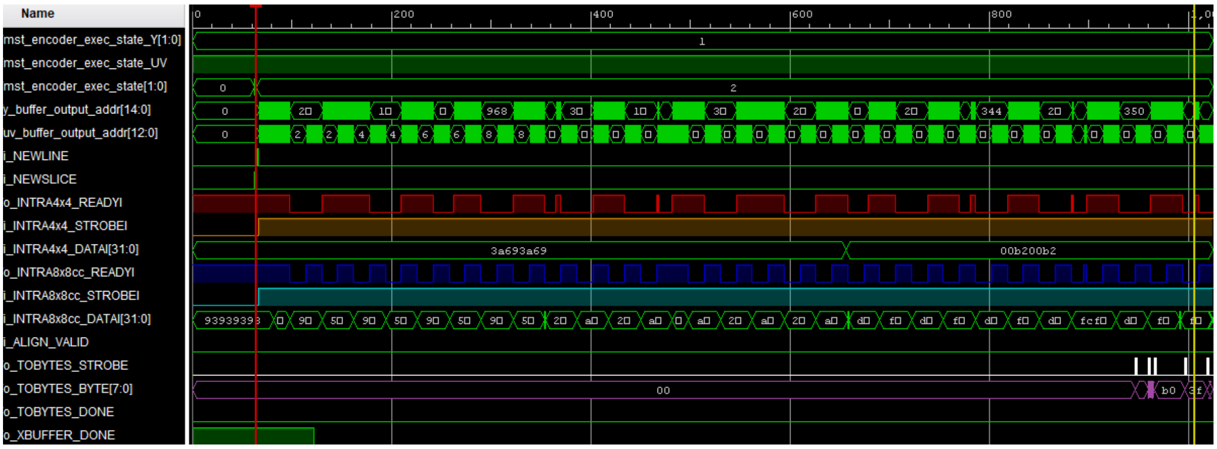


Figure 5.8: Working time for the H.264 Encoder - ILA.

5.2.2 Buffers Addressing Algorithm

In Figure 5.9 is presented an extension of Figure 5.8. It pretends to show the addressing algorithm explained in the equations defined in Section 3.5.2. They are providing the addressing of the content presented in the buffers. If we replicate the calculus for 8k, we achieve the values presented in the Figure 5.9.

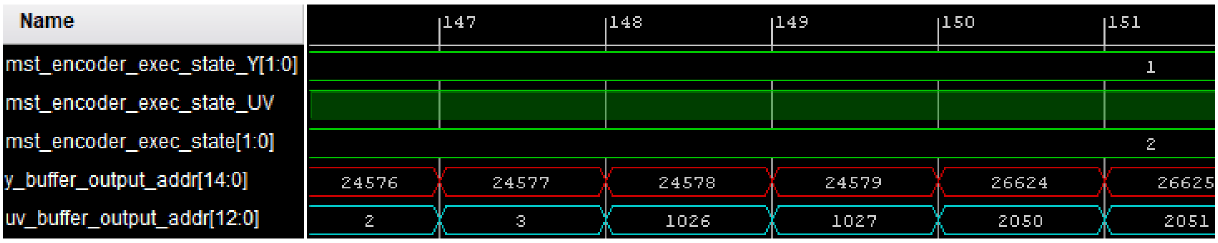

Name
mst_encoder_exec_state_Y[1:0]
mst_encoder_exec_state_UV
mst_encoder_exec_state[1:0]
y_buffer_output_addr[14:0]
uv_buffer_output_addr[12:0]
147
148
149
150
151
1
2
24576
24577
24578
24579
26624
26625
2
3
1026
1027
2050
2051

Figure 5.9: Buffers addressing algorithm - ILA.

5.3 Hardware Costs

For extracting the hardware costs for both IP cores developed, it was analyzed the *Vivado* post-implementation utilization report. The results are shown in Table 5.1. It is possible to identify that it was only used just a small percentage of the resources available, as expected, once the board taken for the project deployment is one of the largest available in the market. Despite this, it was used a considerable quantity of BRAMs due to the demand for a high buffering in both modules developed to meet the 8k goal and for optimizing memory accesses. There was no *LUT as memory* used by the Memory Manager once LUTs would be consumed fast if we use them as memory. Moreover, the using of *LUT as memory* may originate timing errors during the placing of the implementation process in *Vivado* tool-chain. Even though the H.264 Encoder uses *LUT as memory*, the percentage of use is very low, not justifying the engineering effort necessary to update the HDL of the open-source encoder [47] used as the basis for this system.

Table 5.1: Resource utilization.

Resource Name		Memory Manager	H.264 Encoder	Available	Used (%)
Slice Registers		1479	3315	599 550	4794 (0.8%)
LUTs	**as Logic**	1505	6976	274 080	6242(2.3%)
	as Memory	0	2239		2239(0.8%)
Slices		401	1398	28800	1799 (6.2%)
DSP Slices		9	4	2 520	13 (0.5%)
Block RAM		8	89	912	97 (10%)

In Figure 5.10 is provided a graphic that compares both Memory Manager and VDMA solutions in terms of required resources from the FPGA IP cores. As we can see in results, we achieved the same main features as the VDMA with a general lower using of resources. Moreover, we added new precious features.

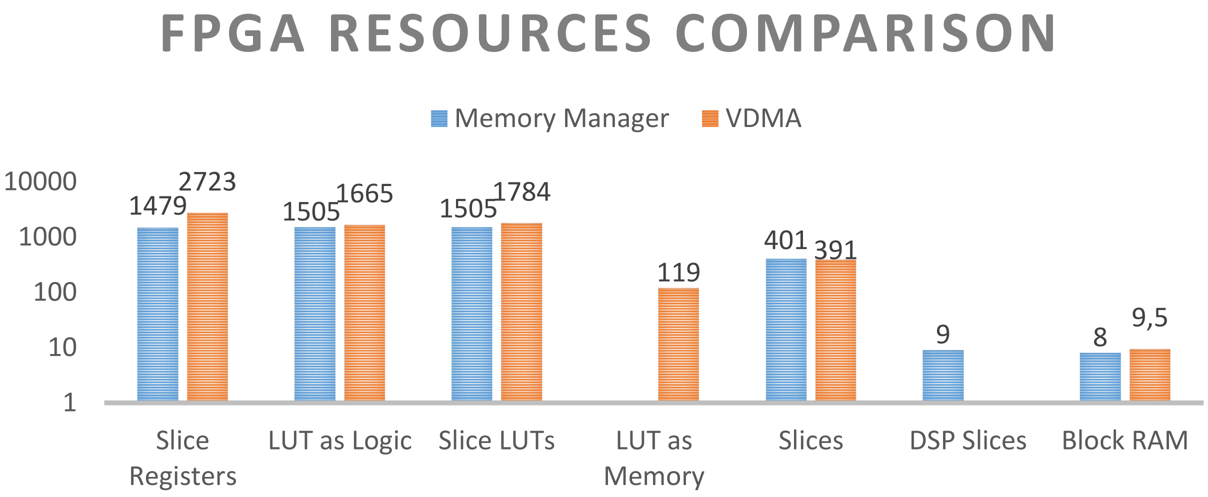


Figure 5.10: FPGA Resources - Memory Manager vs VDMA.

5.4 Conclusion

As it is possible to verify during this chapter, the Memory Manager achieved all goals successfully and it is provided with useful mechanisms for monitoring video transferring to memory. In spite of the Memory Manager being similar to the VDMA IP core provided by *Xilinx*, we not only achieved the same most relevant error controls but also added an important one to detect if there is data being lost during the writing process to memory. Also, the Memory Manager is not limited to 32 frames to be buffered in external RAM, instead, it can be used all the configured memory region, independently of the number of frames. Moreover, it achieved better specifications using less FPGA resources. In Figure A.1 is shown the result in a picture, as example.

Although we can encode and play encoded video from the simulation developed to test and understand the H.264 Encoder open-source properly, the H.264 Encoder IP core deployed was still not able to encode correctly the video content in the FPGA. The author suspects this problem arises due to a timing problem in driving some signals, which need to meet some criteria still not completely known. Despite this, the encoder works as expected in the simulation, except for 8k, whose result is shown in Figure A.2 and A.3 in the Appendix. Additionally, during the tests, noticed that FHD resolution (1920x1080) would be generating an error inside the encoder. After some tests, concluded both width and height frame size need to be divisible by 16. However, this is not a problem because the FHD is the only standard resolution not divisible by 16. It can easily be solved by excluding the last 8 frame lines. The user would not even notice. All the other goals have been met once the header is working correctly when encoded video is reproduced in *FFMPEG* and the interface is capable to deliver data in real-time. The encoder is configurable and the conversion from YUYV 4:2:2 10-bit color to 4:2:0 8-bit color succeeded as well.

6. Conclusion

The presented thesis revealed an interesting challenge since the very beginning due to the fact of dealing with a state of the art technology to be developed. Once the technologies available to accomplish the 8k video transmission are still very tangential, the implemented solution would require a small delaying margin on hardware designing for this project to succeed. Therefore, the performance was most often the approach chosen, once the board has very large resources to be exploited. Another challenge in performing this work is related to never been working on video processing at the hardware level, bit and byte. Additionally, the volume of work involved to accomplish such goals was perceived already during this MSc thesis, which resulted in more than 5000 of HDL lines coded (2800 fully performed and the missing 2200 are COTS adapted to fit the required system).

For memory management, the system is provided by a Memory Manager capable to run uninterruptedly and able to deal with every problem that may come from the video source. This core was able to read and write frames properly according to the protocols with no errors. An uninterrupted running of the core provides an essential achievement for the whole project once it prevents requiring for a reset, in case of an error from a video source occurs, for running properly. Thanks to the double buffering mechanism developed and the deep exploitation around AXI4 protocol, the memory accesses were optimized as much as possible for memory resource exploitation. The configuration of parameters required was achieved, which allows changing the video parameters on-the-fly by only restarting the core to apply the changes immediately, via control register. A scalable memory region, along with a rotation mechanism, produces a circular buffer effect, which prevents overtaking and overwriting unexpected memory regions. By its side, the status register allows checking at any moment the current state of the core. The mechanism for frame error detection, inherent to the status register, provides useful information about the coherency between the Memory Manager and the video source. Moreover, the data loss error detection developed revealed essential to detect a different edge of the system, once the mechanisms implemented for errors associated to incoherent alignments of frames would not detect such error that may occur in the memory interface. Additionally, the Read subsystem allows setting in the delay between the frames writing and reading. This configuration is performed from a graphical user interface before hardware *Bitstream* generation. In other words, it is possible to configure how much frames the reading will be delayed in the video streaming output, comparing to the writing process. Every task defined to complete the memory management was successfully performed.

The encoding subsystem provided with the H.264 Encoder was able to receive the raw video content from the Memory Manager and deliver encoded video in the H.264 format (although the content is only correctly encoded in simulation). This core is designed with a double buffering mechanism as well but with a much larger number of BRAMs to avoid multiple memory accesses due to specific pixel receiving order to the H.264 Encoder, as explained during this document. The equation developed and implemented for addressing every pixel from buffers can select each buffered pixel in the required order for the H.264 Encoder successfully. Additionally, it was added a header for each encoded frame along with an end of the encoded frame signal to be delivered to the Memory Manager. The simulation is working correctly for every frame size (as long as both width and height are divisible by 16) except for 8k, which is encoding data wrongly according to Figure A.2. Nevertheless, the header is correctly coded, once the video can be played by the *FFMPEG* API. Moreover, the addressing algorithm is reading selectively from buffers as planned and the interface is optimized and capable to deliver content in real-time. Although the open-source encoder used as basis cannot encode 8k video in real-time, the subsystem developed to organize the memory content in MBs is ready to deliver real-time content. This is useful in case of encoder upgrading. Additionally, the parameters are configurable and the video is correctly streamed to Memory Manager as well.

In conclusion, the system developed offers the *MOG WALL SCREEN* project, in which this MSc thesis is inserted, a reliable and efficient system to write and read frames from a video source up to 8k, and it is provided with different edges of error detection mechanisms. Although the H.264 Encoder cannot encode 8k video properly, all the other goals for this IP core has been met.

6.1 Future Work

This thesis belongs to a project that will continue to be developed at least until July 2020. Although most of the work is already completed, there are still some improvements to be added to the solutions implemented. Some new functionalities to be added next for the Memory Manager are:

- **Management and writing of ancillary data -** Raw video and audio are independently saved in memory. Therefore, there is a demand for adding an interface for receiving ancillary data to be saved in memory as well. The data will be received by AXI4-Stream, buffered, and later saved in memory. In a configured circular region, similarly to the method used to save the raw and encoded video.

- **Implement two pointer registers for memory read and write of frames -** Both read and write registers, although those pointer registers could be read by AXI4-Lite, they should be available in the outside of the Memory Manager IP core as well to be directly accessed from other core by direct assignment of wires to each write frames in memory pointer and read frames from memory pointer.

- **Add in the outside of the core a bit for error tracking -** Adding a bit outside of the Memory Manager IP core will avoid the system to be constantly reading the status register for checking if some error may have occurred. It may work as an interruption, for example.

- **Add one control bit for reset error status -** Once the Memory Manager can realign each frame to be written in memory and keep running properly when the system is notified of error it may reset the status register with no need for restarting the core. In Figure A.3 is presented a print-screen of a succeeded encoded frame in 4k.

Some next steps to be taken in the H.264 Encoder are:

- **Fix the color issue in 8k video encoding -** Whereas some texture of frames is perceptible, there is a problem in encoding video correctly. Therefore, the next step is to find the issue that may be originating faulty color and texture and correct it.

- **Add mechanism of auto-readjustment of width and height frame to be divisible by 16 -** Since the encoder is using 16x16 MB, a frame resolution in which the width or height are not divisible by 16 would disrupt errors both in the encoder and decoder. For example, the FHD resolution (1920x1080) generates an error once the height is not divisible by 16. A possible solution is to exclude the last 6 lines. The frame would become (1920x1072) and no error would be set, neither in the encoder nor the decoder. It may be immediately prepared since the configuration subsystem (by passing already the value of 1072 to the next layer), and the last 6 lines would be discarded before the buffering process.

- **Keep dissecting the header -** Go deeper in the header understanding and add more reconfigurable variables that may allow some important features in the future.

A. Appendix

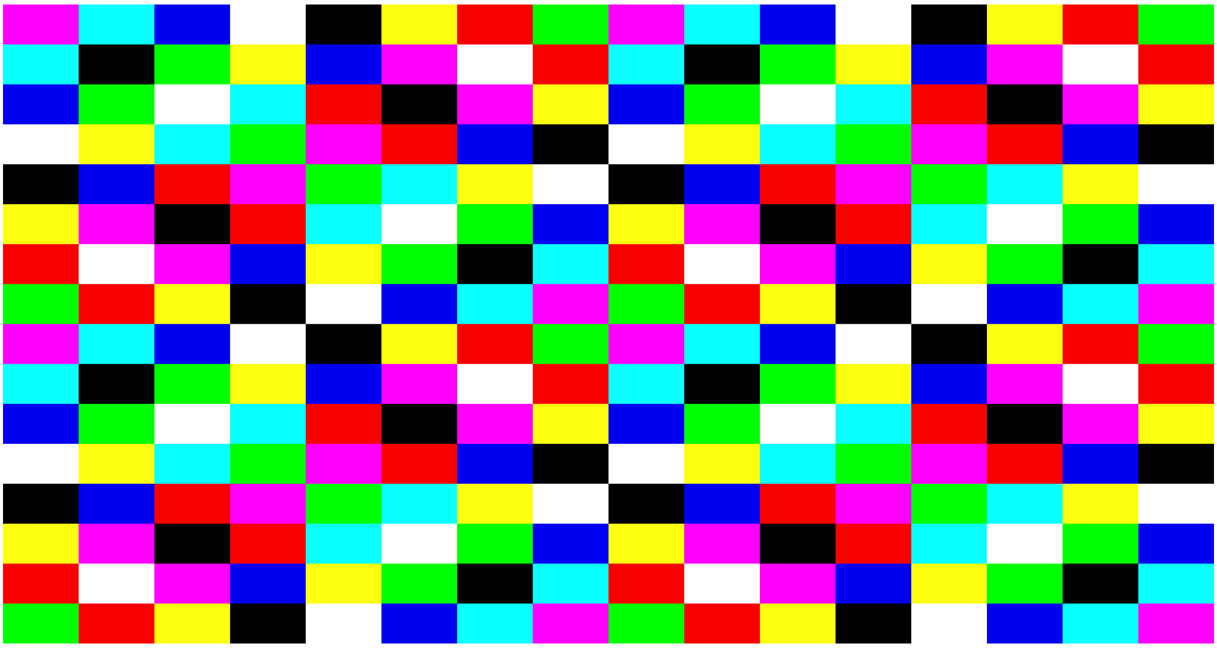


Figure A.1: 8k frame written and read by Memory Manager.

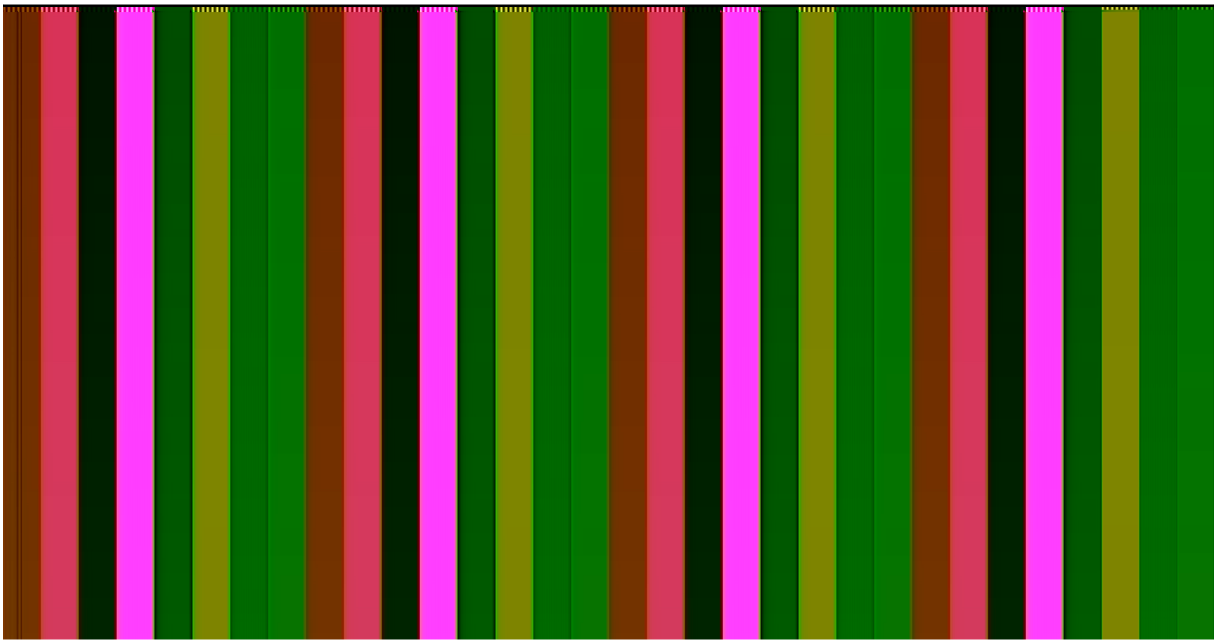


Figure A.2: 8k encoded frame in format H.264 failed.

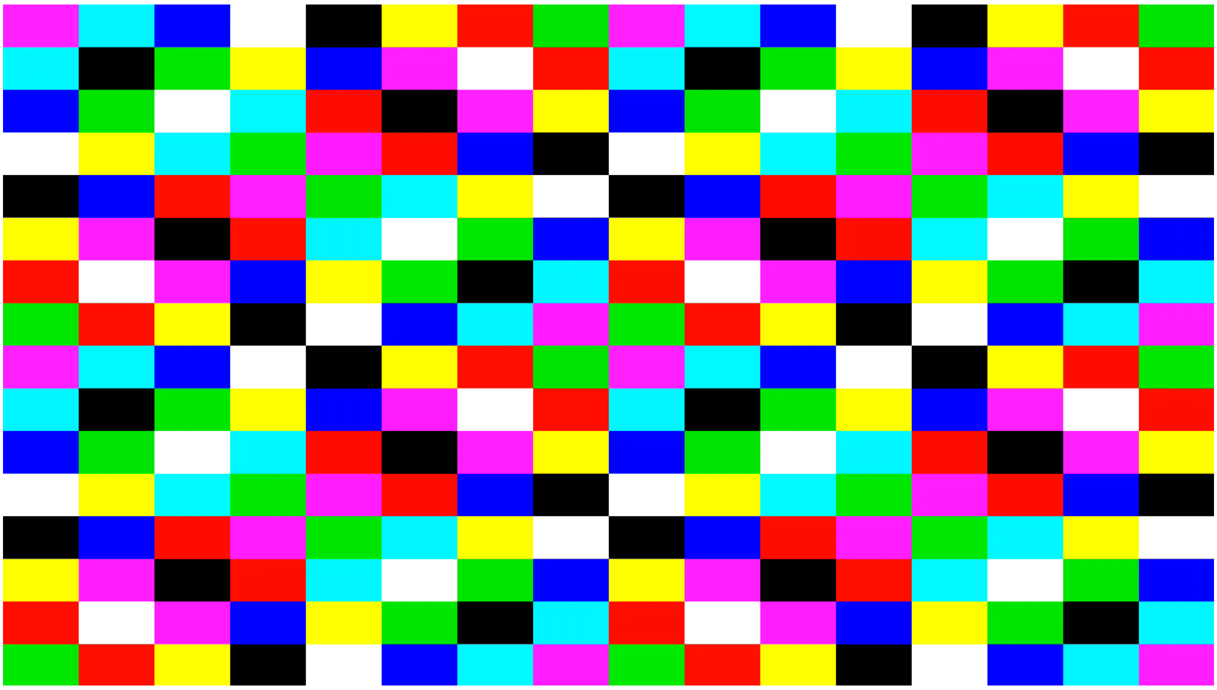


Figure A.3: 4k encoded frame in format H.264 succeeded.

Glossary

Exponential Golomb A type of universal code for encode any non-negative integer.

IDR (Instantaneous Decode Refresh) a Network Abstraction Layer unit of the H.264 standard. The presence of the IDR NAL tells the decoder that the next frame is coded independently from any previous picture, as a new fully encoded frame (IDR frame).

PPS Is a Network Abstraction Layer unit of the H.264 standard. Contains parameters applyed on the decoding of one or more single pictures inside a coded video sequence.

RGB Additive color model composed of three primary colors: (i) red, (ii) green, and (iii) blue. They are added together in various ways to reproduce a broad array of colors.

SPS A Network Abstraction Layer unit of the H.264 standard. It contains parameters applied on a series of consecutive pictures from a coded video.

YUV Color encoding system, an alternative to RGB. It encodes a color image based on human perception, reducing the bandwidth for chrominance components.

Bibliography

[1] J. Hudson and N. Seth-Smith, "3G: The Evolution of the Serial Digital Interface (SDI)," *SMPTE Motion Imaging Journal*, vol. 115, no. 11-12, pp. 472–481, 2012.

[2] Xilinx, "LogiCORE IP AXI Video Direct Memory Access v5.04a," Tech. Rep., 2012.

[3] M. Ghandi and M. Ghanbari, "The H.264/AVC Video Coding Standard for the Next Generation Multimedia Communication," *IAEEE Journal*, 2004.

[4] A. V. Systems, "Kona ip," [Online]. Available: https://www.aja.com/products/kona-ip, Accessed on: Sep. 20, 2019.

[5] Dektec, "Dta-2179," [Online]. Available: https://www.dektec.com/products/PCIe/DTA-2179/, Accessed on: Sep. 20, 2019.

[6] Magewell, "Pro capture aio 4k plus," [Online]. Available: http://www.magewell.com/ products/pro-capture-aio-4k-plus, Accessed on: Sep. 20, 2019.

[7] Bluefish444, "Kronos optikos," [Online]. Available: https://bluefish444.com/ products/developer-cards/details/kronos-range/23/kronos-range.html, Accessed on: Sep. 20, 2019.

[8] Deltacast, "Delta-3g-elp-tico-d 4c," [Online]. Available: https://www.deltacast.tv/ products/developer-products/sdi-cards/delta-3g-elp-tico-d-4c, Accessed on: Sep. 22, 2019.

[9] Xilinx, "ZCU102 Evaluation," vol. 1182, pp. 1–120, 2018.

[10] S. S. Math and R. B. Manjula, "Design of AMBA AXI4 protocol for System-on-Chip communication," *International Journal of Communication Network and Security (IJCNS)*, 2012.

[11] J. Ajanovic and I. Corporation, "PCI Express rev 3.0 Accelerator Features," pp. 1–10, 2008.

[12] IDG, "Pcie 4.0: From specs to compatibility," [Online]. Available: https://www.pcworld.com/article/3400176/pcie-40-everything-you-need-to-know-specs-compatibility.html, Accessed on: Sep. 22, 2019.

[13] ARM, “AMBA AXI and ACE Protocol Specification,” *Arm*, 2011.

[14] J.-W. C. J.-W. Chen, C.-Y. K. C.-Y. Kao, and Y.-L. L. Y.-L. Lin, “Introduction to H.264 advanced video coding,” 2006.

[15] S. Pinto, T. Gomes, J. Pereira, J. Cabral, and A. Tavares, “IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices,” *IEEE Internet Computing*, 2017.

[16] T. Gomes, F. Salgado, S. Pinto, J. Cabral, and A. Tavares, “A 6LoWPAN Accelerator for Internet of Things Endpoint Devices,” *IEEE Internet of Things Journal*, 2018.

[17] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares, “Towards a TrustZone-assisted hypervisor for real-time embedded systems,” *IEEE Computer Architecture Letters*, 2017.

[18] J. de Almeida, M. A. Eliseo, C. I. da Silva, H. Prates, V. V. Poser, B. Stalbaum, and N. G. Furtado, “Sensemaking: A Proposal for a Real-Time on the Fly Video Streaming Platform,” *Creative Education*, 2016.

[19] H. Yamashita, H. Aoki, K. Tanioka, T. Mori, and T. Chiba, “Ultra-high definition (8K UHD) endoscope: our first clinical success,” *SpringerPlus*, 2016.

[20] M. Kitamura, D. Shirai, K. Kaneko, T. Murooka, T. Sawabe, T. Fujii, and A. Takahara, “Beyond 4K: 8K 60p live video streaming to multiple sites,” *Future Generation Computer Systems*, 2011.

[21] P. H. Putman, “Display technology: The next chapter,” in *SMPTE Motion Imaging Journal*, 2016.

[22] D. Zhou, S. Wang, H. Sun, J. Zhou, J. Zhu, Y. Zhao, J. Zhou, S. Zhang, S. Kimura, T. Yoshimura, and S. Goto, “An 8K H.265/HEVC Video Decoder Chip With a New System Pipeline Design,” *IEEE Journal of Solid-State Circuits*, 2017.

[23] H. Takizuka, T. Torikai, A. Mitsui, H. Suzuki, Y. Watanabe, T. Toma, and Y. Koike, “A study on 120 GBPS GI-POF interface for 8k-UHD video transmission,” in *Proceedings of 22nd International Conference on Plastical Optical Fibers, POF 2013*, 2013.

[24] S. Gohshi, “4K-to-8K TV Up-converter with Super Resolution,” *NAB BEC Proceedings*, 2014.

[25] Y. H. Park, J. Kim, M. Kim, W. Lee, and S. Lee, “Programmable multimedia platform based on reconfigurable processor for 8K UHD TV,” *IEEE Transactions on Consumer Electronics*, 2015.

[26] S. Murthy and B. Sujatha, “Multi-Level Optimization in Encoding to Balance Video Compression and Retention of 8K Resolution,” *Perspectives in Science*, 2016.

[27] C. Sarojini and J. Thangaraj, “Implementation and Optimization of Throughput in High Speed Memory Interface Using AXI Protocol,” in *2018 9th International Conference on Computing, Communication and Networking Technologies, ICCCNT 2018*, 2018.

[28] A. Corporation, “White Paper Lower Costs in Broadcasting Applications With Integration Using FPGAs,” no. February, pp. 1–5, 2006.

[29] J. Hudson, N. Seth-Smith, and R. Conrod, “Uhd in a hybrid sdi/ip world,” *SMPTE Motion Imaging Journal*, 2016.

[30] N. Seth-Smith, “Flexible SDI - The Universal Transport for Streamed Media,” vol. 1100, pp. 1–23, 2017.

[31] D. Overview, “6G-SDI Bit-Serial Interfaces — Overview for the SMPTE ST 2081 Document Suite,” vol. 10601, no. 914, 2016.

[32] Society of Motion Picture and Television Engineers, “OV 2082-0:2016 - SMPTE Overview Document - 12G-SDI Bit-Serial Interfaces — Overview for the SMPTE ST 2082 Document Suite,” *32NF - Technology Committee on Network and Facilities Infrastructure*, vol. 10601, no. 914, 2016. [Online]. Available: https://ieeexplore.ieee.org/servlet/opac?punumber=7565448

[33] D. J. Katz and R. Gentile, “Direct Memory Access,” in *Embedded Media Processing*, 2006.

[34] Xilinx, “Logicore ip axi direct memory access v7.1,” pp. 1–95, 2018. [Online]. Available: www.xilinx.com

[35] S. Elzinga and C. Martin, “Bridging Xilinx Streaming Video Interface with the AXI4-Stream Protocol,” *Xilinx Application Notes*, 2012.

[36] M. Gupta and A. K. Nagawat, “Design and implementation of high performance advanced extensible interface(AXI) based DDR3 memory controller,” in *International Conference on Communication and Signal Processing, ICCSP 2016*, 2016.

[37] M. Tkalcic and J. Tasic, “Colour spaces: perceptual, historical and applicational background,” in *The IEEE Region 8 EUROCON 2003. Computer as a Tool.*, vol. 1. IEEE, 2003, pp. 304–308. [Online]. Available: http://ieeexplore.ieee.org/document/1248032/

[38] S. H. Bae, J. Kim, M. Kim, S. Cho, and J. S. Choi, “Assessments of subjective video quality on HEVC-encoded 4K-UHD video for beyond-HDTV broadcasting services,” *IEEE Transactions on Broadcasting*, vol. 59, no. 2, pp. 209–222, 2013.

[39] M. U. K. Khan, J. M. Borrmann, L. Bauer, M. Shafique, and J. Henkel, “An H.264 Quad-FullHD Low-Latency Intra Video Encoder,” no. March, pp. 115–120, 2013.

[40] D. Marpe, T. Wiegand, and G. J. Sullivan, “The H.264/MPEG4 advanced video coding standard and its applications,” *IEEE Communications Magazine*, 2006.

[41] J. Gong, T. Wang, J. Chen, H. Wu, F. Ye, S. Lu, and J. Cong, “An efficient and flexible host-FPGA PCIe communication library,” in *Conference Digest - 24th International Conference on Field Programmable Logic and Applications, FPL 2014*, 2014.

[42] S. R. Mantripragada and P. Mopuri, “Verifying performance of PCI express in a system for multi giga byte per second data transmission,” in *Proceedings of the International Conference on Communication and Electronics Systems, ICCES 2016*, 2016.

[43] P. S. Baranov and L. I. Ivanov, “High-quality UHD demosaicing on low-cost FPGA,” *Proceedings of the 2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering, ElConRus 2018*, vol. 2018-Janua, pp. 277–280, 2018.

[44] Xilinx, “Zynq UltraScale + Device,” vol. 1085, pp. 1–1181, 2018.

[45] ——, “AXI Reference Guide,” vol. 612, pp. 1–186, 2011.

[46] ——, “AXI4-Stream Video Ip and System Design Guide,” 2016.

[47] B. Cattle, “Vhdl hardware h.264 video encoder,” [Online]. Available: https://github.com/bcattle/hardh264, Accessed on: Oct. 28, 2019.

[48] C. J. Madsen, “Visual binary difference,” [Online]. Available: https://www.cjmweb.net/vbindiff/, Accessed on: Oct. 29, 2019.

[49] FFMPEG, “Ffmpeg api,” [Online]. Available: https://www.ffmpeg.org, Accessed on: Oct. 29, 2019.