High-Throughput Data Interfacing
for Real-Time Medical Imaging Applications

Theofanis Floros

Delft University of Technology

High-Throughput Data Interfacing

for Real-Time Medical Imaging Applications

by

Theofanis Floros

in partial fulfillment of the requirements for the degree of

Master of Science
in Embedded Systems

at the Delft University of Technology,
to be defended publicly on Friday February 19, 2021 at 13:30.

Student number:	4934385
Supervisor:	Rob de Jong
	Dr. Zaid Al-Ars

This thesis is confidential and cannot be made public until February 11, 2022.

Abstract

Current high-end medical X-ray intervention devices provide a tremendous amount of high-definition images per second. Combined with the additional inputs from the numerous auxiliary devices, processing and compositing the data in real-time quickly becomes an arduous engineering challenge. Furthermore, the critical nature of this application domain allows no room for error, whereas the process has to be simultaneously flexible enough to permit the unhindered work of the corresponding medical professional.

Such medical imaging applications presently rely on complete hardware implementations of their processing pipelines and compositor engines, as opposed to earlier adopted paradigms of hardware-accelerated solutions. Said engines are usually realised in FPGA platforms due to their massively parallel computational capabilities and exceptional connectivity. However, given the vast amount of input streams arriving at the compositor's pipeline, switching and routing to available processing resources has remained a persistent issue. Present commercial solutions prove to be either too costly, too slow or too inflexible for developers to consider. On the other hand, academic literature centred around the subject is severely lacking.

This thesis proposes a unique hardware architecture solution to mitigate the aforementioned problem. The proposed architecture, developed in collaboration with Philips Healthcare and TU Delft, exhibits an average end-to-end latency of 100 ns and a throughput of 7.2 Gb/s. Additionally, these results were realised under marginal resource utilisation, proving that such a switch can fit in the FPGA device and, subsequently, be developed as an integrated part of the compositor engine. Finally, the limited amount of reserved memory resources allowed for near-linear scalability of the proposed design should an increased amount of I/O devices be added; as well as dynamic portability should migration to smaller devices be a concern.

Preface

The present document marks the completion of my student obligations at TU Delft. This project began in February 2020, when the world started entering its current seclusive state. Adapting to working from a room and meeting people online has not been without difficulty, but I am proud to say that the quality of the work has not suffered. Furthermore, this journey has proven to be an incredible learning opportunity for me. Many people either directly or indirectly helped bring this project to fruition and I would like to seize this opportunity to thank them.

First of all, I would like to extend my most sincere gratitude to my supervisors **Rob de Jong** and **Zaid Al-Ars** for supporting me every step of the way. Such exceptional support cannot be taken for granted and I could have hardly asked for better mentors.

Last but not least, I am very thankful towards my friends, without whom this would have been a very different and far less interesting journey. In no particular order, thank you Paulina, Vipul, Greg, Chief, Giannis, Julian, Jure, Anastasia, Christos and many many more for being a part of my life and providing some much needed respite. Finally, I am grateful to my parents for backing me through my long years of studying, for their patience and for their love.

Theofanis Floros
Delft, January 2020

Contents

1 Introduction

In the rapidly evolving domain of medical technology, technological advancements play a paramount role in ensuring people's wellbeing. Every innovative contribution is directly responsible for improving and saving numerous lives around the globe. This broad field covers areas such as biotechnology, pharmaceuticals, information technology, the development of medical devices as well as the domain of medical imaging. Medical imaging is the technique and process of creating visual representations of the interior of a body for clinical analysis and medical intervention. As a discipline and in its widest sense, it incorporates technologies such as X-ray radiography, magnetic resonance imaging (MRI), ultrasound and positron emission tomography (PET) among numerous others. In recent years, as these technologies become more sophisticated and push the boundaries of what is possible, the computational demand needed to enable them has grown exponentially. This fact has shifted the research and development direction of medical imaging technologies towards hardware-accelerated solutions such as GPU and FPGA processing, in order to cope with the ever-increasing computing challenges.

This thesis details the investigation and implementation of a switching component embedded within such a hardware-accelerated solution, targeted towards high-throughput X-ray imaging devices, developed by Philips. Since the intended application of the aforementioned switching component was of critical nature, it had to ensure nominal operation under very high input loads while adhering to strictly defined requirements. Furthermore and more importantly, the component had to be realised in a greatly resource efficient and inexpensive manner, rendering the ensuing analysis exceedingly difficult. Unfortunately, in addition to the preceding challenges, the entire project had to be carried out remotely due to unforeseen obstacles as a result of the COVID-19 pandemic plaguing the world. This resulted in the development team being unable to physically meet or have access to the required hardware.

The current chapter serves as an introduction to the problem. It begins by providing context to the project as a whole through a presentation of Philips, followed by an establishment of the general framework of FitOptiVis. Subsequently, the problem statement of the present work is described, succeeded by its translation to concrete requirements and research questions, which constitute the target objective of the thesis. Finally, the outline of the rest of the work is given.

1.1. Context

1.1.1. Philips Healthcare

Koninklijke Philips N.V. (Royal Philips, most commonly known as Philips) is a Dutch technology company primarily focused on improving people's health and enabling better outcomes across the health continuum. Their activities are aimed towards healthy living and range from prevention to diagnosis, treatment and home care. Founded in Eindhoven in 1891 by Gerard Philips and his father Frederik, the company is a leader in diagnostic imaging, image-guided therapy, patient monitoring and health informatics. Philips leverages advanced technology and deep clinical and consumer insights to deliver integrated solutions. These are centred around both clinical informatics and imaging systems which includes ultrasound, PET, Magnetic resonance imaging (MRI) and X-Ray imaging among various others [1].

During my internship I had the honor to work in the Image Guided Therapy (IGT) team in Philips Healthcare at Best. The group is aimed towards providing integrated solutions that advance minimally invasive procedures. Their end products help healthcare providers to decide, guide, treat and confirm the right care in real time during the procedure to enable better outcomes for each patient and at the population level. The development process of the IGT team is heavily geared towards reducing total costs and optimising resource utilisation. As such, those were the principal driving initiatives of the current project as well.

1.1.2. FitOptiVis

FitOptiVis [2] is an EU subsidy project which addresses smart integration of image- and video-processing pipelines for Cyber-Physical Systems (CPS). Smart systems integration is one of the essential capabilities required to maintain and to improve the competitiveness of the European industry in important application domains. This is especially relevant for CPS that increasingly are autonomous distributed integrations of electronic systems, subcomponents and software that are tightly interacting with physical systems and their environment. Several of these are smart systems themselves. Novel design approaches and methodologies are requested to address this new class of computing systems. Images play a central role in human perception and understanding of our environment. In the same way, CPS need visual context and awareness to make the correct decisions (autonomy) and take the appropriate actions (actuation and adaptation). However, advanced image and video processing is compute intensive, whereas for adequate behaviour the results need to be available with low latencies and high throughput. In many cases, several devices of a distributed CPS need to operate with low energy, as power sources might be scarce, or heat dissipation has to be limited to protect the system, its environment or humans interacting with it. Optimisation for other quality aspects, such as security, safety, complexity, or adaptability, is also important.

It is challenging to balance power demand versus performance of the increasingly complex distributed configurations, reflected in the growing number of sensors, actuators and other smart devices, their heterogeneity, their growing autonomy, and the increased need for performance. To address this challenge, FitOptiVis aims to provide end-to-end multi-objective optimisation for imaging and video pipelines of CPS, with an emphasis on energy and performance.

The main objective of FitOptiVis is to develop an integral approach for smart integration of image- and video-processing pipelines for cyber-physical systems covering a reference architecture, supported by low-power, high-performance, smart devices, and by methods and tools for combined design-time and run-time multi-objective optimisation within system and environment constraints.

The proposed integrated FitOptiVis approach will target end-to-end energy and performance optimisation of distributed and adaptive image and video processing pipelines, taking care of the entire system from the edge to the cloud. The targeted use cases by which the project derives its main requirements, architecture, designs, implementation, demonstration and validation are listed below:

- WATER SUPPLY, Lead by: Societa Acquedotti Tirreni (SAT) and Aitek
- VIRTUAL REALITY, Lead by: Nokia
- HABIT TRACKING, Lead by: RGB
- 3D INDUSTRIAL INSPECTION, Lead by: Instituto Tecnologico de Informatica (ITI)
- ROAD TRAFFIC SURVEILLANCE, Lead by: CAMEA
- MULTI SOURCE STREAMING COMPOSITION, Lead by: Philips

- SUSTAINABLE SAFE MRI, Lead by: Philips
- ROBOTS CALIBRATION, Lead by: REX
- SURVEILLANCE OF SMART-GRID CRITICAL INFRASTRUCTURE, Lead by: Seven Solutions (7SOLS)
- AUTONOMOUS EXPLORATION, Lead by: Thales Alenia Space (TASE)

The current project was focused around the *Multi Source Streaming Composition* use case and as such will be the sole instance which will be elaborated upon, with the rest of the cases being out of scope for the current work. Consequently, all the constraints and requirements presented below will be exclusively determined from the aforementioned use case.

1.2. Problem statement

Multi source streaming composition refers to the collection of various image streams from different image sources, the compilation of said sources and their mapping to a desired configuration as well as their projection to the available output devices, e.g. a multitude of screens. The aforementioned image sources in this case correspond to the visual data supplied by various medical devices. Additionally, in order for the configuration to be both flexible and accurate, the image streams undergo some elemental processing in the form of image scaling. The objective of the formerly mentioned use case, which encompasses the current development directive of the IGT group, seeks to implement a fast, resource efficient, low-energy, flexible device that can effectively perform the predescribed task. Figure 1.1 illustrates the hardware architecture concept of the compositor engine whereas Figure 1.2 depicts the general assortment of various medical devices which provide input streams to said compositor. As of the time of writing, the various components of the aforementioned architecture are under different stages of development, whereas the platform used for the realisation of the core compositor engine is a Xilinx Ultrascale+ Field Programmable Gate Array (FPGA) board. The necessary background information regarding these devices is presented in Section 2.1 below, whereas the general acceleration paradigm is discussed in Section 2.2.

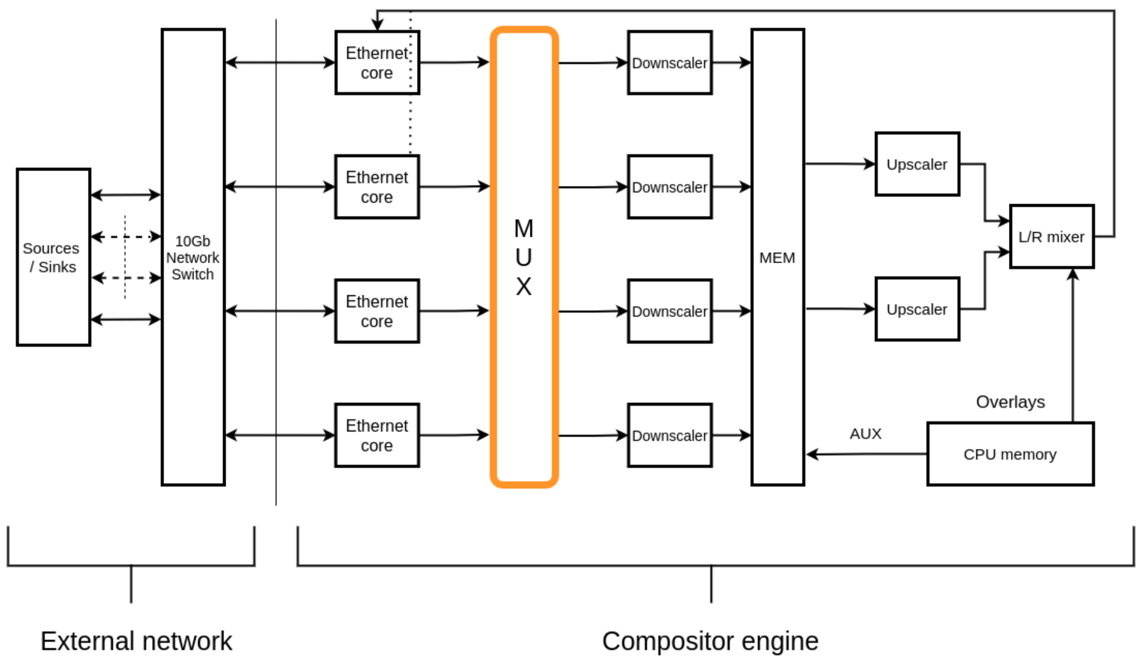


Figure 1.1: Hardware architecture concept of the compositor engine

The current work focuses entirely on the multiplexer (MUX) component of the engine, which is responsible for the mapping and forwarding of the various input streams originating from the Ethernet cores to the downscalers. As such, the overall functionality of the entire compositor will not be detailed further, as it was deemed out of the scope of this document. While the name "MUX" was reserved for the component during the conceptualisation phase, it does not fully reflect on its intended functionality. A hardware multiplexer acts as a simple selector of its given inputs, choosing one of them over assertion of a `select` signal and forwarding it to the output in a Nx1 manner. An NxN mapping is already well beyond the typical capabilities of a MUX. Such an operation requires careful inspection and subsequent forwarding of incoming packets, akin to the functional behaviour of a router or a network switch, as further described in Section 2.3. Furthermore, the module is required to incorporate a specialised functionality similar to multicasting, i.e. a transmission of data from a single source to multiple recipients, which is detailed in the following section. This added group communication requirement further solidifies the identity of the module being closer to a switch rather than a MUX. Additionally, The Ethernet cores, as shown in Figure 1.1 can contain up to 4 independent streams each. Said streams, as well as their containing fibres, all exhibit entirely asynchronous behaviour. To complicate matters further, each individual stream may arbitrarily begin or end transmission at any given point in time, which may also cause the entire mapping configuration of streams-to-scalers to adjust accordingly. The module needs to not only be able to accommodate for this case, but also do so with virtually no delay. The maximum allowed end-to-end latency for entire compositor falls under 1ms, meaning that the routing mechanism should be near instantaneous, allowing for no internal processing or significant amounts of buffering. Namely, the module is required to constantly perform at full capacity while managing changes to the configuration without disturbing the flow of transmission. Finally, the developed module needs to be resilient to potential network jitter and ensure that it exhibits no destructive behaviour under any error-induced circumstances due to the severely critical nature of its intended application.

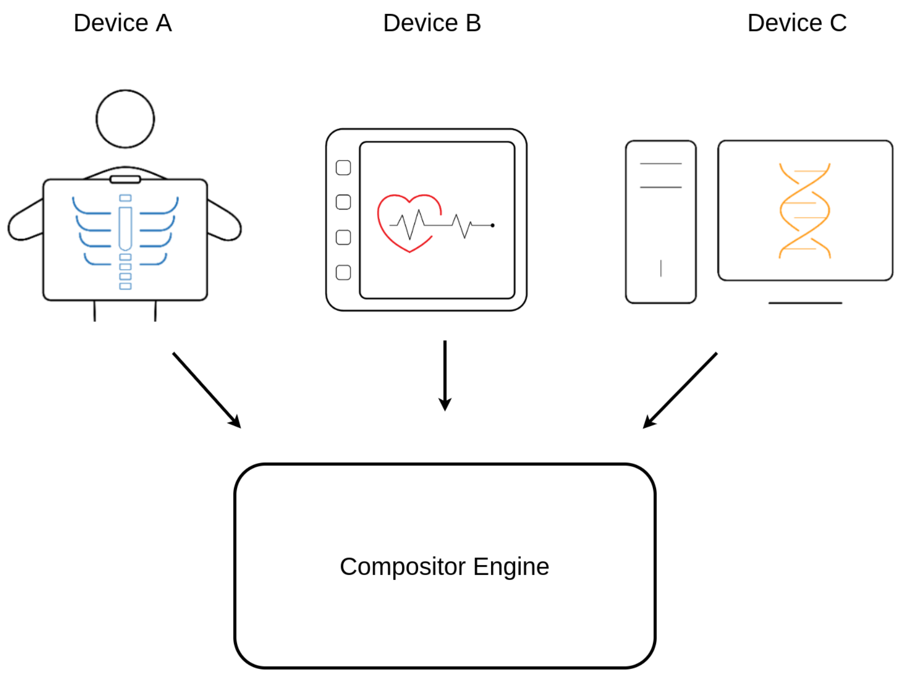


Figure 1.2: Scaling and compositor engine inputs from various medical devices

These prerequisites of the module are explicitly listed and quantified in Section 1.2.2 below. While, as mentioned above, the remaining components functionality will not be detailed, some of their limitations need to be identified as they directly define the development requirements. Namely, those include the number of Ethernet fibres in the system, the number of downscalers as well as the architectural limitations of both. These strict demands imposed to the "MUX" by the nature of the system as well as it being the component that interfaces the compositor with the external network, render the "MUX" a major possible bottleneck in the design of the overall engine. Developing a component that performs under the imposed constraints subsequently enables the nominal operation of the rest of the system. For the remainder of this thesis, in order to avoid confusion with classic multiplexers and for lack of a better term, the module will simply be dubbed "the Switch".

1.2.1. Hardware constraints

Since the general functionality of the Switch has been outlined, some of the finer architectural elements that impose limitations on the Switch's design need to be addressed. As briefly mentioned before, these restraints stem from the module's immediate environment, i.e. the Ethernet cores, the downscalers and the FPGA device that encompasses the core compositor engine. To begin with, the Ethernet cores, as depicted in Figure 1.1, consist of standard full-duplex Ethernet fibres transmitting frames at a rate of 10 gigabits per second (10Gb). The bandwidth is further divided into 64-bit wide packets transferred at 156.25 MHz. An artificial restriction is placed upon the Ethernet fibres, limiting the maximum bandwidth utilisation of the contained streams at 7 Gb due to the presence of sideband and control data. As long as this bandwidth constraint is not violated, however, each fibre can contain up to 4 individual, uniquely identifiable streams. The streams themselves, in turn, are not bound by standard video resolutions and can be instead composed of any arbitrary number of pixels for their width and height. That said, each stream cannot exceed 60 frames per second (fps and) the Switch is not required to handle 4K resolutions, or any steam whose line width is larger than 1920 pixels, on its initial release. However, this functionality needs to be implemented in later versions as per specification, so the design should account for this eventual functional expansion. Finally, the designed module needs to be able to handle 4 such Ethernet fibres at launch, but should be likewise expansible to accommodate at least 8.

On the other hand, the downscalers, which are incidentally the most refined part of the compositor currently, enforce additional hardware restrictions of their own. Similar to the Ethernet fibres, 4 downscalers are available at launch but the compositor will soon be broadened to incorporate 7. As such, the switch must also be designed in a way that can contain the maximum traffic. Both the horizontal and vertical dimensions of a stream are divided into 32 segments of 128 pixels each at the input of a scaler. When all segments are in use, no additional stream can be processed. These constraints are encapsulated in the following formula:

$$\sum \frac{ceil(line\_width/height_i)}{128} \leq 32$$

Each scaler can additionally support up to 4 streams as long as their combined height or width does not oppose this formula. In other words, any physical scaler can support up to 4 virtual scalers. Furthermore, being part of the FPGA device, the downscalers operate with a driving clock of 300 MHz, almost twice as fast as the Ethernet cores. The Switch is additionally responsible for bridging the frequency domain difference between the input streams and the FPGA device, with the two clocks being completely unassociated with each other.

1.2.2. Requirements

In order to demonstrate some of the previously described functionality, Figure 1.3 showcases a possible output configuration on three different displays. The illustration includes input image streams from various medical devices, as previously seen in Figure 1.2, being processed and projected to 3 individual screens.

The output targets include a large Ultra High Definition (UHD) screen with a resolution of 3840 x 2160 pixels and two smaller Quad High Definition (QHD) screens supporting a resolution of 2560x1440 pixels each. Since their combined resolution does not exceed two 4K screens, it can be supported by 4 Ethernet fibres. This restriction stems from the fact that a single 10 Gb Ethernet fibre is not capable of containing a 4K stream at 60 fps. Thus, such a stream is equally divided and subsequently transported by two such fibres. With the future extension of additional Ethernet fibres, the ability of the system would be upgraded to include support for more 4K resolution screens. Furthermore, as can be seen in the aforementioned figure, a single stream can be replicated numerous times on both the same and different devices, as is evident for all the input streams of the provided example. These replications can further be under varied scaling factors, not necessarily preserving the aspect ratio of the original stream, as is the case for Device C in the figure. Various other configurations of displays is also possible, with every display having the ability to be segmented in a near arbitrary way, barring architectural

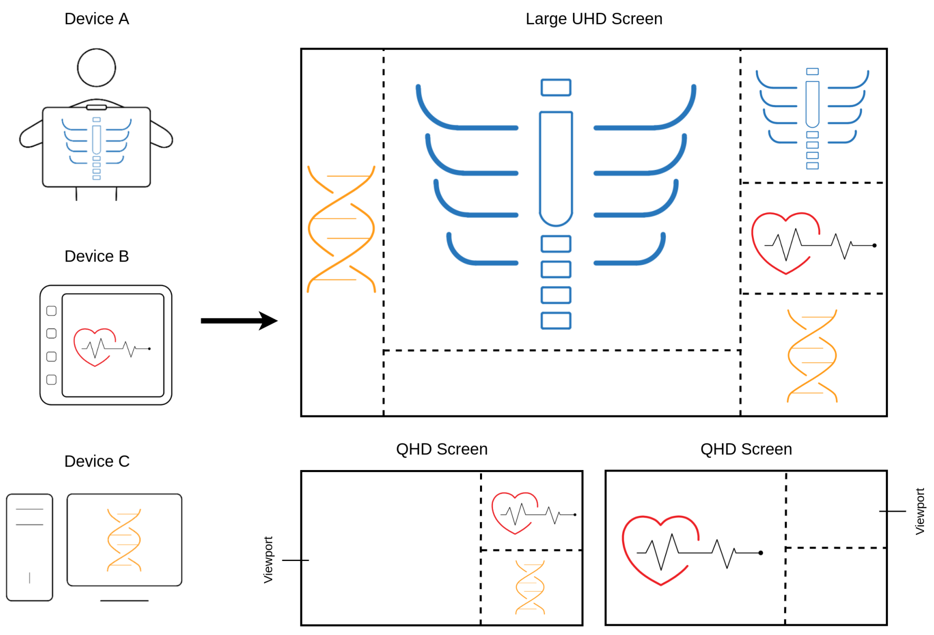


Figure 1.3: Possible display configuration of the compositor engine on three output screens

constraints of the provided streams.

These previously analysed hardware constraints and architectural limitations of the desired module can be formulated in a set of concrete requirements, which the Switch should ultimately meet by the end of its design cycle in order to ensure the appropriate intended functionality. These are the following:

- Each viewport requires a virtual port of a physical scaler in order to be processed.
- Any given stream originating from any fibre can be mapped to any downscaler that has the required available segments.
- A single stream should be able to be mapped to a minimum of 4 different scaling factors in any combination of physical or virtual downscalers, as long as the corresponding bandwidth limitation is not exceeded.
- A single stream may be reproduced up to 8 different times on a screen at the same or different scaling factors, per specification.
- The compositor engine should provide initial support for up to 2 large screens of 3840 x 2160 (4K) resolution, or any combined resolution that does not exceed this amount, when 4 Ethernet fibres are providing input. As such, the Switch must be able to handle the transportation of the corresponding image stream load.
- The Switch must be able to accommodate for asynchronous input on the fibres and manage modifications in the mapping configuration without disrupting the flow of transmission of any of the contained streams.
- The module must be resilient to possible network jitter as well as timing conflicts and timing collisions on both its inputs and outputs. Likewise, it must not be susceptible to any error-related hazard that would impede the functionality of the system as a whole.
- The produced architecture must consistently perform at full capacity and must not stall the compositors operation.
- The design must be scalable so as to integrate future support for an increased number of input Ethernet fibres, downscalers and higher supported image resolutions.

- FitOptiVis is a project heavily focused around resource management and resource reduction. Thus, the proposed design must come at an absolute minimal resource utilisation and cost on any development platform of choice.

It can be derived that the resulting requirements closely resemble those of a high bandwidth multicast-capable network switch. Figure 1.4 depicts how the intended configuration displayed figure 1.3 would reflect in possible a stream-to-downscaler mapping. Once again, this is simply one of many potential configurations that is not violating the requirements presented above and is intended as an example. However, a variety of feasible mapping cases can be identified in said example. These include a single stream (heart monitor marked in red colour) mapping to a single scaler at 3 different scaling factors, a stream (DNA, marked in yellow) being routed to 3 different scalers as well as a single downscaler handling 4 streams, i.e. one replication of the blue stream (X-ray) and 3 replications of the red one.

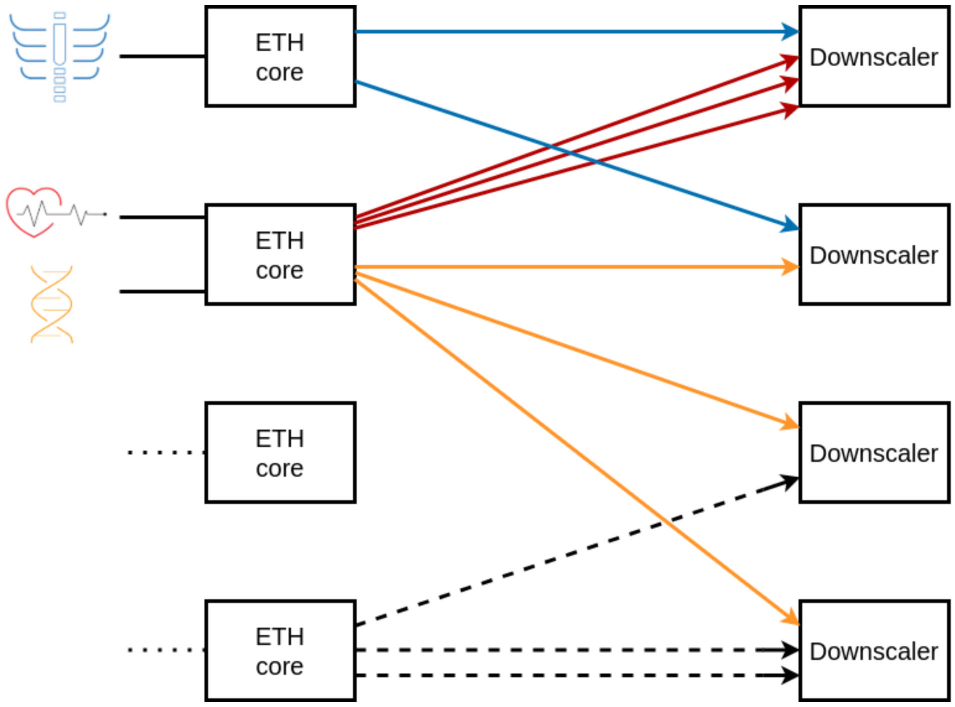


Figure 1.4: Possible downscaler mapping corresponding to the previous configuration example

1.2.3. Research questions

Since the given problem, its accompanying constraints as well as the requirements that have to be met have been introduced, they should be distilled into a higher abstraction level in order to formulate the research questions associated with the current thesis. Namely, would it be possible to develop a real-time constrained hardware implementation of a switching module that would:

- Be able to provide the necessary switching functionality on the input streams while constantly performing at maximum capacity under very strict timing requirements.
- Only marginally increase the latency of the overall compositor pipeline, pushing the state of the art in high frequency packet exchange.
- Shift between operational frequency domains without disrupting the image processing pipeline.
- Be realised under minimal resource utilisation, especially concerning scarce memory resources, to possibly fit in the FPGA device along with the rest of the compositor.
- Be portable enough to render migration of the design to smaller devices possible and scalable enough to accommodate for a future expansion of inputs and outputs.

- Come at a low overall cost.

The rest of this document will try to provide an answer to the above questions.

1.3. Structure of the report - outline

The remainder of this report is structured as follows. Chapter 2 provides an overview of the required background knowledge necessary to interpret and follow the work conducted in the present thesis. In Chapter 3, the potential alternative solutions to the proposed design are presented, compared and contrasted. The complete architectural implementation of the proposed solution is meticulously elaborated upon in Chapter 4. Chapter 5 reflects on the results ultimately achieved by the design. The work concludes in Chapter 6, where the project is discussed briefly in its entirety and prospective future efforts are examined.

2

Background

This chapter introduces the necessary background information that, after an elaborate evaluation of the various requirements, establishes the basis needed to further understand the work conducted in this thesis. This includes an introduction on the basics of FPGAs, which the core of the compositor engine is based on, their advantages and applications as well as the typical accompanying development cycle. It is followed by a brief summary on hardware acceleration for streaming interfaces and the available acceleration platforms. The chapter concludes with a presentation of network switches, succeeded by their uses and typical architectures.

2.1. FPGAs

The fundamental architecture along with the basic internal structure of an FPGA is depicted in Figure 2.1, whereas the finer architectural details are illustrated in Figure 2.2 below. Field Programmable Gate Arrays (FPGAs) are semiconductor integrated circuits that are based around a matrix of configurable logic blocks (CLBs) connected via a hierarchy of "reconfigurable interconnects". CLBs themselves consist of look-up tables (LUTs) as well as other logic and memory elements such as registers or logic gates. In addition, recent generations of FPGAs also come equipped with dedicated hardwired processing blocks such as embedded processors, media access control address (MAC) units, and embedded memory. LUTs are perhaps the core element of CLBs as they can be programmed to store any given output for each combination of their input signals. They provide FPGAs with the ability to implement any logical function [3]. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing, hence the term "field programmable". This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks. [4] The FPGA configuration is generally specified using a hardware description language (HDL), similar to that used for an ASIC. The programming can be a single, simple logic gate (an AND or OR function), or it can involve one or more combinations of complex functions, ranging all the way to functions that, together, act as a comprehensive multi-core process.

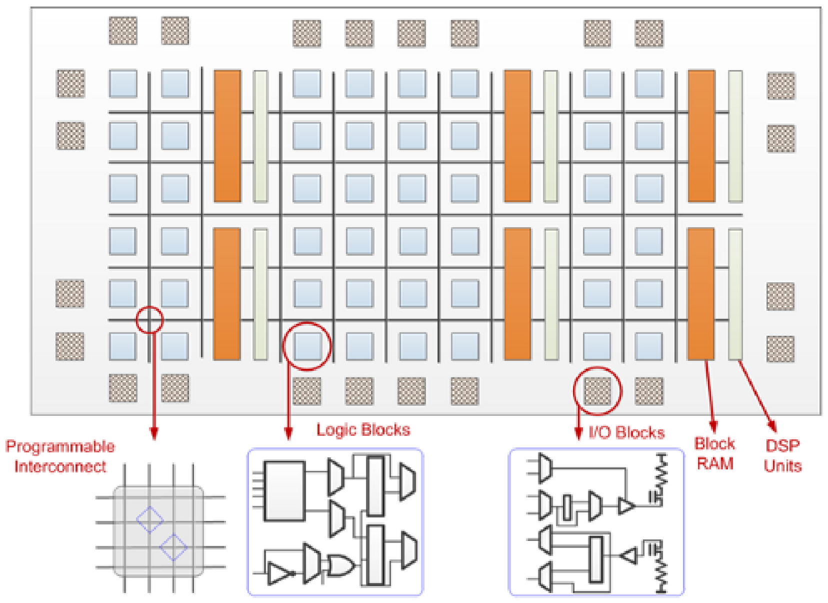


Figure 2.1: Diagram of the basic FPGA structure and internal components.[3]

Many FPGAs can be reprogrammed to implement different logic functions [5], allowing flexible reconfigurable computing similar to the one performed in computer software. FPGAs have a remarkable role in the embedded system development due to capability [6] to start system software (SW) development simultaneously with hardware (HW), enable system performance simulations at a very early phase of the development, and allow various system partitioning (SW and HW) trials and iterations before final freezing of the system architecture. Major FPGA manufacturers include Intel, Xilinx, Lattice Semiconductor, Microchip Technology and Microsemi.

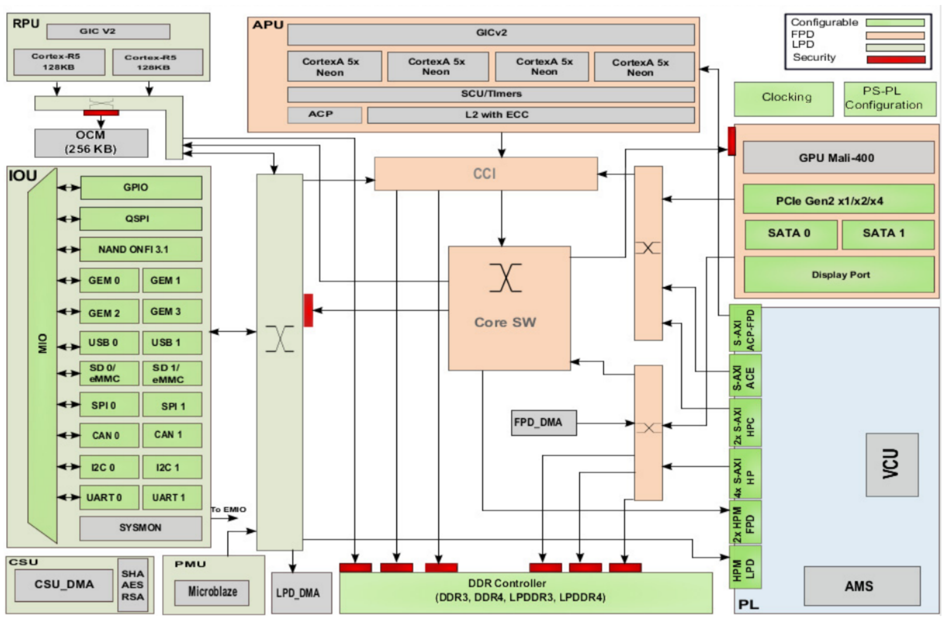


Figure 2.2: Xilinx ZYNQ Ultrascale+ FPGA architecture (Block diagram) [7]

2.1.1. Advantages of FPGAs

An FPGA can be used to solve any problem which is computable. This is trivially proven by the fact that FPGAs can be used to implement a soft microprocessor, such as the Xilinx MicroBlaze or Altera Nios II. Some of their typical advantages are demonstrated in Figure 2.3. Their main advantage, however, lies in that they are significantly faster for some applications because of their parallel nature and optimality in terms of the number of gates used for certain processes. FPGAs have been used to accelerate various types of applications, ranging from image processing [8] to medical applications [9], and can be programmed using high-level languages like OpenCL [10]. However, parallelism can also be extracted by most modern CPUs and is predominantly encountered in GPUs [11–13]. The differences from the aforementioned computing units and the reasons as to why a user would prefer and FPGA solution can be broken down to the following areas [14]:

- **Latency**: The amount of computation time needed for a specific task.
- **Connectivity**: The amount and type of input/output connections along with their respective bandwidths.
- **Energy efficiency**: The amount of energy needed for computation.
- **Engineering cost**: The amount of effort needed in order to express the computation.

Low latency is defined by minimising the time between an input and its response. In this area FPGAs far exceed CPUs in performance, with the former being able to achieve a latency around or below 1 microsecond. Moreover, the latency of an FPGA is much more deterministic due to its high degree of specialisation since they do not depend on a generic operating system nor do they have the need to communicate via generic buses, in sharp contrast to GPUs and CPUs. Furthermore, very high I/O bandwidth can be achieved by directly connecting the data source to the pins of the chip on an FPGA by not having an operating system act as an intermediate. Additionally, FPGAs are much more energy efficient than both GPUs and CPUs when it comes to logic and fixed precision computations, with floating point arithmetic energy efficiency being very close to that of GPU solutions. However, the main disadvantage of FPGAs emerges in the form of programming/configuration difficulty. The engineering cost is typically much higher than for instruction based architectures, so it must be taken into consideration.

FPGAs are also often used to provide a custom solution in situations in which developing an ASIC would be too expensive or time-consuming. An FPGA application can be configured in hours or days instead of months, providing remarkable flexibility at the cost of some speed, area and power. Even when an ASIC will be designed for high-volume production, FPGAs are widely used for system validation, including pre-silicon validation, post-silicon validation and firmware development. This allows manufacturers to verify their design before the chip is pushed to production.

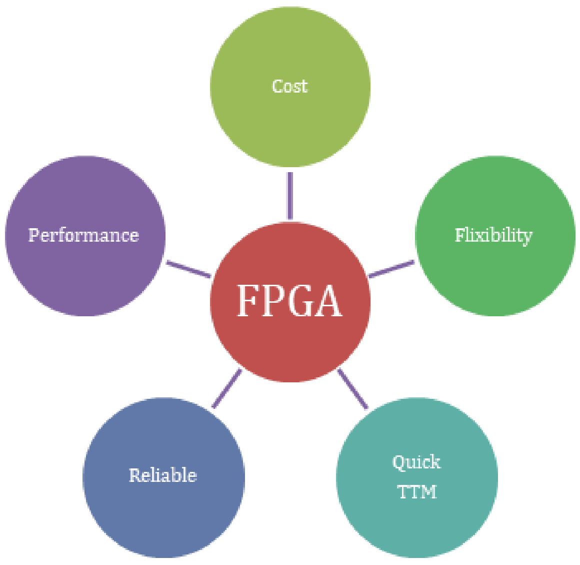


Figure 2.3: FPGA advantages [15]

2.1.2. FPGA applications

FPGAs originally began as competitors to complex programmable logic devices (CPLDs) to implement glue logic for printed circuit boards. As their size, capabilities, and speed increased, FPGAs took over additional functions to the point where some are now marketed as full systems on chips (SoCs). Particularly with the introduction of dedicated multipliers into FPGA architectures in the late 1990s, applications which had traditionally been the sole reserve of digital signal processor hardware (DSPs) began to incorporate FPGAs instead [16, 17].

The ability to configure the hardware of the FPGA, reconfigure it when needed and optimise it for a particular set of functions makes the FPGA an attractive option in a plethora of applications. Many of these application areas are changing rapidly as requirements evolve and new protocols and standards are adopted and FPGAs enable manufacturers to implement systems that can be updated when necessary. As an example, a non-exhaustive list of different markets for which Xilinx provides comprehensive solutions is presented in the list below [4]:

- **Aerospace & Defense** - Radiation-tolerant FPGAs along with intellectual property for image processing, waveform generation, and partial reconfiguration for SDRs.
- **ASIC Prototyping** - ASIC prototyping with FPGAs enables fast and accurate SoC system modelling and verification of embedded software.
- **Audio** - FPGAs enable higher degrees of flexibility, faster time-to-market, and lower overall non-recurring engineering costs (NRE) for a wide range of audio, communications, and multimedia applications.
- **Automotive** - Automotive silicon and IP solutions for gateway and driver assistance systems, comfort, convenience, and in-vehicle infotainment.
- **Broadcast & Pro AV** - Adapting to changing requirements faster and lengthening product life cycles with Broadcast Targeted Design Platforms and solutions for high-end professional broadcast systems.
- **Consumer Electronics** - Cost-effective solutions enabling next generation, full-featured consumer applications, such as converged handsets, digital flat panel displays, information appliances, home networking, and residential set top boxes.
- **Data Center** - Designed for high-bandwidth, low-latency servers, networking, and storage applications to bring higher value into cloud deployments.
- **High Performance Computing and Data Storage** - Solutions for Network Attached Storage (NAS), Storage Area Network (SAN), servers, and storage appliances.
- **Industrial** - FPGAs for Industrial, Scientific and Medical (ISM) enable higher degrees of flexibility, faster time-to-market, and lower overall non-recurring engineering costs (NRE) for a wide range of applications such as industrial imaging and surveillance, industrial automation, and medical imaging equipment.
- **Medical** - For diagnostic, monitoring, and therapy applications, the Virtex FPGA and Spartan® FPGA families can be used to meet a range of processing, display, and I/O interface requirements.
- **Security** - Solutions that meet the evolving needs of security applications, from access control to surveillance and safety systems.
- **Video & Image Processing** - FPGAs enable higher degrees of flexibility, faster time-to-market, and lower overall non-recurring engineering costs (NRE) for a wide range of video and imaging applications.
- **Wired Communications** - End-to-end solutions for the Reprogrammable Networking Linecard Packet Processing, Framer/MAC, serial backplanes, and more.

- **Wireless Communications** - RF, base band, connectivity, transport and networking solutions for wireless equipment, addressing standards such as WCDMA, HSDPA, WiMAX and others.

Another trend in the use of FPGAs is hardware acceleration, where one can use the FPGA to accelerate certain parts of an algorithm and share part of the computation between the FPGA and a generic processor [5]. The current work utilises FPGAs under this context, specifically in the field of video and image processing in streaming applications in medical electronics.

2.1.3. Design and programming

The general design flow centred around FPGAs is showcased in Figure 2.4. A typical FPGA-based design begins by having the developer define the behaviour of the FPGA through the use of a hardware description language (HDL) such as VHDL or Verilog. The process is similar to that of a software development cycle except that the goal is to architect the hardware itself rather than a set of instructions to run on a predefined hardware platform. The RTL description is then put through an iterative process of simulating by creating testbenches that replicate the system behaviour, and making subsequent changes to the code until the intended results are observed.

Once the design has been created and verified using HDL, technology-mapped netlist is generated through a process called "synthesis" from an electronic design automation tool. This netlist is a configuration file containing information on how different components should be interconnected [7]. The netlist is then translated to a gate-level description and configured on the actual FPGA architecture using a process called place-and-route, usually performed through the same automation tool, typically provided by the company. The developer can then assess the routed design, perform time analysis and various other verification and validation methodologies to confirm that the results are without error. Upon completion of the validation process, a binary file (bitmap) is generated by the software which is used for the reconfiguration of the FPGA. Finally, this file is transferred to the device via a serial interface or external memory.

One challenge with an HDL driven approach is that configuring an FPGA requires both coding skills along with detailed knowledge of the underlying hardware, and the required expertise is not widely available. This serves as the primary inhibitor to FPGAs being a mainstream computing platform. As a result, in an attempt to reduce complexity by designing in HDLs, vendors have begun offering software development kits that allow designers to develop FPGA solutions in popular high-level languages such as C/C++, Python and OpenCL. High-level synthesis (HLS) design tools are also available which often feature graphical block diagrams instead of lines of code.

Additionally, to further simplify the design of complex systems in FPGAs, there exist libraries of predefined complex functions and circuits that have been tested and optimised to speed up the design process. These predefined circuits are commonly called intellectual property (IP) cores, and are available from FPGA vendors and third-party IP suppliers. However, they are rarely used in specialised designs because of their inflexible nature and inability to fine-tune the behaviour of the modules.

A significant part of the difficulty of programming FPGAs can also be attributed to the long compilation times due to the place-and-route phase, a complex optimisation problem which requires significant computation, often resulting in overnight compile phases.

2.2. Hardware acceleration

Hardware acceleration refers to the process by which an application offloads certain computing tasks into specially made computer hardware to perform some functions more efficiently than is possible in software running on a general-purpose central processing unit (CPU). Any computation or algorithm can be performed purely in software running on a generic CPU, purely in custom-made hardware (ASIC), or in some combination of the above. An operation can be computed faster, however, in application-specific hardware designed specifically to compute the operation, than it would had it been specified in software and performed on a general-purpose computer processor. The implementation of computing certain tasks in hardware to decrease latency and increase throughput is known as hardware acceleration.

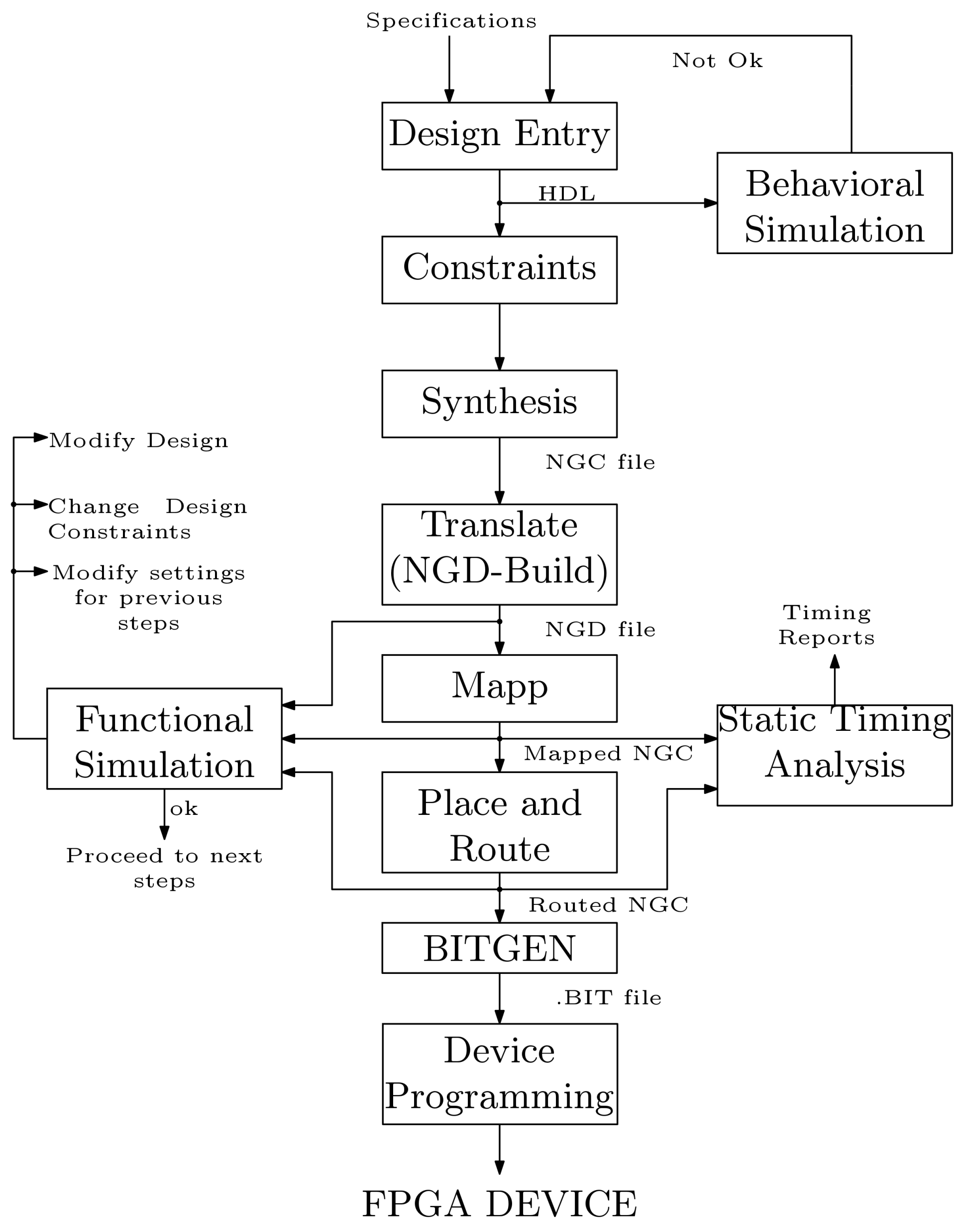


Figure 2.4: FPGA design flow [18]

There exists a perceived hierarchy of digital computing platforms, ranging from GPUs to fully specially designed hardware, there is a significant tradeoff between flexibility and efficiency, as depicted in Figure 2.5. Advantages of software based development typically include a faster time to market, lower non-recurring engineering costs, heightened portability and ease of updating features or patching bugs at the cost of overhead to compute general operations. Advantages of hardware, on the other hand, include speedup, reduced power consumption, lower latency, increased parallelism and bandwidth and better utilisation of area and functional components available on an integrated circuit; at the cost of lower ability to update designs once etched onto silicon and higher costs of functional verification and times to market.

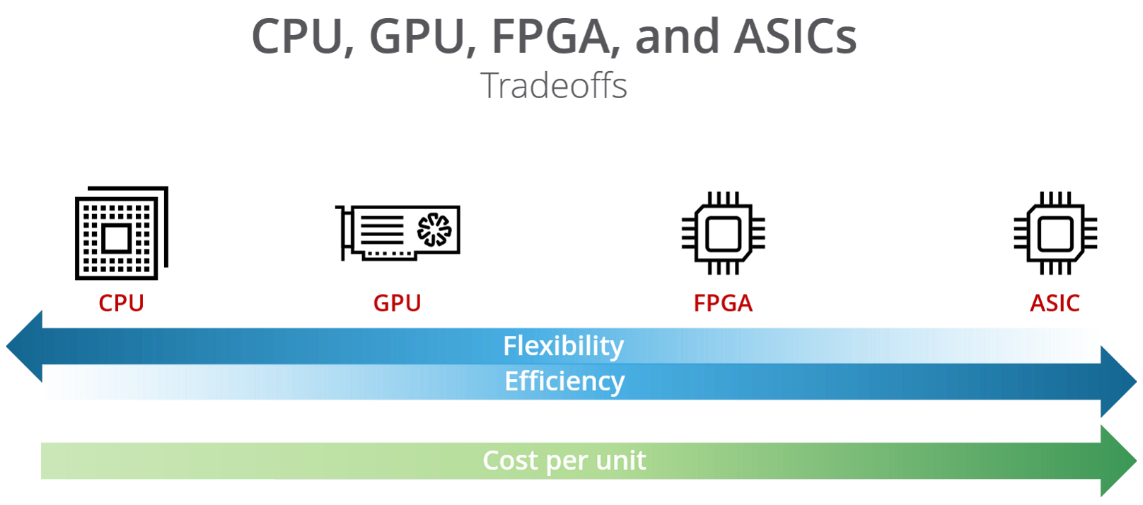


Figure 2.5: Hierarchy of digital computing systems [19]

Hardware acceleration combines the flexibility of general-purpose processors, such as CPUs, with the efficiency of fully customised hardware, such as GPUs and ASICs, increasing efficiency by orders of magnitude when any application is implemented higher up the hierarchy of digital computing systems. For example, visualisation processes may be offloaded onto a graphics card (GPU) in order to enable faster, higher-quality playback of videos and games, while also freeing up the CPU to perform other tasks [20].

Hardware acceleration is also advantageous for performance, and practical when the functions are fixed so updates are not as needed as in software solutions. With the advent of reprogrammable logic devices such as FPGAs, which have been discussed above, and the unprecedented flexibility they offer to hardware implementations, the restrictions on hardware acceleration have eased. This led to more and more fields with inherent opportunities to exploit parallelism adopting the practice. One such field is found in the domain of medical imaging, which has been the focus of the current work. In particular, when X-ray imaging purposed towards surgical interventions is concerned, a tremendous amount of streaming image data need to be processed concurrently. These come attached with severe timing requirements due to the critical nature of the application, rendering it impossible to be processed on time by a general-purpose solution. As such, the field has steered towards hardware acceleration solutions instead, with each application being treated as a unique problem requiring a unique solution.

2.3. Network switch

A network switch is networking hardware device functioning as a multiport bridge that uses packet switching to receive and forward data to the intended destination device. Switches manage the flow of data across a network by transmitting a received network packet only to the one or more devices for which the packet is intended. Each networked device connected to a switch can be identified by its network address, allowing the switch to direct the flow of traffic maximising the security and efficiency of the network, unlike repeater hubs, which broadcast the same data out of each port and let the devices pick out the data addressed to them [21].

Traditionally, a network switch operates by using MAC addresses to identify devices, which places it at the data link layer (layer2) of the Open Systems Interconnection (OSI) model. Some switches can also operate at network layer (layer 3) by additionally incorporating routing functionality, which is necessary for them to support Virtual Local Area Networks (VLANs). Such switches are commonly known as layer-3 switches or multilayer switches [22]. Each device connected to a switch port can transfer data to any of the other ports at any time and the transmissions will not interfere. Switches are a common component of networks based on Ethernet, Fibre Channel, Asynchronous Transfer Mode (ATM), and InfiniBand, among others. In general, however, most switches today use the Ethernet protocol.

To reduce the chance for collisions between network traffic going to and from a switch and a connected device at the same time, most switches offer full-duplex functionality in which packets coming from and going to a device have access to the full bandwidth of the switch connection [23].

2.3.1. General switch structure

In general, a switch can be decomposed into the following parts as shown in Figure 2.6.

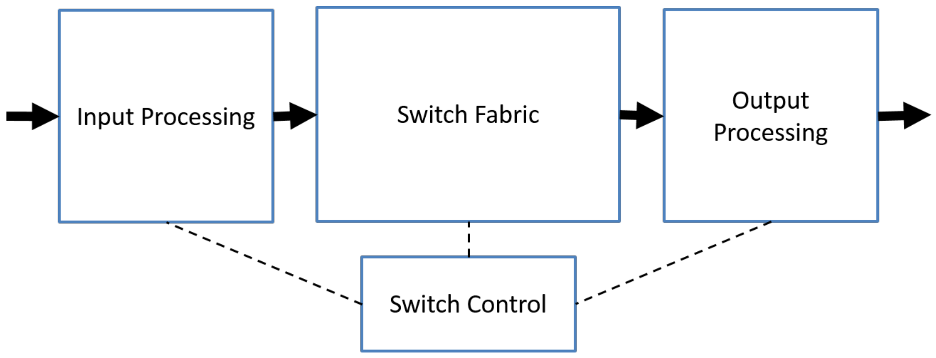


Figure 2.6: General switch architecture [24]

The functionality of the blocks depicted in Figure 2.6 can be briefly described as follows [25][24]:

- **Input processing** - Several functionalities may be performed in the input processing block, mostly centred around the pre-processing of input data. These may include physical and link layer processing, input packet processing, lookup and forwarding, optical/wavelength switching or various forms of conversions (serial to parallel, optical to electrical, etc.).

- **Switch fabric** - The switching fabric is responsible for physically connecting the input ports to the output ports of the switch. It handles for the transportation of packets between the input and output processing sections. Some of the most common techniques used to create the fabric are discussed in Section 2.3.2 below.

- **Output processing**. -Similar to the input processing, a multitude of functionalities might be packed into this block. Namely, those may include queuing, optical amplification, wavelength combining and several conversions. Quite often, it may occur that the output processing performs the reverse functionality of the input processing.

- **Switch control** - The switch processor coordinates the three previous blocks in order to produce the desired switching outcome. This is achieved via execution of routing protocols network management functions.

2.3.2. Switching fabrics

As mentioned previously, the switching fabric is at the physical core of the switch. It is through this that the data can actually be forwarded from an input port to an output port. Switching can be accomplished in a number of ways, as indicated in Figure 2.7 [25].

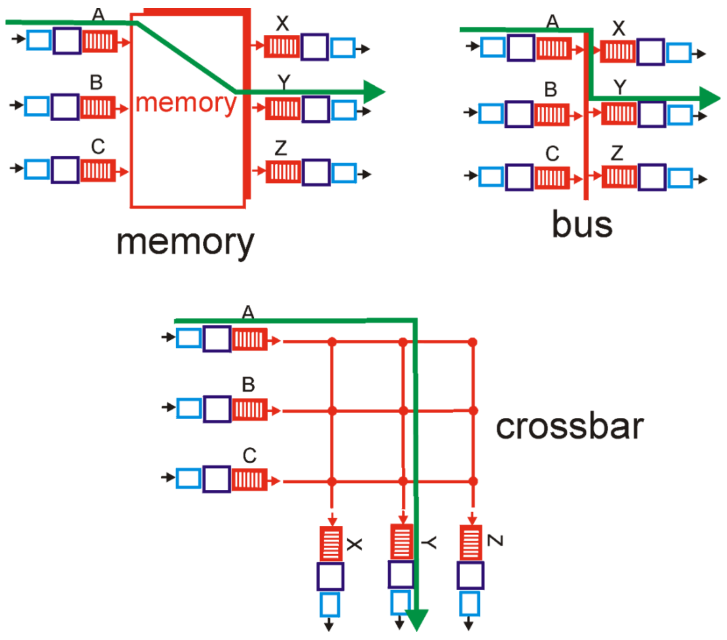


Figure 2.7: Different switching techniques [25]

In the case of switching via memory, the input and output ports function similar to traditional I/O ports in an operating system. The packet is first stored into the processor memory before the destination data is extracted and matched through a lookup table. Subsequently, the data are copied into the buffer of the appropriate output port. This technique was often used by some of the earliest switches and routers, but is still being utilised by many modern devices.

In the bus approach, the input ports transfer data directly to the output ports via a shared bus, without intervention by the routing processor [25]. The downside is that only a single packet can be transferred at a time, with inputs finding the bus busy having to block until the end of transfer. The switching bandwidth in this case is of course limited by the transfer speed of the bus.

One way to overcome the bandwidth limitation of a single, shared bus is to use a more sophisticated interconnection network. The simplest concept of such a switch fabric is that known as a crossbar and is shown schematically in Figure 2.8. It consists of an assembly of individual switches arranged in a matrix configuration and is capable of connecting N input ports to N output ports. An input packet traverses along the horizontal axis until it intersects with an enabled cross point element, called a "cross connect". If the vertical axis pointing to the output is free, the transfer is successfully completed. In case that the channel is busy, the arriving packet is blocked and must be buffered at the input. However, other concurrent connections do not prevent connecting other inputs to other outputs. As a crossbar is one of the fundamental switching solutions and answers a lot of related problems fairly efficiently, it was the first architecture to be put under consideration to be the basis of the current work.

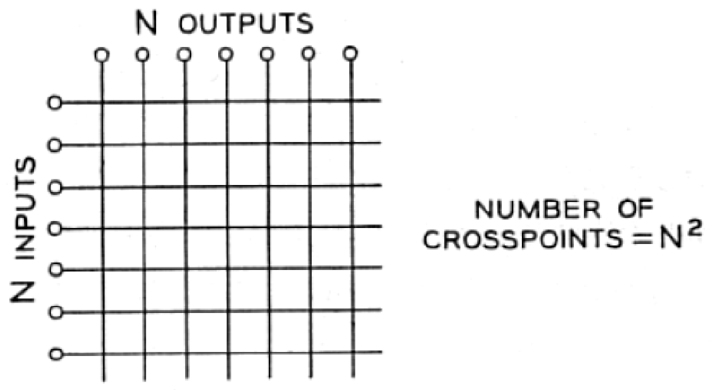


Figure 2.8: The crossbar switching fabric [26]

3 Alternative solutions

Before arriving at the implementation of the Switch module that was ultimately used at the project, research was performed on the possible alternative solutions that could have been used instead. In this chapter, the aforementioned solutions are presented, detailed and compared. A reasoning is provided for each individual solutions validity and efficiency in solving the current problem. The type of solutions investigated include off-the-shelf commercial network switches, different approaches to hardware acceleration and a primitive custom switch design that was initially considered. There are advantages and disadvantages tethered to each solution which are examined in contrast at the end of the chapter.

3.1. Commercial solutions

The obvious first investigative direction for a possible solution to the problem defined in Chapter 1 was, of course, the available hardware devices on the market. 10 Gigabit network switches are quite popular since modern business and home networking applications have become increasingly digital, reflecting in a wide range of products available in the market. However, for the needs of this particular problem, an additional set of non-typical constraints needs to be satisfied. Namely, as stated in Chapter 1, a form of multicasting is needed since a single stream can be replicated up to 4 times on screen at 4 different scaling factors. This narrows down the search to Layer 3 switches with Internet Group Management Protocol (IGMP) and packet forwarding support, as they can handle multicasting much more efficiently than their Layer 2 counterparts. A Layer 2 switch sends the multicast packets to every device and, If there are many sources, the network will slow down because of all the traffic. And, without IGMPv2 or IGMPv3 snooping support, the switch can handle only a few devices sending multicasting packets. A comparison of the characteristics of 4 different such devices is presented in Table 3.1 below.

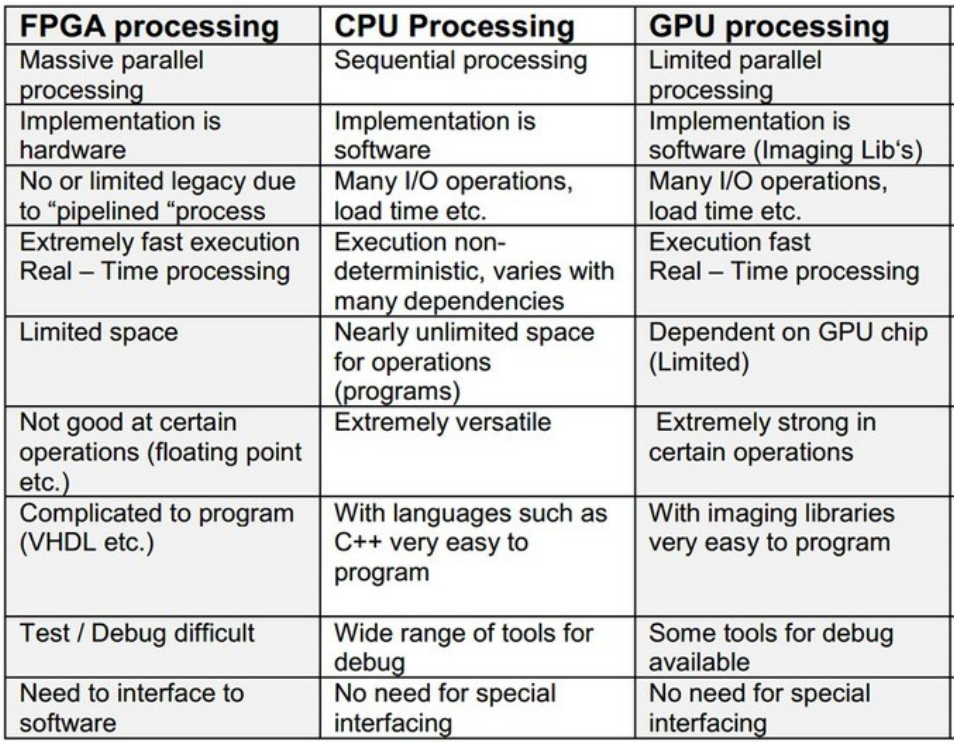
Model	Alternative A	Alternative B	Alternative C	Alternative D
Ports	48 SFP+ (10Gb) and 4 QSFP+ (40Gb) ports	48 SFP+ (10Gb) and 6QSFP+ (40Gb) ports	44 10 Gigabit and 4 SFP+ (10Gb) ports	48 10 Gigabit and 4 40 GbE uplink
Switch Class	3	3	3	3
Switching Capacity	1.28Tbps	1.44Tbps	-	1.28Tbps
Latency	-	612ns	-	-
Forwarding Rate	909Mpps	1071.43Mpps	714.2Mpps	952.32Mpps
Price (US dollars)	7190.10	3699.00	3426.35	3800.00

Table 3.1: Comparison of commercial network switches

The vendors of the devices have been left undefined to avoid undesired marketing. As can be observed, a lot of the technical details such as the switching capacity and latency are often left unclear, making the choice between them even more difficult. More importantly though, the price of a device with such switching capabilities is exceedingly high, comparable to that of the entirety of the rest of the hardware used in this project. Additionally, these switches provide powerful capabilities that are unneeded in this project which would have resulted in severe underutilisation of the product. Furthermore, while effective, most of the proposed products could not handle the imposed task without fine-tuned memory management and additional buffers, which was one of the main issues to be avoided. Thus, even with such an off-the-shelf solution, there would be a need to design additional hardware in order to handle the traffic. Finally, such a solution is neither scalable nor resource optimised in any significant way, rendering it the least feasible out of the considered options.

3.2. GPUs vs FPGA vs ASIC

As a commercial answer was ruled out from consideration, the focus shifted towards a custom, resource optimised, scalable and targeted solution. Since a general-purpose CPU would automatically be inadequate for the current issue due to the sheer size of the data and the bottleneck that would inevitably be imposed by the need to communicate through memory buses, research aimed at hardware acceleration instead. The possible candidates for the acceleration to be designed around were an FPGA board, an ASIC and a GPU. The ASIC option was quickly discarded due to a variety of reasons. While an ASIC is faster than an FPGA alternative, that gap has been narrowing down significantly lately, with the latter option being more than enough to handle the computational needs of the current problem. Another advantage that ASICs offer is the ability to mix analog and digital design to a certain extend, which also offers no benefit to the desired application in the current context. As such, for the purpose of this project, an ASIC can be accurately compared to a "frozen FPGA", with the second being a clearly superior alternative. Figure 3.1 contrasts the remaining alternatives' features to each other and to the general-purpose solution.



FPGA processing	CPU Processing	GPU processing
Massive parallel processing	Sequential processing	Limited parallel processing
Implementation is hardware	Implementation is software	Implementation is software (Imaging Lib's)
No or limited legacy due to "pipelined "process	Many I/O operations, load time etc.	Many I/O operations, load time etc.
Extremely fast execution Real – Time processing	Execution non-deterministic, varies with many dependencies	Execution fast Real – Time processing
Limited space	Nearly unlimited space for operations (programs)	Dependent on GPU chip (Limited)
Not good at certain operations (floating point etc.)	Extremely versatile	Extremely strong in certain operations
Complicated to program (VHDL etc.)	With languages such as C++ very easy to program	With imaging libraries very easy to program
Test / Debug difficult	Wide range of tools for debug	Some tools for debug available
Need to interface to software	No need for special interfacing	No need for special interfacing

Figure 3.1: Comparison of hardware acceleration alternatives [27]

FPGA and GPU platforms can be effectively compared on the following points [28][29][30][31][32]:

- **Raw Compute power:** Except for some limited applications which are tailor-made for extracting severe parallelism from a GPU, the two devices are very close when it comes to compute power. When it comes to on-chip memory, which is essential to reduce the latency, FPGAs result in significantly higher computing capability. The high amount of on-chip cache memory reduces the memory bottlenecks associated with external memory access as well as the power and costs of a high memory bandwidth solution. In addition, the flexibility of FPGAs in supporting the full range of data types precisions is one of the strong arguments towards the platform.

- **Efficiency and Power:** FPGAs have been well-established as power efficient devices. In contrast, the main reason for GPUs being power-hungry is that they require additional complexity around their compute resources to facilitate software programmability. This leads to FPGAs performing nearly four times better in general purpose compute efficiency in applications other than deep learning.

- **Flexibility:** Data flow in GPUs is defined by software and is directed by the GPU's complex memory hierarchy (as is the case with CPUs). The latency and power associated with memory access and memory conflicts increase rapidly as data travels through the memory hierarchy. In the case of receiving a large number of GPU threads, only parts of a workload can be mapped efficiently into the vastly parallel architecture, and if enough parallelism cannot be found within the threads, this results in lower performance efficiency. FPGAs can deliver more flexible architectures, which are a mix of hardware programmable resources, Digital Signal Processing (DSP) and Block Random Access Memory (BRAM) or Ultra Random Acess Memory (URAM) blocks. The user can address all the needs of a desired workload by the resources provided by FPGAs. This unique re-configurability means the user is free from certain restrictions, like Single Instruction Multiple Threads (SIMT) or a fixed datapath, yet massively parallel computations are possible. Another important feature of FPGAs, and one that makes them even more flexible, is the any-to-any I/O connection. This enables FPGAs to connect to any device, network, or storage devices without the need for a host CPU.

From a system design perspective, it is possible to apply either a GPU or an FPGA to the current task. That said, an FPGA can be configured to directly access hardware I/O without the latency of internal bus structures common to general-purpose processors as described in Section 2.1.1, providing a significant advantage over GPUs. The processing functionality of FPGA can be customised at logic port level, while the GPU can only be programmed using its static vectored instruction set. This difference makes FPGAs more responsive and allows more dedicated special-purpose innovation [33]. Furthermore, since a GPU is typically connected as a co-processor of a CPU, its connectivity to elsewhere in the system is quite restricted, limiting the design potential for the desired switching mechanism. As such, bandwidth very quickly becomes an issue that has to be dealt with, both concerning the GPU's connectivity as well as its internal memory. Implementing and utilising buffers in a GPU proves to be far more difficult and inefficient than in an FPGA platform. Finally, a GPU that can handle the application load turns out to be highly expensive, categorising it as an ineffective solution under the current context when compared to an FPGA.

3.3. The crossbar switch paradigm

With an FPGA device being the target platform of choice, an initial design direction of the switching fabric in the form of an enhanced crossbar switch was investigated. Figure 3.2 showcases the resulting architecture of the proposed switch. The idea behind this execution was building upon the implementation of a well-established and extensively documented hardware that connects N-to-N ports, which aligns with the end goal of the current project. However, as previously mentioned, a form of multicast needs to be supported with a single input mapping to multiple destinations, or even the same destination multiple times. This significantly complicated the design of the crossbar since such a functionality required buffering information not only at the input, but on every single output as well. Additionally, the destination downscalers run at a different frequency (300 MHz) than the origin Ethernet ports (156.25 MHz) with no relation between the two driving clocks. Thus, a frequency domain crossing has to occur at either buffering point, perplexing the design of the resulting FIFOs even further.

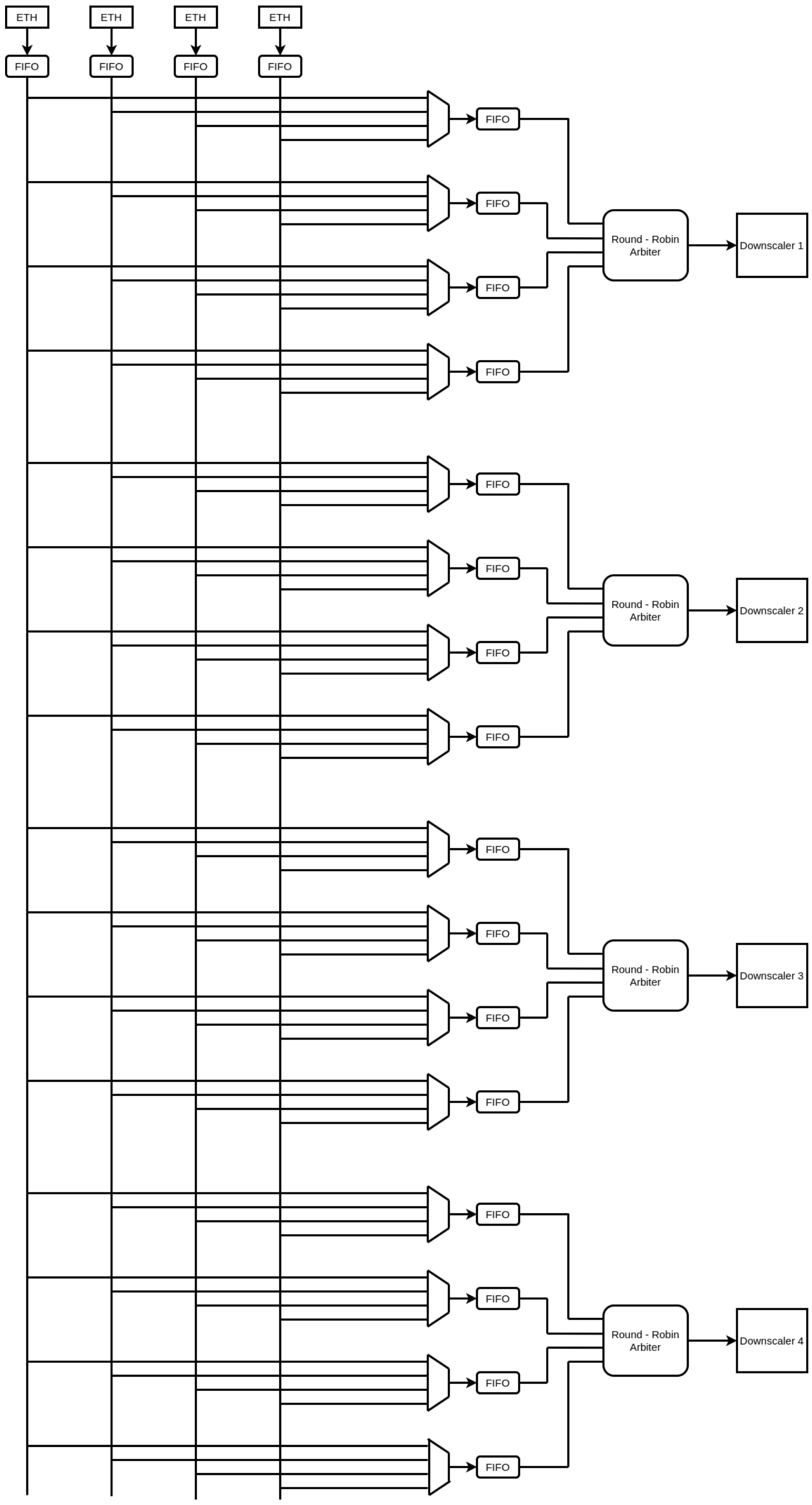


Figure 3.2: Proposed architecture of the crossbar switch paradigm

It is immediately apparent that this approach results in a bloated, difficult and large design. An example of how a random configuration would propagate through this architecture is presented in Figure 3.3. The way this design functions is the following. A packet originating from an Ethernet core is initially buffered into the input FIFO in order to account for the clock difference and absorb potential errors such as missing packets and network jitter. Consequently, as the packet traverses through the vertical direction, it is mapped 4 times to every possible output, gated behind a multiplexer that controls which of the available inputs will be propagated onward. Since every downscaler has 4 unique input ports, this configuration allows for any possible mapping of input streams to downscalers. This includes cases such as a single stream being mapped 4 times to a single scaler at different factors or being mapped once to every available scaler. After the multiplexers have selected the appropriate packet according to the configuration, the data are buffered again in order to account for the downscaler being busy as well as multiple packets arriving concurrently. Finally, an arbiter regulates access to the available downscalers since they are a shared resource.

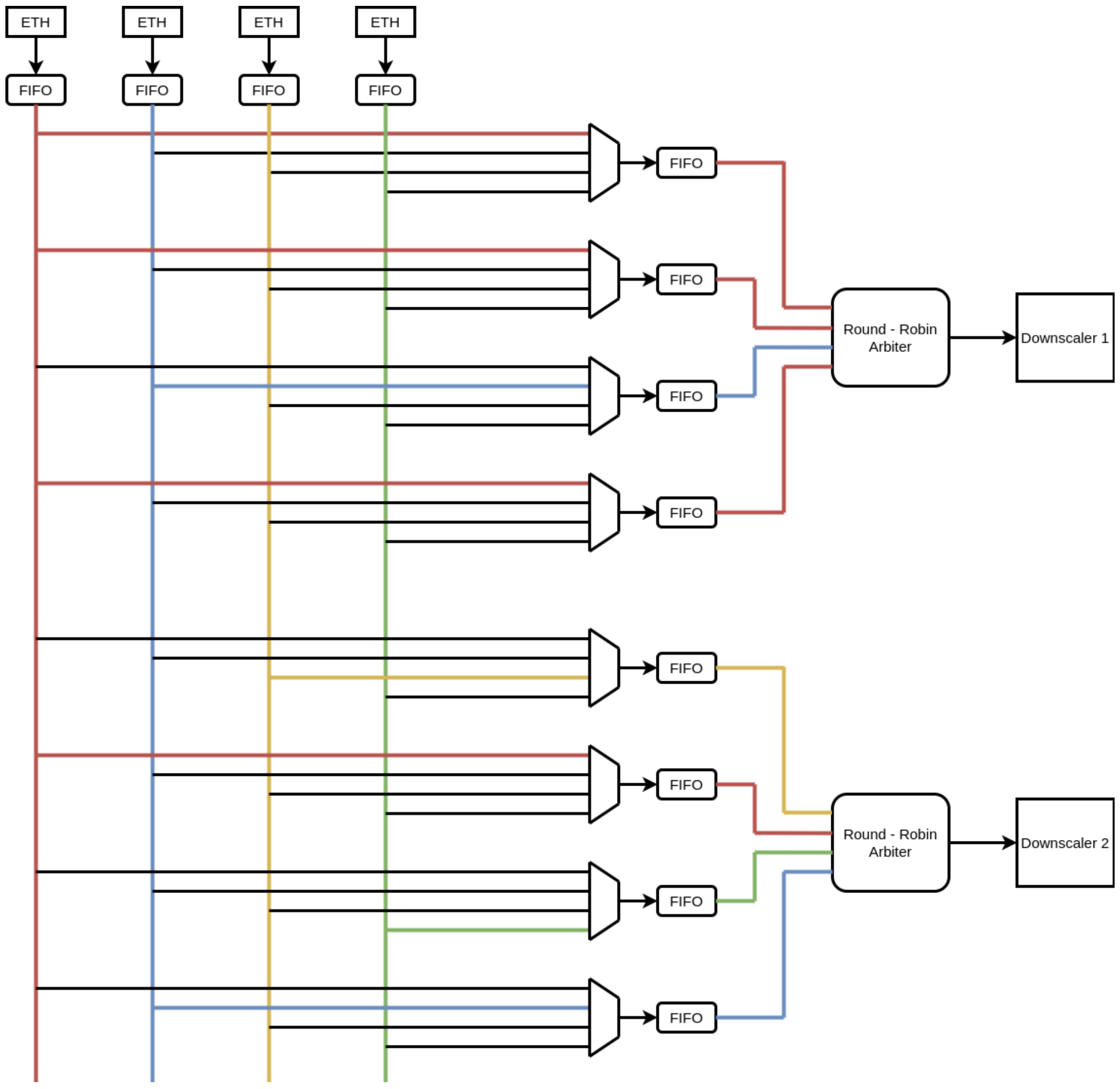


Figure 3.3: Example of a possible configuration

While such a design would indeed possibly work as intended, it quickly becomes very hard to prove that it will be functionally correct under every and all circumstances. It takes significant effort to simulate its behaviour for both functional and optimisation purposes, and is similarly difficult to investigate every possible resulting corner case that might occur. Additionally, one of the most considerable downsides of this architecture is that every one of the FIFOs depicted in Figure 3.2 need to account for the worst-case scenario in terms missing packets, jitter or both. This immediately leads to explosively sizeable buffers with very low overall utilisation except for under unlikely and extreme scenarios, making this feel more like a "brute force" solution. These issues led the design direction to shy away from this solution and focus attention towards a more elegant, single-streamed architecture.

It should, however, be noted here that this by no means an optimised form of this design and simply serves to showcase the potential downsides of basing the Switch architecture around a crossbar. It is entirely possible that the design can be tweaked enough to prove functional. For example, there is a possibility that the resulting hardware could be folded down to 1/4 on each downscaler output if the input packets are encoded with a "destination vector", i.e. `1101` for the red line on the first scaler in the example shown in Figure 3.3, and multiplexed in a single output FIFO. While ideas around this were discussed, such a route was never fully explored.

3.4. Comparison

A summary of the solutions proposed in this chapter is presented in Table 3.2 below.

Solution	Advantages	Disadvantages
Commercial Switches	Relatively easy to incorporate and design around	Unnecessary features Very expensive No scalability No resource optimisation
GPU	Massive parallel processing Easy to programme and design around	Expensive Memory issues Dependent on CPU No direct I/O connection
Crossbar Switch	Simplistic design, easy to implement A plethora of available literature	Brute force solution Large and difficult design Memory management issues Unnecessary resource usage Difficult to scale
FPGA	Easy to integrate with the rest of the compositor Fast, supporting high parallel processing Direct I/O connections Resource optimised Inexpensive solution	Difficult development flow Requires extensive knowledge of both HW and SW

Table 3.2: Comparison of the proposed alternative solutions

It is easy to discern that for the current application, an FPGA-based solution is incredibly compatible. The main focal points of the FitOptiVis project are speed and resource optimisation, as mentioned previously, both of which are ideally addressed through an FPGA platform. Furthermore, the rest of the compositors design is based around a Xiling ZYNQ UltraScale+ device, which can fully accommodate for the various I/O ports needed for the switch module, while providing a homogeneous development environment for the entire compositor, compensating for the difficulty in the design process. The device operates at 300 MHz, a frequency high enough to handle the image streaming bandwidth of the Ethernet fibres with little to no latency. Finally, it is also the most inexpensive of the proposed alternatives, crossing another major target of the project. Thus, it was decided that the Switch will be developed as an integrated part of the rest of the design on the ZYNQ UltraScale+, xczu7ev-ffvc1156-2-e device.

4 Implementation

This chapter describes and thoroughly details the design and implementation decisions that led to the final version of the Switch. Due to confidentiality reasons, some of the finer details that affect intellectual property of Philips or other collaborating companies might be omitted or abstracted away from. The sequence of the current chapter accurately reflects the actual design flow that was followed during the development of the module. Initially, an ambitious architecture was proposed and examined for its validity. Said architecture was then simulated in order to prove that it would exhibit functionally correct behaviour under all circumstances and to explore possible optimisations. Consequently, the architecture was broken down to its individual components, each of which was examined, elaborated on and implemented separately in order to achieve its intended functionality. Finally, the irregular situations were investigated and handled independently and some optimisations were performed on the original design.

4.1. Proposed hardware architecture

Figure 4.1 depicts the initial approach of the final design that was followed. The showcased approach aims to utilise four-fold time division multiplexing to support 4 Ethernet fibres and up to 4 streams per downscaler using only a single source string. It is immediately apparent that this design leads to a much simpler and more streamlined architecture than the one examined in Figure 3.2.

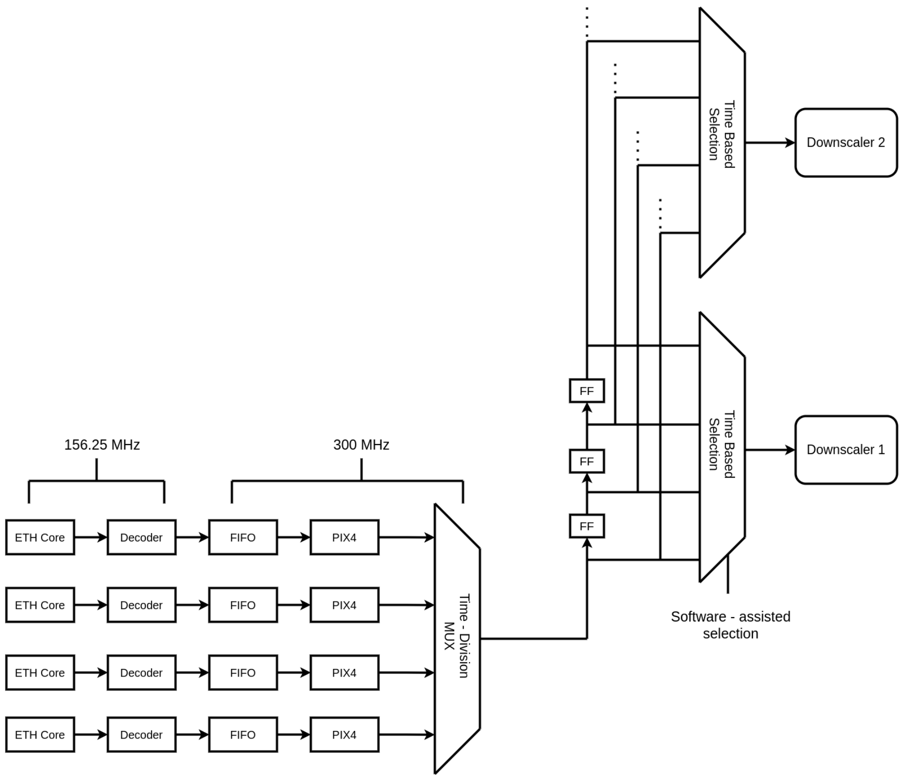


Figure 4.1: Initial proposed architectural approach for the Switch

The various elements of the depicted switching architecture work as follows:

- **ETH core** - The core receives 64-bit wide message data at 156.25 MHz from the Ethernet fibre and relays them forward. The received data follow the Gigabit Jumbo Frame, which is depicted in Figure 4.2, with the addition of VLAN support. The payload of the Jumbo frame, in turn, follows a certain streaming Ethernet protocol and comes attached with its associated headers and information packets. Finally, the ETH core appends a cyclic redundancy check (CRC) at the tail of each forwarded packet which can be asserted for error correction purposes.

- **Decoder** - The "Decoder" block performs stream decoding by stripping unnecessary information from the stream. Once the unfiltered payload has been extracted, the block injects information about the identity and state of the packet. Additionally, info about the integrity of the payload are encoded alongside it and some corrective measures are taken for specific cases. Furthermore, the block includes synchronisation information that is used by the rest of the system further down the line. During this stage, the original pixel data is retained at 64 bits.

- **FIFO** - The "FIFO" block is comprised from two cascaded buffers, a sizeable synchronous one and a smaller asynchronous one. The former one is responsible for absorbing the incoming traffic, including irregularities such as missing packets and network jitter, while the latter is responsible for the frequency domain crossing between the Ethernet core and the FPGA device.

- **PIX4** - The "PIX4" block regroups triplets of 64-bit input samples into 4 pixel bundles, for a total of 96 bits per group and forwards them to the multiplexer. The block is additionally responsible for reconstructing missing and broken packets under the directive of the decoder, if such a case is detected.

- **Time-Division MUX** - The time-division MUX multiplexes the pixel groups provided by the PIX4 blocks in a uniform manner, providing an effective data frequency of 75 MHz per core. The 96 bit pixel bundle along with the accompanying synchronisation data is then forwarded to a 4-stage delay line.

- **Time based selector** - The selector is responsible for selecting and propagating the appropriate bundle from the delay line according to each downscaler's assigned stream through the use of a unique stream-core identifier and software assistance.

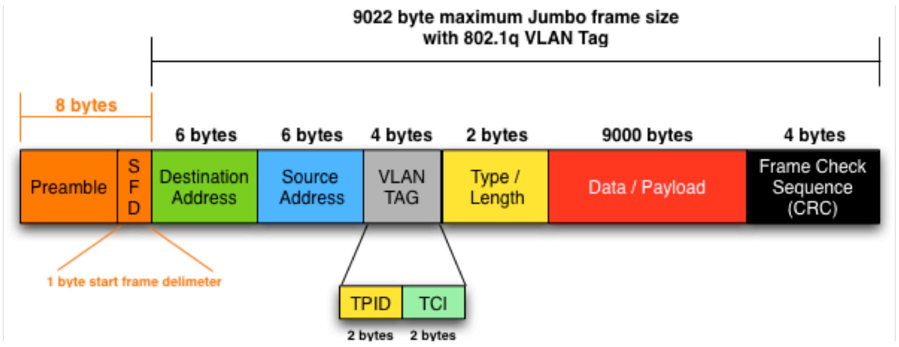


Figure 4.2: Breakdown of the Gigabit Ethernet Jumbo frame [34]

The intention behind this arrangement is to create a single unified dataflow from the sources to the scalers. This merging of streams is achieved through time multiplexing the sources in a single wire, which also has the adverse side effect of limiting the effective data frequency to a fourth of the FPGA clock, as previously mentioned. In order to offset that effect and preserve the bandwidth, the data are bundled in groups of 4 pixels, leading to a steady flow of 4 pixels every 4 clocks per Ethernet core. The software assisting the time-based selector is then responsible for cherry picking through the multiplexed streams and forwarding the ones intended for each downscaler. For any given configuration, the combination of the source Ethernet core and 2-bit stream identifier is enough to uniquely describe any possible stream so that the software can pick them out. The role of the delay line is to integrate the custom multicasting functionality by providing each downscaler with 4 opportunities to fetch each packet. This way, the packets can even be fetched multiple times by one or different downscalers in order to replicate a given stream into the screen multiple times. Incidentally, if such a configuration is not needed, each scaler can grab its intended stream immediately, i.e. from the initial step of the delay line. Every single one of the aforementioned chain of components, shown in Figure 4.3, and their functionalities as well as how they behave in relation to each other will be examined in greater detail in the following sections.

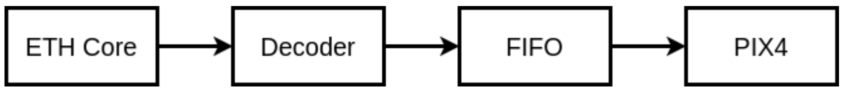


Figure 4.3: Functional components of the Switch's processing pipeline

An example of the described behaviour is illustrated in Figure 4.4. For comparison purposes, the depicted configuration is the same as the one previously shown in the crossbar variant example in Figure 3.3. It should be noted that both of those examples have been simplified for demonstration purposes by assuming a single stream per core, marked by a different colour. In a real scenario this logic is extended to accommodate up to 4 streams per core, with the depicted functionality, however, not changing drastically. As can be observed, the first scaler utilises the delay line to fetch the red stream during three different instances in time, whereas the second one always picks and forwards the first available stream, since it needs to process each of them exactly once. Furthermore, in stark contrast to the crossbar variant, the proposed Switch architecture does not need both input and output buffering, instead only utilising a single FIFO per core. Additionally, it is immediately apparent that four clock cycles are necessary to achieve the same result as the crossbar paradigm. However, due to the bundling of the transmitted data, the transfer rate remains the same between the two. Moreover and very importantly, this design does not exhibit the same explosive scalability that is present in the crossbar counterpart. Instead, multiple source strings can be simply placed in parallel to support more than 4 fibres. In that case, the multiplexer at the input of the scaler needs to become wider to be able to select between the available chains.

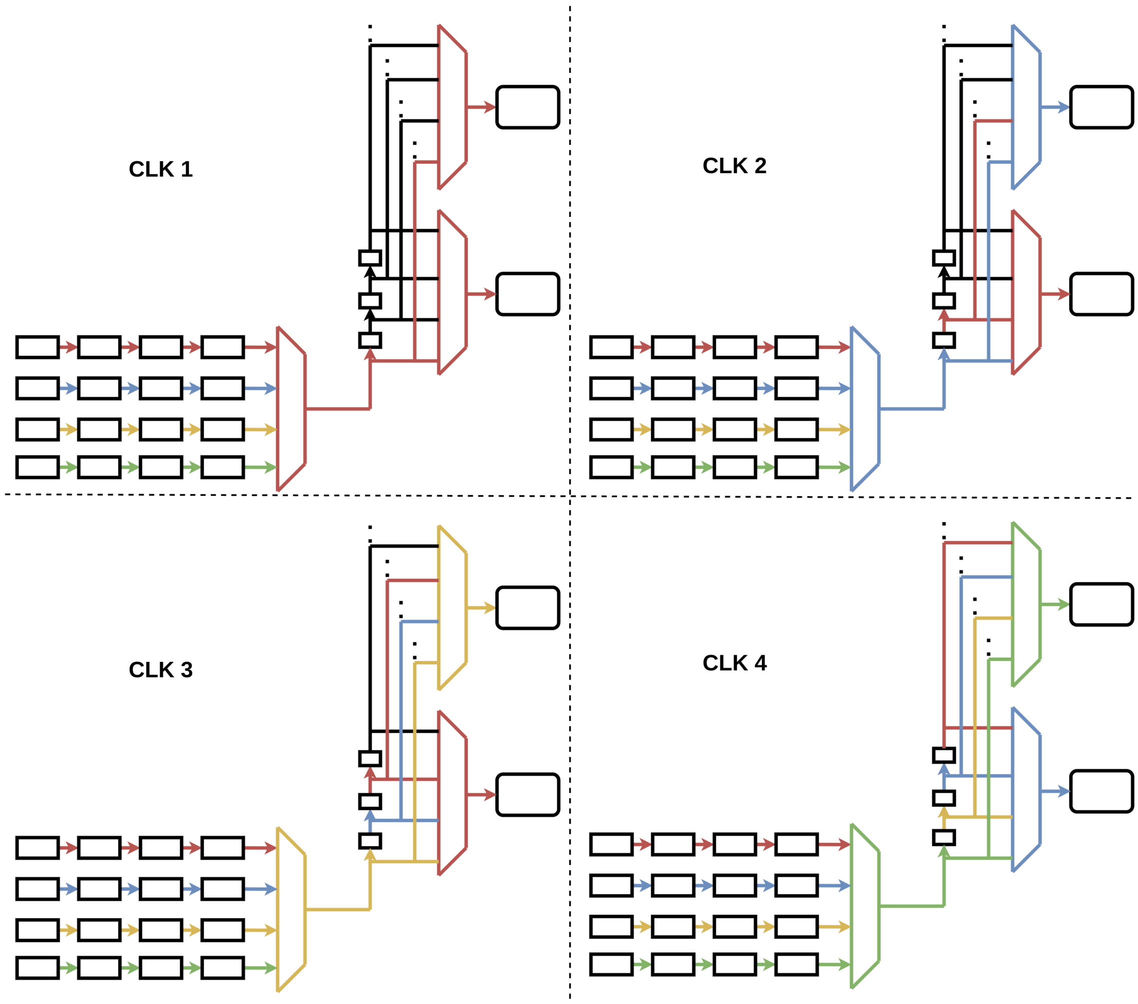


Figure 4.4: Example of a possible configuration of the Switch, unfolded over four clock cycles

4.2. The simulation

Before proceeding with the actual hardware implementation, a simulation of the proposed architecture was built on a higher abstraction level. Its purpose was to, initially, act as a proof of concept for the design by making sure that no deadlocks occurred and that the dataflow would work as theorised. Any unpredicted or undesirable behaviour observed through the simulation could then be rectified in a revised iteration of the model which would then be, subsequently, assessed again. Furthermore, the same simulation model was later enhanced and repurposed to assist with the detection of corner cases and resource optimisations concerning the final version of the hardware. This simulation builds upon tools and concepts that were used in the Almarvi project, a predecessor to FitOptiVis.

4.2.1. The Synchronous Data Flow model

The aforementioned simulation was fabricated using a Synchronous Data Flow (SDF) model on a custom Java hardware simulator (courtesy of Rob de Jong). Synchronous Data Flow models are a restriction of Kahn process networks. In turn, Kahn process networks (KPNs) is a distributed model of computation where a group of deterministic sequential processes are communicating through unbounded FIFO channels. The resulting process network exhibits deterministic behaviour that does not depend on the various computation or communication delays [35]. SDFs differ only in the fact that nodes produce and consume a fixed number of data items per firing, which allows for static scheduling. A simple example of an SDF network is shown in Figure 4.5. In an SDF, hardware components are abstracted away from their implementation details and are instead replaced by a purely behavioural representation called a "process". Processes read and write atomic data elements, called "tokens", from and to channels. In this particular case, tokens are an imitation of the pixel data being transferred and acted upon through the circuit. Writing to a channel is non-blocking, i.e. it always succeeds and does not stall the process, while reading from a channel is blocking, i.e. a process that reads from an empty channel will stall and can only continue when the channel contains sufficient data items (tokens). Processes are not allowed to test an input channel for the existence of tokens without consuming them because timing or execution order of a processes must not affect the result.

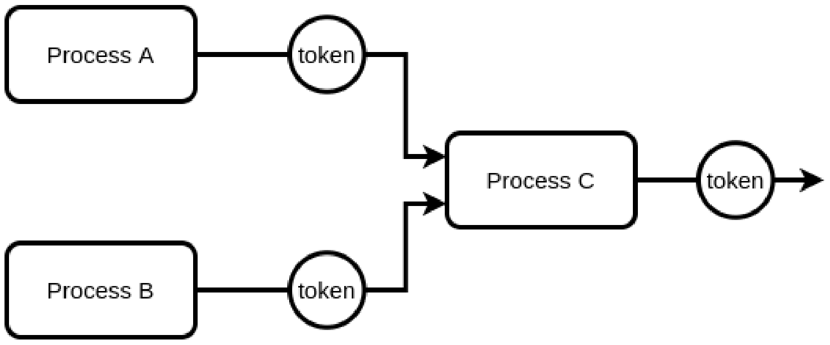


Figure 4.5: Example of an SDF network where a process is consuming two tokens to produce an event

4.2.2. The switch SDF model

The SDF model corresponding to the proposed Switch is depicted in Figure 4.6. In the context of an image data streaming interface, the only operations where process modelling makes sense are those that transform or consume a bundle of data under certain conditions (e.g. filters, scalers, etc.). In other words, operations which function unconditionally and under a set delay as long as there is an input would make for redundant processes in the simulation environment, and would hold no significance to the end result. Following this assumption, the decoder module would fall under this category since its job is to scan data for structural integrity and prevent unnecessary information from being propagated further. Functionally, the decoder could be replaced by a process that replicates its input after a set amount of cycles, or in this case be omitted entirely. Similarly, the time-division multiplexer operates continuously, following a well defined input-output relation under a fixed amount of time and will, as such, receive the same treatment. Consequently, the model of the proposed architecture can be functionally reduced down to the PIX4 block and the accompanying FIFO. Finally, the choice of which type of data the tokens represent would be between bits and pixels. Since a pixel is represented by 24 bits of data, the 64-bit wide interface of the Ethernet core does not correspond to a whole amount of pixels. Therefore, since fractional tokens are not permitted, a bit representation was the only real choice. This could be further be narrowed down to either 64 tokens in a single clock at the input of the PIX4 process, or a single token representing a 64-bit "bundle", with both representations being interchangeable.

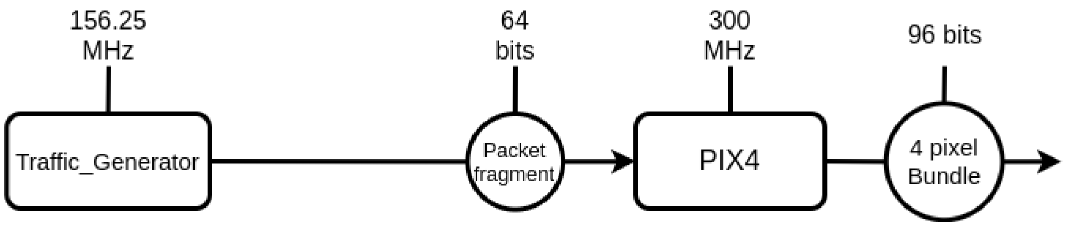


Figure 4.6: SDF functional equivalent of the proposed switching hardware architecture

Due to the nature of the traffic generation though, the standard SDF model alone would prove insufficient for an accurate representation of the data flow of the module. Namely, the complication stems from the fact that up to four unique image sources are time multiplexed within the same channel, which at worst case scenario can fully occupy the bandwidth, leaving no downtime between transmissions.

The actual streams carry identifiers so as to not be mixed by the hardware, but are indistinguishable on the simulation level. For that reason, some additional modifications had to be made to the model and the simulation framework in order to incorporate the changes.

4.2.3. The PSDF compliment

The model was complimented with the ability to add an arbitrary number of parameters or attributes to the tokens. By convention, it was agreed upon that the first parameter would indicate the size of said token, with the rest of the potential parameters' applications left up to the user. In this particular use case, two additional parameters were added to the tokens. Those were a stream identifier, as mentioned above, and an error counter signifying the amount of faulty or missing data in a stream, respectively. Additionally, the ability to increment and decrement token sizes was introduced in order to smooth out the simulation results by avoiding "chunks" of data being processed at once. This enhanced version of the model was dubbed PSDF, standing for Parametrisable Synchronous Data Flow. Figure 4.7 below showcases the described model.

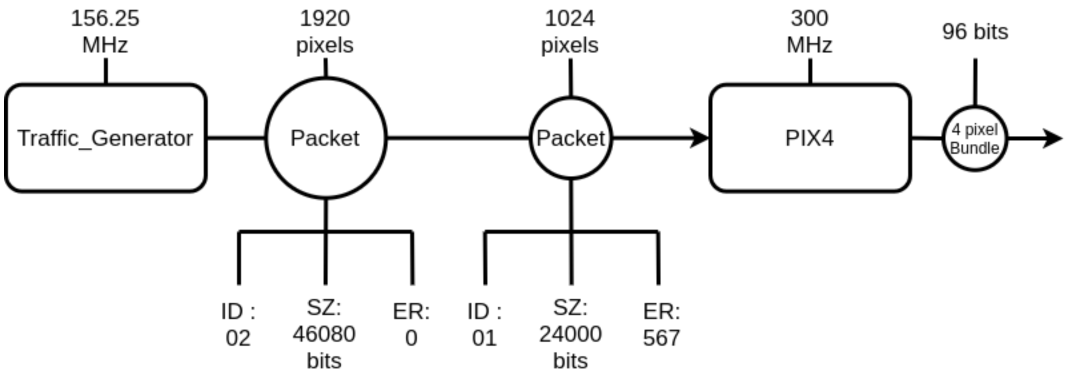


Figure 4.7: PSDF model of the proposed switching hardware architecture with two source streams

Two additional parameters can be configured for any processing node, the `processing_weight` and the `initiation_interval`. The former determines the processing cycles needed for a single input element while the later dictates the minimum amount of time required between the processing of two successive elements. For the PIX4 block, the `processing_weight` has been simply set to 4 while the `initiation_interval` has been left to its default value, which is the minimum time between the start moments of two consecutive tokens. The complete behaviour of the node is, then, described as follows: Every four clock tics, if there exists an available input, the node decrements the input token size by 96 bits (4 pixels) and outputs a token (4 pixel bundle). When the token size is 0, it is consumed and the next one in the FIFO takes its place. This behaviour can be observed in Figure 4.8 below.

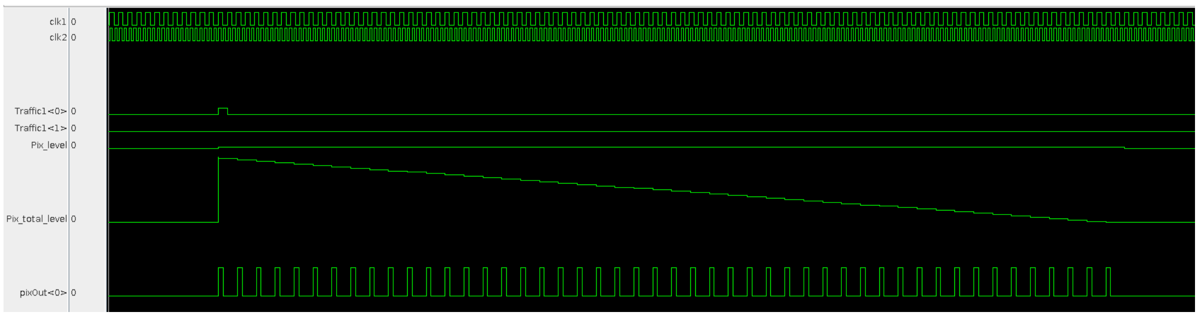


Figure 4.8: Output of the PIX4 process when given a 192 pixel wide input

In Figure 4.8, as well as any of the following figures that display simulator outputs, the fields depicted are analysed as follows:

- **clk1 & clk2** - These signals correspond to the 156.25 MHz driving clock of the Ethernet core and the 300 MHz driving clock of the PIX4 block, respectively.
- **Traffic1<0>** - Marks the first parameter of the token produced by the traffic generator and as per convention denotes its size in bits.
- **Traffic1<1>** - Marks the second parameter of the token and denotes the amount of missing or faulty bits in the packet, i.e. a 192 pixel packet arriving 50% complete would be represented as a token of 2304 bits in size and 2304 bits of error.
- **Pix_level** - This field corresponds to the amount of active tokens in the process' input queue (which simulates the hardware FIFO).
- **Pix_total_level** - Denotes the summation of the sizes of the aforementioned active tokens, and is the field which coincides with the size of the hardware FIFO as well as the primary target field of this analysis.
- **pixOut** - Designates the output of the process, which as described before is a four pixel bundle every four clocks and holds no further significance to the model other than displaying when the process is active.

4.2.4. Simulation results

The first observation from the simulation is that the proposed architecture will function correctly, even when the bandwidth is nearing 100% occupancy, since the input will be on average slower than the output. This acts as a proof of concept for the design. The linear nature of the system also ensures that no deadlocks occur. Additionally, the system exhibits a particularly useful property in the form of it being constrained by a single bottleneck, which further simplifies the analysis of the worst-case scenarios. Those are all originating from burst behaviour on the input of the system and are further elaborated upon in the subsections below. This analysis was performed in order to determine the maximum jitter tolerance of the module along with figuring out the optimal buffer size for the FIFO. It is important to note that the simulation is leaning towards a pessimistic approach. A token, in contrast to an actual packet, arrives as a chunk of input and only then does the processing actually begin. In a real scenario, however, the processing commences as soon as the first 64-bit element of the packet arrives instead, leading to the FIFO not getting filled with an entire packet under normal circumstances. While a more precise simulation is indeed possible, it is not really necessary for a worst-case scenario, with the pessimistic method even providing some extra leeway for error.

Worst case - packets

In order to identify the worst case scenario that can occur, it is important to briefly mention how error handling is managed by the module. A full explanation of the underlying mechanisms is given later on in Section 4.7. Packets are scanned and modified in order to ensure structural integrity. That means that a packet is guaranteed to arrive to the downscalers in the exact form that has been described in its accompanying information package. To achieve that, long packets are cropped to the correct size, short packets are padded and missing packets are reconstructed with placeholder data. Too many missing packets will cause the entire frame to be dropped.

Since the cropping operation is happening on the side of the decoder, the timing behaviour of the system is not affected. Padding, on the other hand, is handled by the PIX4 block under the instructions of the decoder. Those packets as well, however, do not disturb the timing behaviour of the system since padding and propagating a certain amount of data takes the same amount of processing cycles as bundling the same amount of data from a non-faulty packet. In other words, errors do not come, nor are they processed faster than packets. As such, if a packet arrives at all, the processing time needed by the PIX4 block is deterministic and correlates solely to the size information of said packet.

Thus, the only case where a delay can be caused from packet errors is in the event of missing packets. The amount of potentially missing packets will only be detected by the time that the first packet to arrive and have to be reconstructed and propagated before the arrived packet can actually be processed, in order to respect the arrival order. This behaviour is akin to several packets arriving simultaneously due to jitter, which will be discussed in the following section. The processing time in this case is multiplied for each missing packet, with the worst case scenario being the maximum allowed number of missing packets, which at the time of writing is set to 4. Figure 4.9 illustrates the behaviour of the model when a packet arrives accompanied by 4 missing packets from a 1080p stream at 60 fps. Figure 4.10 showcases the system behaviour and recovery time needed through a multitude of such instances.

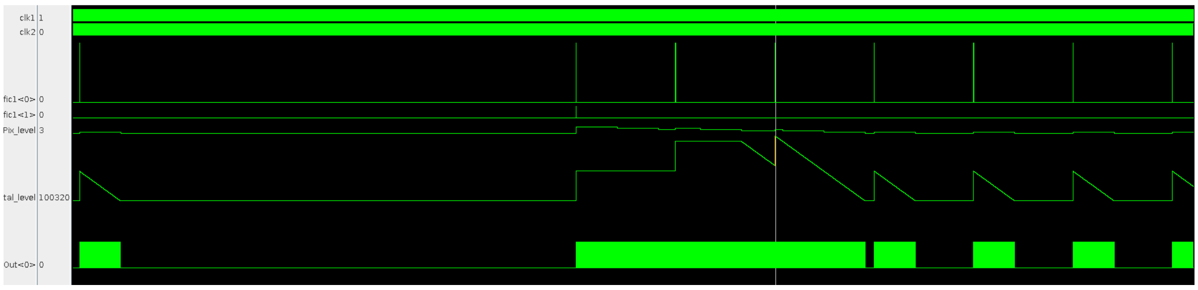


Figure 4.9: Output of the PIX4 process during burst behaviour due to missing packets

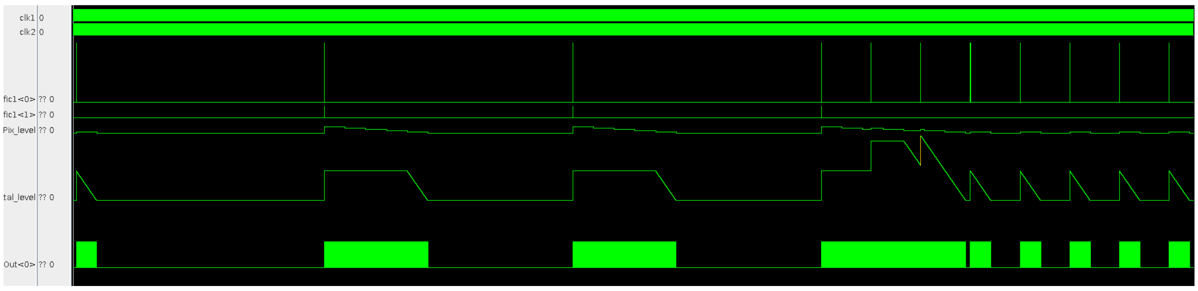


Figure 4.10: Output of the PIX4 process during burst behaviour due to consecutive instances of missing packets

As can be observed, the worst case results from normal traffic continuing to flood the input buffer while the missing packets are still being reconstructed, with consecutive instances of such missing packets on a single stream posing less of an issue as it has enough time to recover during the downtime of activity on the channel. Additionally, the “clipping” that is present on the buffer size corresponds to the period of time when the PIX4 block is busy reconstructing missing packets instead of processing the queued ones, leaving the buffer size stable for the duration unless new data arrive. Furthermore, under real world circumstances most missing packets happen to be consecutive, marking the significance of this part of the simulation. Ultimately, The highest observed buffer size needed for a single 1080p stream through this analysis was 100320 bits, or 12.52 kBytes.

Worst case - jitter

In order for the effect that jitter can have on the system to be analysed, the possible sources of jitter have to be identified and discussed first. The potential causes of jitter that can be singled out by examining the schematic in Figure 4.11. In said figure, the Switch module is once again referred to as “MUX” in order to avoid confusion with the input network’s switch.

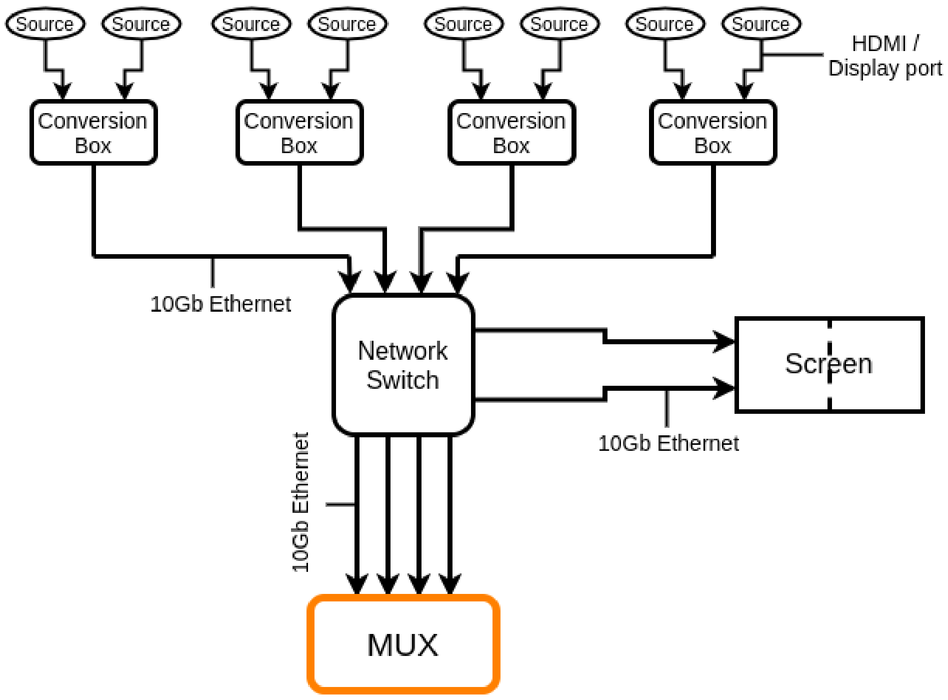


Figure 4.11: Input network topology

The conversion boxes depicted in Figure 4.11 combine two display port or HDMI isosynchronous sources into a single 10Gb Ethernet connection. This conversion box itself can be the first potential source of jitter since the two sources are competing for the same output slot. Small fluctuations in source timings can lead to timing collisions on the output, which in turn lead to two near simultaneous packets arriving at a single input of the network switch. Furthermore, the switch exhibits the same kind of competition on its output ports, with 4 inputs (conversion boxes) having the probability of sending concurrent packets to the same channel, which accounts for the second potential source of jitter in the network. The assumption is made here that the network switch acts in an ideal manner and does not introduce any additional jitter due to memory management, computation or other origins. This is a safe assumption in this particular case due to the actual switch used being a very powerful hardware device.

As per specification, the maximum amount of time multiplexed streams in a single input Ethernet core is 4. Thus, the worst case scenario for jitter comes down to any combination of the above that would lead to 4 concurrent packets arriving from 4 different sources in a single ETH core. Figure 4.12 showcases the behaviour of the system in the unlikely event that four distinct 720p streams send simultaneous packets at 60 fps. Figure 4.13 shows the behaviour of the system under the event that jitter constantly results in bursts of data at the input.

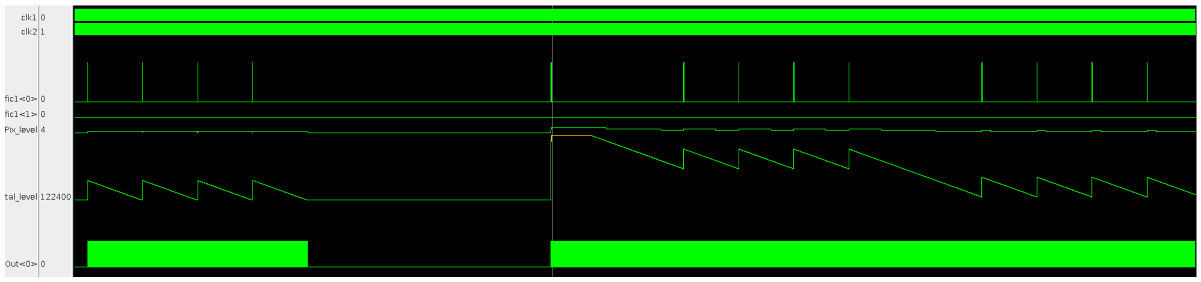


Figure 4.12: Output of the PIX4 process during burst behaviour due to network jitter

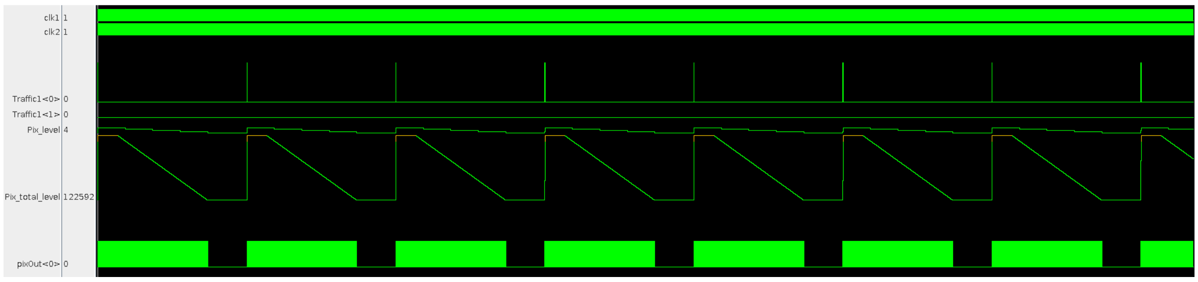


Figure 4.13: Output of the PIX4 process during burst behaviour due to constant network jitter

While both scenarios examined two similar cases consisting of four concurrent packets, some critical differences become immediately apparent in the behaviour of the system. In contrast to the previous scenario, where missing packets were involved, no clipping of buffer can be observed here, having instead been replaced by an almost perfect "sawtooth" behaviour. This behaviour is expected since, unlike with missing packets, the block is dealing with actual tangible data in this case. Once again, while comparing the two figures, it is apparent that the bursts cause more damage when experienced erratically rather than consistently. In conclusion, the highest observed buffer size needed through the current analysis resulted in 122592 bits or 15.32 kBytes, a noticeable increase from the previously examined case.

Worst case - combined

Since the two cases of buffer loading have been sufficiently analysed individually, it is sensible to examine the case of both of them happening at the same time. The following Figures 4.14 and 4.15 display the system behaviour when four concurrent packets arrive at the input, each accompanied by 4 missing packets of their respective stream. While such a scenario is exceedingly unlikely to ever take place, it is however possible in theory and deserves its own examination. The streams in this example, as with before, are 720p at 60 fps as not to violate the bandwidth constraints of the channel.

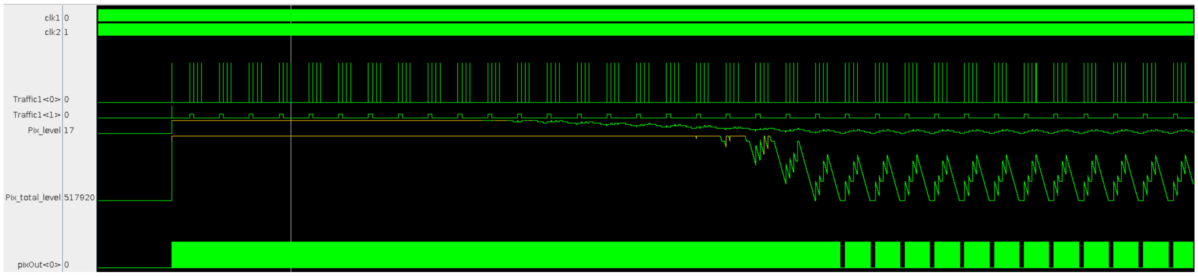


Figure 4.14: Output of the PIX4 process during burst behaviour due to network jitter and missing packets

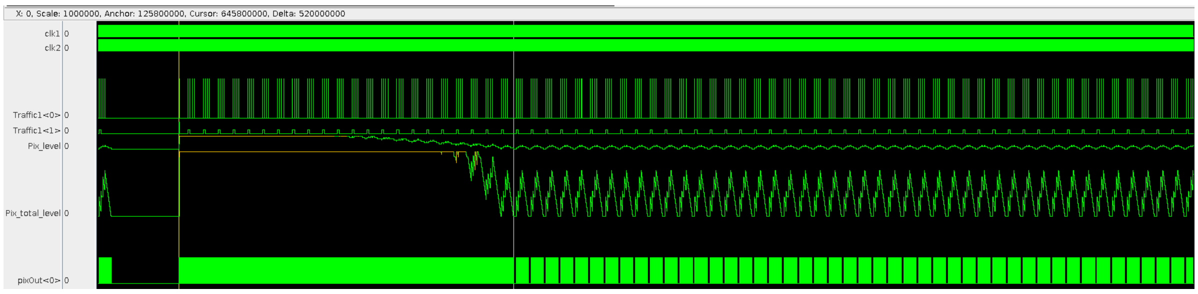

X: 0, Scale: 1000000, Anchor: 125800000, Cursor: 645800000, Delta: 520000000

Figure 4.15: Recovery time of the PIX4 process during burst behaviour due to network jitter and missing packets

A complex behaviour can be observed in this case with the system exhibiting high fluctuation in both the total active tokens of the queue as well as their respective size. It can be surmised that even in this exceptional overload however, the system does eventually recover to its normal operation in 520 us as is shown in Figure 4.15. This acts as a powerful proof of concept for the proposed architecture of the module, ensuring its normal operation under every possible case. Finally, the maximum overall needed buffer size is capped at 517920 bits or 64.74 kBytes, which helps determine design decisions regarding the FIFO later on.

4.3. The Ethernet core

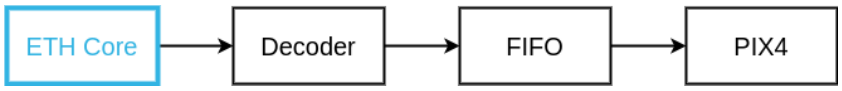


Figure 4.16: Components of the Switch - The Ethernet core

Following the data path, the first component one comes across when inspecting the design is the Ethernet core, as seen in Figure 4.16. All data processed and forwarded by the switching module originate from the aforementioned core. While technically the core itself was not developed as part of the design, a piece of hardware that faithfully mirrors its functionality had to be built for verification and validation purposes. Additionally, it is sensible for the working of the core to be inspected first, since the technical structure of the data as well as the various protocols they follow mandate the functionality of the remaining modules in the rest of the processing pipeline. These protocols, as well as the emulation module, will be presented and examined in greater detail in the sections to follow.

4.3.1. Packet structure

To better understand the structure of the testbench, the structure the packet flow originating from the Ethernet core should be examined first. As briefly mentioned in Section 4.1, the data produced by the core follow the Gigabit Ethernet Jumbo frame protocol, shown in Figure 4.2. The protocol allows for a payload sized of up to 9000 bytes. A single line of a typical 1080p stream consists of 1920 pixels or 5760 bytes, which is enough to be contained in a single Ethernet packet. Things become complicated when examining 4k streams, whose pixel density of a single line is 3840, or 11520 bytes, exceeding the capacity of a single packet. As such, for the remainder of this work, 1920x1080 will be considered the maximum possible resolution of any given stream with each packet accommodating exactly one line of said stream. The 4k case will be examined and elaborated upon separately, in the Future Works section. Thus, the structure of a typical Ethernet packet generally resembles the one illustrated in Figure 4.17.

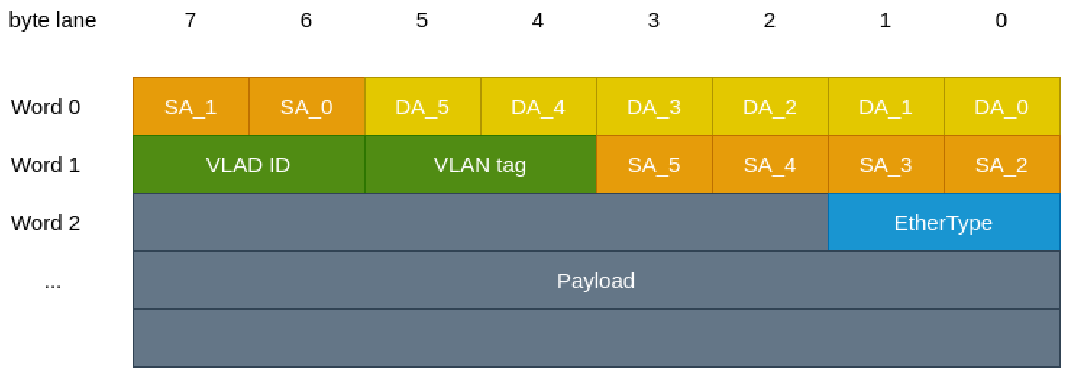


Figure 4.17: Typical structure of an Ethernet packet

The Ethernet core delivers a bandwidth of 10 Gigabits split into 64-bit wide strokes operating at 156.25 MHz, which means that the data are bundled in groups of 8 bytes, as shown in the figure. `DA` denotes the destination MAC address while `SA` corresponds to the source MAC address of the packet, both of which are unnecessary for the internal routing of the packet. The `VLAN tag` is predefined and should match the expected value in any correct packet. Furthermore, the 16-bit `VLAN ID` encapsulates the unique identifier of a given stream and is utilized further down the processing pipeline to forward packs to the correct destination. Finally, the `EtherType` field is typically used as a command/message identifier, but holds no significance to the switch.

With the distribution of data concerning the Ethernet protocol explained, the structure of the payload has to be examined next. The payload follows a streaming Ethernet protocol that is used to define how streaming data is transmitted over the network. It includes different 8-byte headers at the beginning of each packet, reserving unique values for the first and last packet of a frame. The headers additionally contain info regarding the frame number and packet number that the current payload belongs to. This information is used to identify missing or structurally incoherent packets in the later steps of the pipeline as well as take corrective measures when necessary. Furthermore, a completely separate packet is generated before a given stream begins transmission, containing information about the stream's resolution, pixel size and packet size among others. A stream can only be processed if the correct information regarding its transmission have been received prior to its beginning. Thus, a packet which corresponds to a single line of a frame of a given stream is generally structured as shown in Figure 4.18.

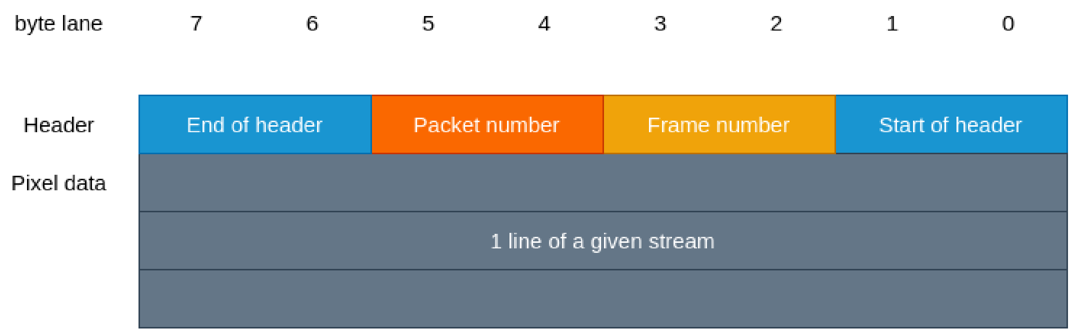


Figure 4.18: Typical structure of the payload

While pixel data representation can vary, during the scope of the current work we will only deal with three-channel pixels, corresponding to Red, Green and Blue, with each colour's information contained in 8 bits. Thus, every pixel will be represented by 3 bytes, or 24 bits. Since the core produces 64-bit wide signals, an irregularity is created where it is impossible to contain a whole number of pixels in a single stroke. As such, the transmission of the pixel data takes the form illustrated in Figure 4.19 in terms of RGB value alignment and in Figure 4.20 in terms of pixel alignment. It should also be noted that the design fully includes BGR colourspace support, for which the following figures still hold true with minor adjustments.

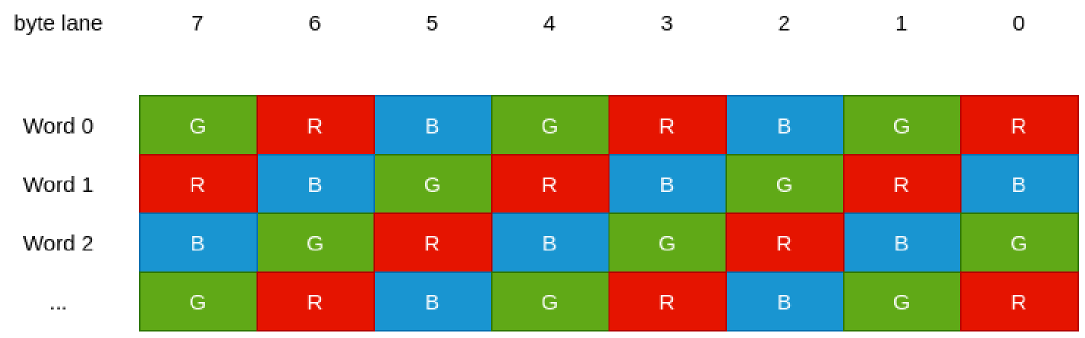


Figure 4.19: Pixel data structure in terms of RGB values

As can be seen, the pixel alignment shifts by 1 byte every clock cycle, with the pixel values lining up every 4th cycle. In other words, 8 whole pixels are transferred per 3 clock cycles from any given Ethernet stream. This fact may result in a misalignment at the end of the packet of a stream if the width of said stream, measured in pixels, is not divisible by 8. To combat this, padding may be appended to the packet to result in a whole number of transported pixels and is signalled along with the information packet of the stream. While none of the standard image resolutions falls under this category, this practice allows for the support of irregular stream resolutions as long as they do not violate the Ethernet

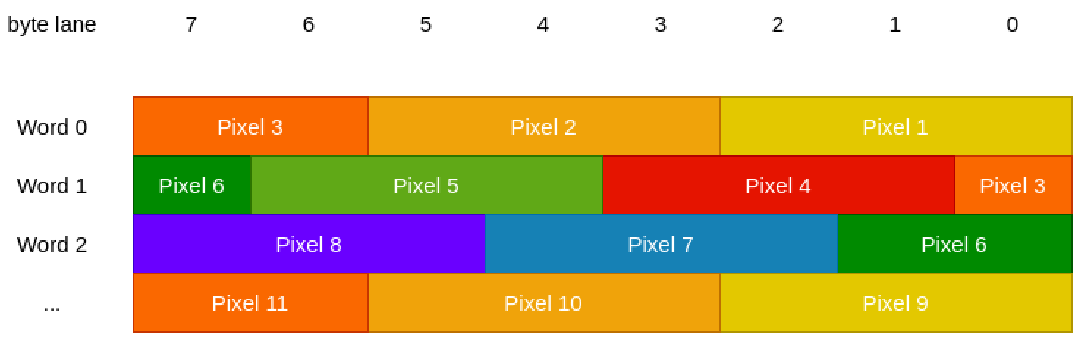


Figure 4.20: Pixel data structure in terms of individual pixels

bandwidth restrictions. One final note should be made here concerning endianness. As can be seen from the previous figures, the Ethernet core is little-endian, however the FPGA operates on a big-endian core. This should be taken into consideration when examining the pixel data on later modules.

4.3.2. The AXI-Stream interface

The packets, following the structure described above, connect to the FPGA via a 64-bit wide AXI4-Stream interface. AXI stands for Advanced eXtensible Interface and is a transmission protocol adopted by Xilinx and predominantly used in its devices. The AXI4-Stream protocol is used for applications that typically focus on a data-centric and data-flow paradigm where the concept of an address is not present or not required. Each AXI4-Stream acts as a single unidirectional channel for a handshake data flow [36]. Figure 4.21 showcases a typical transmission through AXI-Stream.

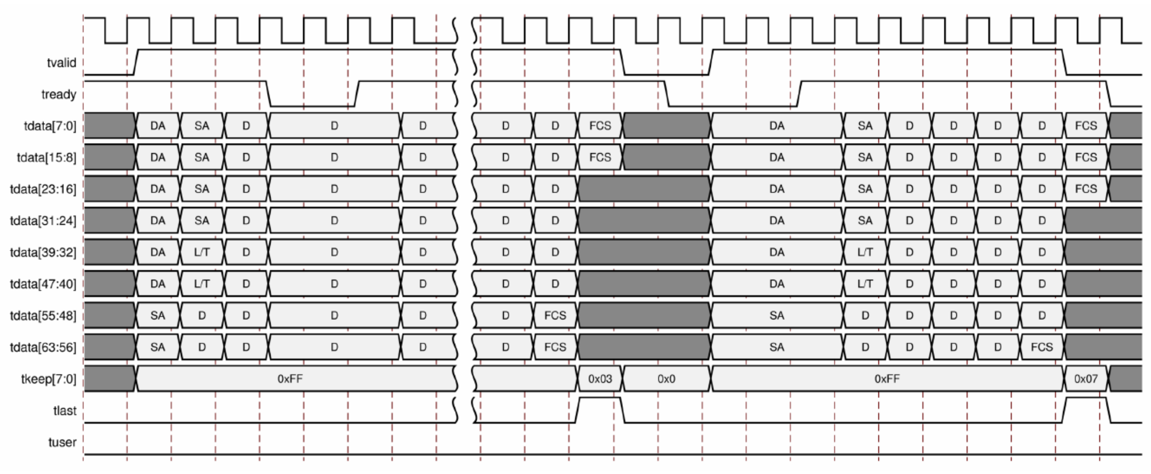


Figure 4.21: Transmission of data using the AXI-Stream protocol [36]

The signals depicted in Figure 4.21 are the typical AXI interface signals, and are present for the greater part of this switch design. A brief summary of their intended use is as follows. AXI signals not included in the summary are not used by the design in any part of it.

- **aclk** – While not explicitly labelled in Figure 4.21, aclk typically denotes the driving clock signa of an AXI bus.
- **tvalid/tready** – These are data handshake signals. The tvalid is generated by the source while tready by the sink on the bus. A data transfer takes place at the rising edge of the clock when both signals are '1'. In this particular case, the sink is always ready to receive data, and thus tready is tied to '1'.
- **tdata[(8n-1):0]** – Marks the actual data to be transferred. Xilinx IP blocks and Arm mandate that the data is always a multiple of 8 wide.

- **tkeep[(n-1):0]** – A byte enable collection signal which dictates whether a corresponding byte contains data or not. Used to mask the bytes in the last transfer of the packet. For this designs' purposes, tkeep is tied to "0xFF", meaning that all values in the bus are kept.
- **tlast** – Signals the last transfer of a packet. It is only valid when tvalid = '1'. Sampled at the rising clock when tvalid and tready are '1'. Ignored when tvalid = '0'. Since the stream has no knowledge of packet contents, this is the only signal signifying packet boundaries.
- **tuser** - These signals are typically used to add sideband data to a stream. They are not masked by tkeep. In the current context, this signal is a flag signifying that there is some error in the packet if tuser = '1'.

4.3.3. The testbench

In order to be able to verify the correct functionality of the system during and after its development, a testbench was initially built that imitates the behaviour of the immediate environment surrounding the Switch. That is to say, in this particular case, the testbench is faithfully simulating the functionality of the Ethernet core as has been described above. To achieve this, a piece of VHDL code was developed that could emulate the Ethernet core directly from the embedded CPU core of the FPGA, which was integrated as a block IP for simplicity purposes. To emulate the functionality of the core, the fabricated hardware consists of a 64-bit FIFO that is populated using two 32-bit writes. Initially, the high part is stored in a register for a single clock cycle after which the low part arrives and is written to the buffer in conjunction with the stored data. There exists a single control register supporting this procedure. Setting bit 0 of this register triggers the transmission of the packet, whereas setting bit 1 causes the `t_user` signal to be active during `t_last`, signifying an error in the packet. `t_keep` is omitted and assumed to be all '1's. It has been included in the interface since the actual core does produce it, but is not used by this design.

The emulated core design, whose block design is depicted in Figure 4.22, ultimately consists of:

- A clock wizard used to generate a 156.25 MHz and a 300 Mhz clock from the 99.99 MHz CPU clock which is provided as input.
- An AXI interconnect block operating at 156 MHz.
- A simple 8-bit output port to be used as proof of correct functionality on the actual board.
- An AXI clock converter to 300MHz, since the interconnect is unable to operate at that speed.
- An AXI interface logic towards two (one for each of the two frequencies) even simpler interfaces without use of a handshake, instead opting for a direct CPU bus without wait signals.
- The emulated Ethernet core and some test logic to prevent the synthesizer from optimising necessary components.

In order to exercise the simulated core, a test bench was constructed around the previously described entity accompanied by a Python script that generates a stimulus file. The script accepts input in the form of the following parameters for up to 4 different streams: height, width, frames per second and starting time (in ns). In return, it provides the precise timestamps (in ns) for the transmission of each packet of every stream, should there be no bandwidth violations. This stimulus file is then used by the testbench in order to trigger the actual packet generation, in compliance with the structure described above, and forward it through AXI-Stream. Additionally, the same stimulus file was used to trigger packet generation during the simulation phase in Section 4.2, allowing for direct comparison of theoretical and actual results. Since there are no actual pixel data to transport, they have instead been replaced by simple counters, which help identify pixel order and quickly pinpoint any missing or misplaced data. Because the module was never tested on the actual FPGA due to other modules still being under development, all validation for the Switch, including timing behaviour and functional correctness, was performed through the use of the emulated Ethernet core and the described testbench.

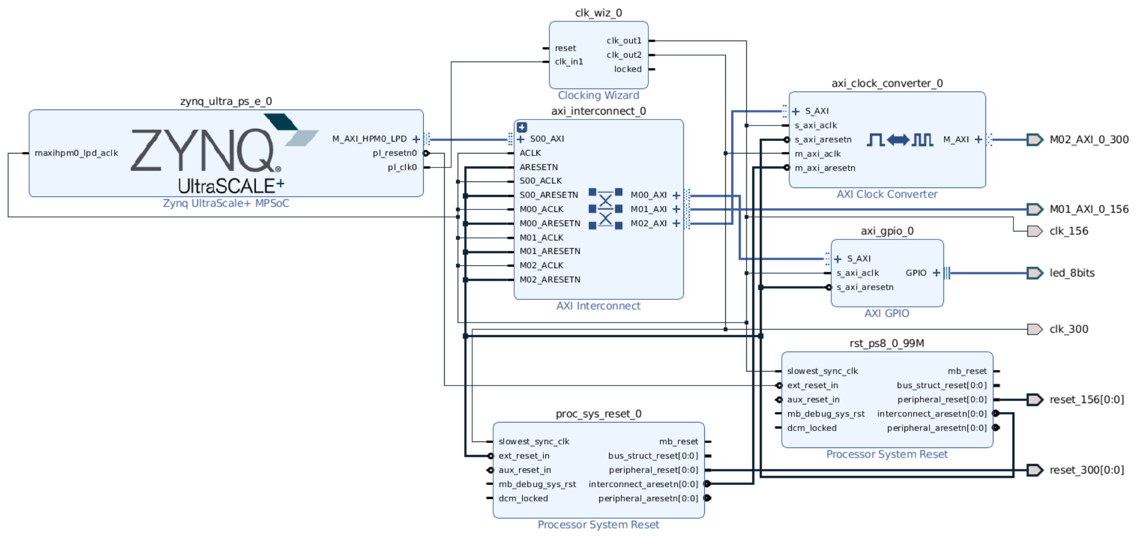


Figure 4.22: Block design of the emulated Ethernet core [captured through Vivado 2019.1.3]

4.4. The Decoder

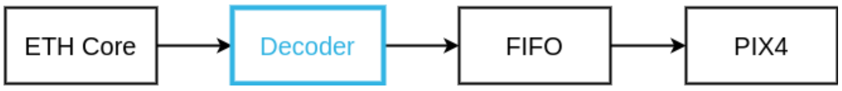


Figure 4.23: Components of the Switch - The Decoder

The next component in the pipeline, as shown in Figure 4.23, is the Decoder module. Here, the first real processing of the raw data provided by Ethernet core takes place. The Decoder is responsible for extracting raw information from the header packed input payloads, attribute information about the structural integrity and synchronisation of the packets as well as relay them forward. It additionally acts as a filter for unknown or non-relevant streams and sideband packets that arrive through the core.

4.4.1. The algorithm

The Finite State Machine (FSM) depicted in Figure 4.24 illustrates the approximate algorithm by which the block functions. The Decoder is constantly monitoring all data passing through the Ethernet core, scouting for the `tvalid` AXI signal to be active. Upon positive identification of `tvalid`, the VLAN tag, as shown in Figure 4.17, is asserted, whereas the VLAN ID is stored in a register. Should the tag be accepted, the next info to be processed is the packet header, as showcased in Figure 4.18. The header is examined for correctness as well as for the type of information contained in the packet. Data from an information packet are temporarily stored in registers until packet integrity is guaranteed, at which point they are copied to permanent memory locations and a flag is raised for the corresponding stream, signalling that incoming packets from said stream can now be processed. On the other hand, if a data packet is identified and its information has been stored prior to its arrival, the contained pixel data is simply forwarded to the FIFO in order to be processed by the PIX4 block. Accompanying the pixel data, the Decoder bundles information such as a `vsync` signal on the first packet of a frame, a 2-bit `stream_id`, which is an encoded version of the VLAN ID and a `colourspace` single bit signal indicating either an RGB or BGR stream. Finally, should any of the `tvalid`, VLAN or header information be inaccurate, the module transitions to a state where it is simply consumes the input until a delimiter arrives, and the process is reset. It should be noted here that both Figure 4.24 as well as the above outline are only intended as a comprehensive summary of the module and do not fully reflect the working of the actual hardware. While the Decoder is indeed modelled as a finite state machine, the actual schematic is significantly more complex.

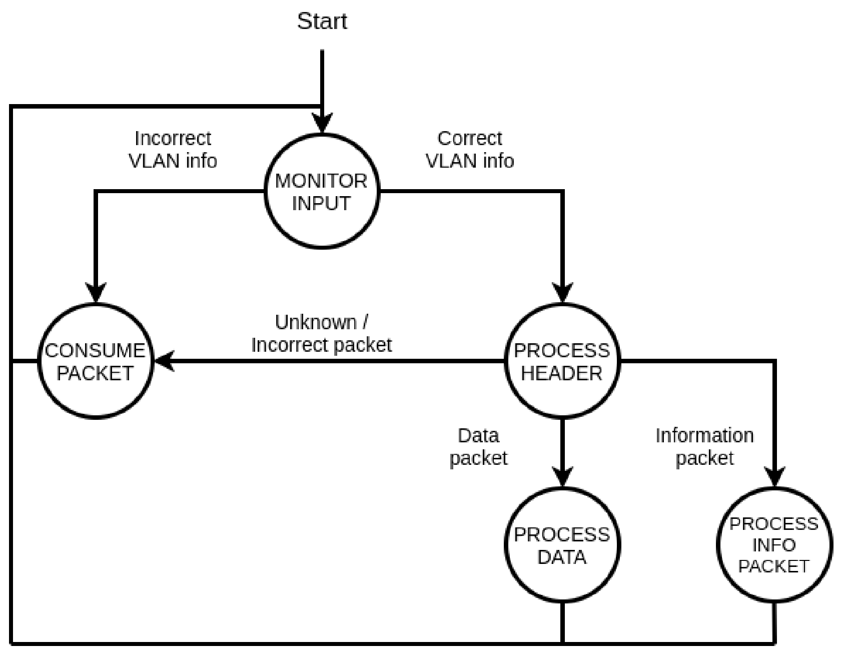


Figure 4.24: Finite state machine of the Decoder

As described previously, the only indication of packet boundaries for all these processes is the `tlast` signal, which follows the final piece of data and is asserted for a single clock cycle. As such, is constitutes the sole indicator for when the state machine should terminate any processing state and proceed to the next one. One complication stemming from this fact is that the last piece of data produced by the core is the 4 CRC bytes and not the actual pixel information. Thus, the Decoder delays the entire output signal by a few clocks to correctly synchronise with `tlast`.

Additionally, the decoder is responsible for detecting structural faults in incoming packets and signalling errors, since it is in sole possession of the complete information of each stream. The scalers are able to process information with '0' coefficients applied, essentially ignoring part of the data, but they are not able to cope with a complete lack of information. Such a case would occur if a packet is either shorter or missing entirely. This fact places the structural integrity of the packets at a much higher importance than that of their actual content. Due to this reason, the Decoder is responsible for making sure the data arriving at their destination match their intended size. To achieve this, 2 distinct mechanisms are applied, padding and cropping. When `tlast` is asserted, the Decoder compares the current size of the received data with the corresponding stored information variable. If the former is bigger, the packet is simply cropped to the correct size and forwarded. In the opposite situation, an `error` signal flag is raised and information regarding the amount of missing data is appended to the packet, adhering to the interface described in the following section, in order to command the PIX4 block to reconstruct it. Missing packets are handled in a similar fashion. The Decoder additionally performs a plethora of error signalling to the assisting software. The more comprehensive algorithm is presented in Section 4.7.

4.4.2. The Decoder - PIX4 interface

During the design process, it was decided that the Decoder and PIX4 modules would interlink in a master-slave relationship. This decision was made so that entire burden of error detection could be concentrated on the Decoder instead of opting for a more distributed approach, which is prone to more design errors and breeds significantly more corner cases. As such, a strict interface was devised for the transmission of data between the two components. The intention was that should both components be fully compliant with the interface, error-free operation would be guaranteed. This interface is defined below and is separated between transmission of integral and misshapen packets.

Normal transmission

The timing of a normal packet transfer is shown in Figure 4.25. When the decoder module wants to transmit a packet, the 64-bit `pixel_data` (D) are accompanied by the corresponding, translated 2-bit `stream_id` and `colourspace` info (0 indicating RGB, 1 indicating BGR). The `vsync` signal, if any, is transported along with the first pixel data instance of the appropriate stream. The end of the packet is indicated to the PIX4 module by `tlast` asserted for one cycle. Under normal transmission `error` is unused. All data pushed by the decoder module into the FIFO are considered valid and as such, no `tvalid` signal is required.

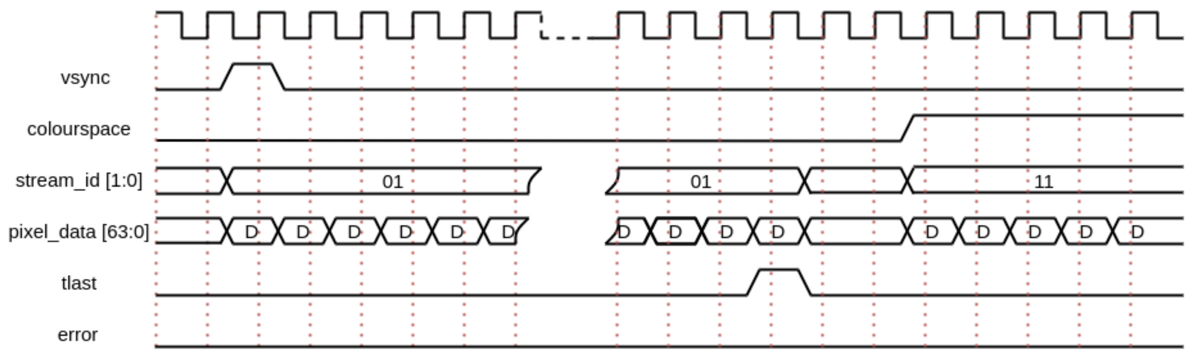


Figure 4.25: Normal packet transfer of two different streams

In this example, two different streams, with IDs 01 and 11 respectively, are transmitted from the Ethernet core and through the Decoder. Stream 01 is accompanied by a `vsync` signal as well a '0' `colourspace`, indicating that this is the first packet of a frame, and the image is in the RGB domain. Stream 11, however, is operating on the BGR domain and is transmitting some intermediate, or the last packet instead.

Transmission with error

Figure 4.26 depicts how the transmission of data from a stream containing errors occurs. When a mismatch in the data length is detected in the Decoder module, `error` is raised along with `tlast` and the final bits of the `pixel_data` are, instead, set appropriately to indicate the number of missing pixels to be reconstructed (PP). On the PIX4 module, `tlast` and `error` are asserted in the same clock cycle.

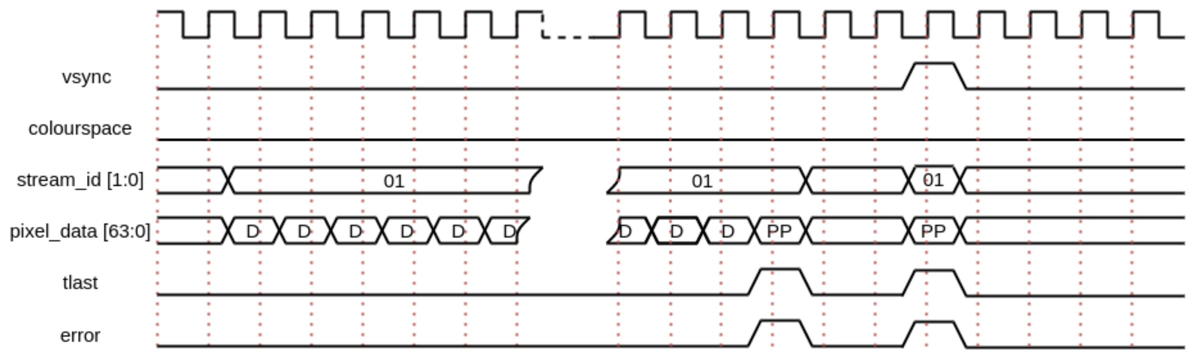


Figure 4.26: Packet transfer of a single stream containing errors

In this example a given stream with an ID of 01 transmits two packets containing errors. The first packet, which is incidentally also the first packet of the frame as indicated by `vsync`, is missing entirely, while the second packet is clipped. Thus, on both instances, the Decoder appends the amount of missing data that need to be padded by the PIX4 and raises the error flag. For the first packet, this amount corresponds to the accompanying packet size of the stream, which has been received by its information packet, triggerring the PIX4 block to reconstruct it completely rather than partially. This back-to-back transfer is common in missing packet situations, since they can only be detected by the time the some data of the same stream arrive.

4.5. The cascaded FIFO

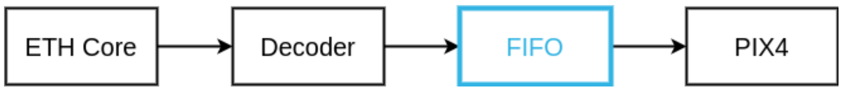


Figure 4.27: Components of the Switch - The FIFO

Once the filtered and synchronised data leave the Decoder, they are buffered into the FIFO, as seen in Figure 4.27. As has been briefly described before, the FIFO here is actually not a single element, as can be derived from Figure 4.28. It is, instead, comprised of two smaller First-In-First-Out (FIFO) queues. The former is a sizeable, synchronous buffer while the latter is a small and asynchronous buffer, interconnected in a cascaded manner. The role of the synchronous queue is to absorb the difference that is created between the operational frequency of the Decoder and the PIX4 block, along with any irregularities that stem from burst behaviour, as detailed in Section 4.2.4. On average, during a typical 1080p stream transmission, the large queue needs to absorb 538 pixels, or 12912 bits, which equals 202 64-bit entries. Of course, this does not account for burst behaviour occurring due to errors, jitter or both, which is the reason for the extended simulation analysis presented earlier. In contrast, the small queue needed to be able to accommodate only a few elements, possibly even a single element, at a time. That is because it was tasked with handling the difference between the driving clocks of the Decoder and the PIX4 block, those being 300 MHz and 156.25 MHz, respectively.

The comprising queues were actualised through use of the FPGA fabric's own embedded memory elements. Xilinx devices typically offer 3 different available options of embedded memory types that that can be used as random-access memory (RAM), read-only memory (ROM), or shift registers. Those are the following [37]:

- **URAM** - Ultra Random Access Memory or URAM is a large capacity, synchronous memory block. URAM blocks can be cascaded together to create large on-chip memories. Every UltraRAM block is a dual-port synchronous 288Kb RAM with fixed configuration of 4,096 deep and 72 bits wide. Port A and Port B share the same clock signal. Within a single cycle of the external clock, the Port A operation always completes before the Port B operation. Each port can independently perform either one read or one write operation per clock cycle.

- **BRAM** - Block Random Access Memory or BRAM is a dual-port RAM module instantiated into the FPGA fabric to provide on-chip storage for a relatively large set of data. The two types of BRAM memories available in a device can hold either 18k or 36k bits, and the available amount of these memories is device specific. The dual-port nature of these memories allows for parallel, same-clock-cycle access to different locations. BRAMs can be configured to support asynchronous behaviour.

- **Distributed** - As previously discussed, the LUT is a small memory in which the contents of a truth table are written during device configuration. Due to the flexibility of the LUT structure in Xilinx FPGAs, however, these blocks can be used as 64-bit memories and are commonly referred to as "distributed memories". This is the fastest kind of memory available on the FPGA, because it can be instantiated in any part of the fabric, which drastically improves the performance of the implemented circuit.

The available elements here were presented in order ranging from most inflexible to most flexible, but also from biggest to smallest. It can be seen that any choice of memory type comes with significant tradeoffs, which leads to difficult design decisions. While the big FIFO could theoretically be built by any of the three memory types, its size had to be at minimum 12912 bits and at maximum 517920 bits, rendering it far too large for a LUT-based implementation. As far as the remaining options are concerned, every URAM block has a fixed size of $72 * 4096 = 294912$ bits, which does not perfectly

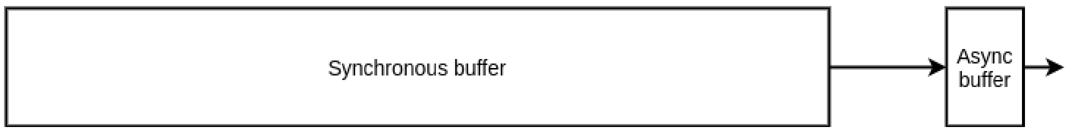


Figure 4.28: Components of the FIFO module

align with the 64-bit width of the Decoder data. When accounting for the accompanying control signals, the total width of the output signal of the Decoder becomes 70 bits, as described in Section 4.4.2. This leaves the total maximum utilisation of each URAM block to $70 * 4096 = 286720$, or 97%. This also means, however, that a single URAM block is enough to resolve every possible data burst scenario except the single, most extreme case examined. Contrary to the above, BRAM blocks offer much more flexibility, and through this increased granular behaviour the user is capable of building a buffer that will achieve close to 100% utilisation. The downfall in this case is that every BRAM block has a capacity of 36864 bits, leading to a need of 8 such blocks in order to achieve the previously mentioned memory size of 286720 bits, that a single URAM block can provide. While positives and negatives can be argued towards either side, the decision was ultimately taken to build the large buffer using a single URAM block. This verdict takes into account that the provided capacity is enough to easily handle 4 missing packets of any stream, 4 concurrent input packets due to jitter as well as almost any combination thereof. The event of 4 input streams colliding, each accompanied by 4 missing packets, as analysed in Section 4.2.4, was deemed so exceptionally unlikely that it worth handling as a separate case by a dedicated handler.

On the other end of the spectrum, the available memory options for building the small buffer are limited to either BRAM blocks or LUTs, since URAMs are strictly synchronous. Given that optimising read and write speed is not a concern here, both of the aforementioned options produce the same result. However following the distributed memory paradigm offered a few subtle but distinct advantages. Firstly, as discussed previously, this buffer can provide its intended functionality even when its capacity is as small as a single Decoder output element, rendering the excess storage provided by the BRAM unnecessary. This is especially true when considering that the choices taken regarding the synchronous buffer led to more than enough overall capacity and it is extremely unlikely that this extension will be utilised. As such, the incentive was to make the asynchronous buffer as small as possible, minimising resource usage. A second, less apparent, but possibly even more important advantage concerns the overall design's timing behaviour. Unlike the rest of the design, whose fabrication is based on programmable LUTs and can be placed seemingly arbitrarily on the reconfigurable fabric of the FPGA, the embedded memories physically exist in set locations on the device. Opting for "distributed memory" here, which is also LUT-based, allows the software to optimise routing with a single constraint, a path towards a single URAM block and back. Choosing a BRAM block would further burden this constraint by including an additional necessary physical path from the URAM to the BRAM component before going back to the design. Additionally, in that case, the order of the two buffers suddenly plays a significant role in the routing, with the async buffer being first leading to noticeably worse results.

It needs to be noted that all the choices made during the previous analysis are only deemed optimal in a vacuum. Namely, available FPGA resources which might have been reserved for a different part of the compositor were not taken into consideration due to the project still being under development, and could change the direction of the FIFOs design vastly. Additionally, not every Xilinx FPGA supports URAMs, possibly leading to a different configuration if portability and migration is a concern. That said, it is entirely possible for both the buffers to be implemented efficiently using only BRAM blocks.

As far as actual implementation is concered, both FIFOs were instantiated components from the Xilinx Parametrisable Macros (XPM) library. Figure 4.29 showcases the resulting hardware modules and their basic interconnection, whereas figures 4.30 and 4.31 provide an example of the buffers during a synchronous write and an asynchronous read operation respectively.

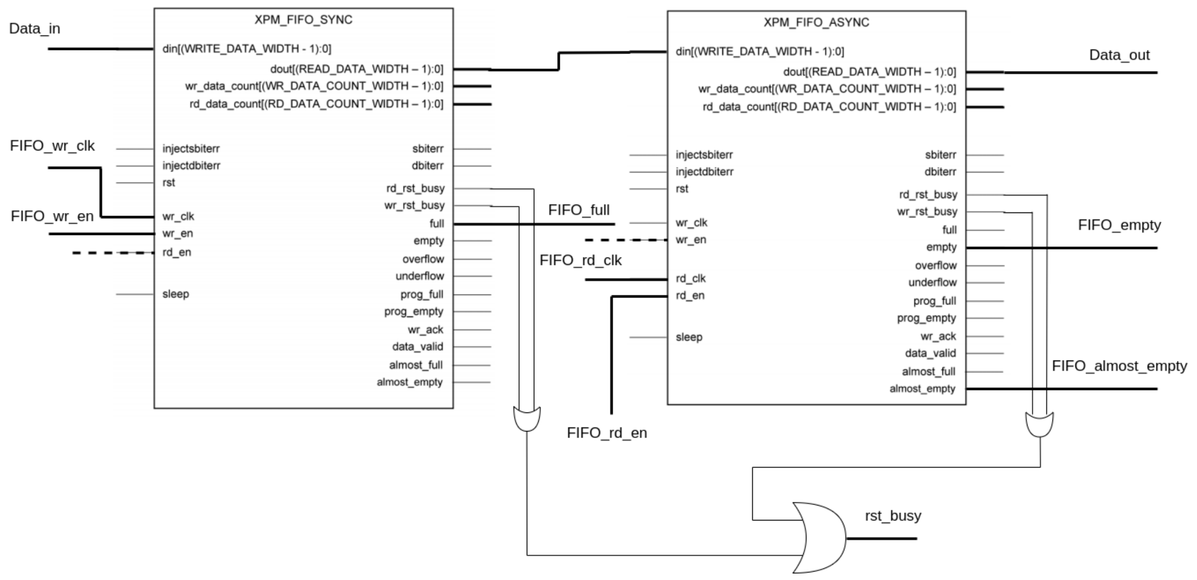


Figure 4.29: Schematic of the XPM instantiated FIFOs and their basic interconnections [38]

The macros allow for impressive parametrisation, allowing the user to fine-tune their behaviour. However, under the context of the current work, only the parameters relevant to the FIFO functionality will be discussed. Probably the most critical of said parameters is the memory type to be used during the instantiation. This was the topic of the previous extensive analysis, the results of which have been presented above. Another important parameter is the `READ_MODE` of the resulting queues, which has been set to First Word Fall Through (FWFT), as depicted in figures 4.30 and 4.31. During this mode, the first data element in the queue is immediately forwarded towards the output, without the need for a read enable (`rd_en`) signal. This essentially remodels the `rd_en` to act like an acknowledgement that the data have indeed been received. Since this can lead to adverse effects regarding perceived timing behaviour, the advanced feature to include the `almost_empty` was enabled. This signal becomes active when there is a single element in the queue, which due to the previously mentioned behaviour is set to be forwarded leaving the queue empty. As such, this signal replaces the traditional `empty` in function.

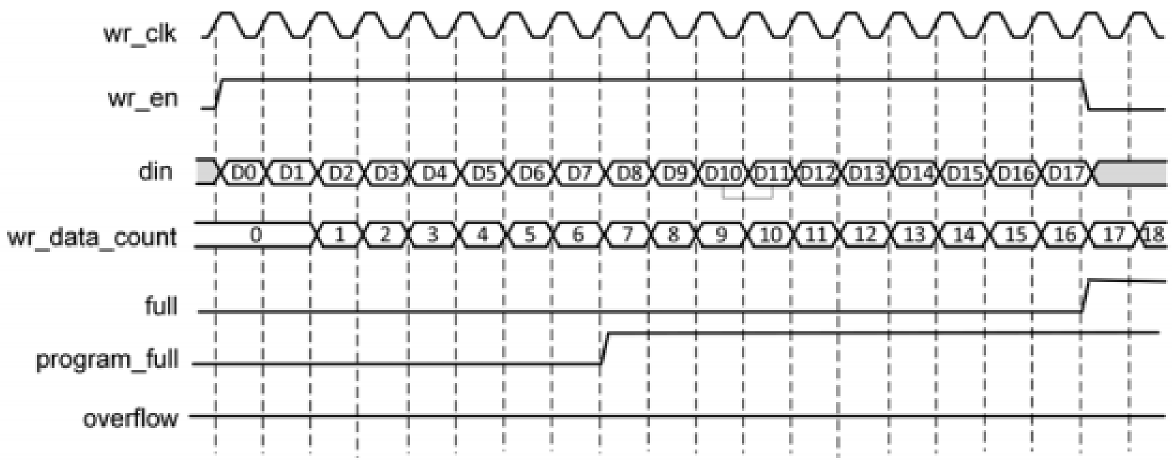


Figure 4.30: Synchronous FIFO during a write operation under FWFT mode [38]

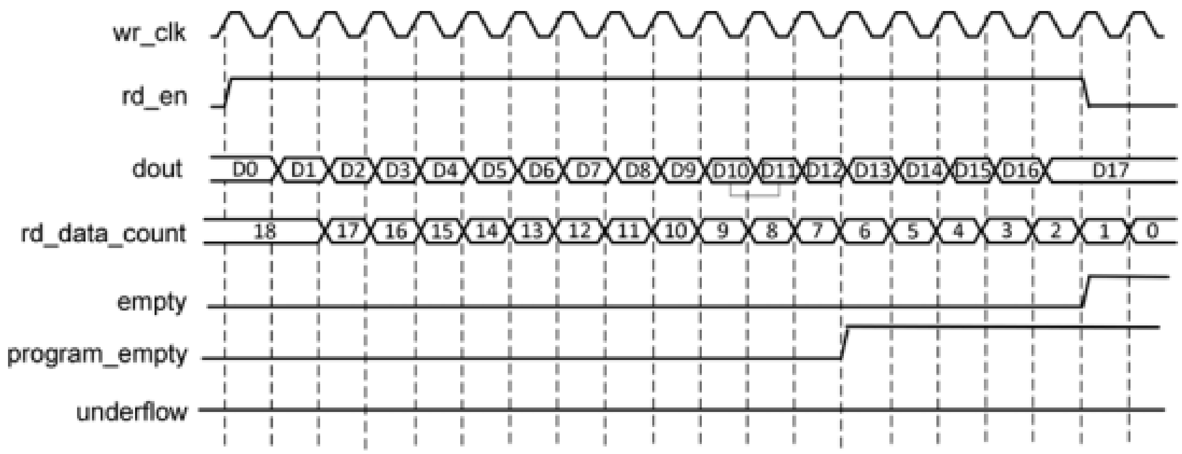


Figure 4.31: Asynchronous FIFO during a read operation under FWFT mode [38]

4.6. The PIX4 block

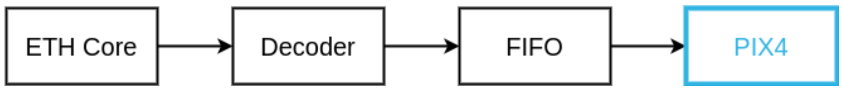
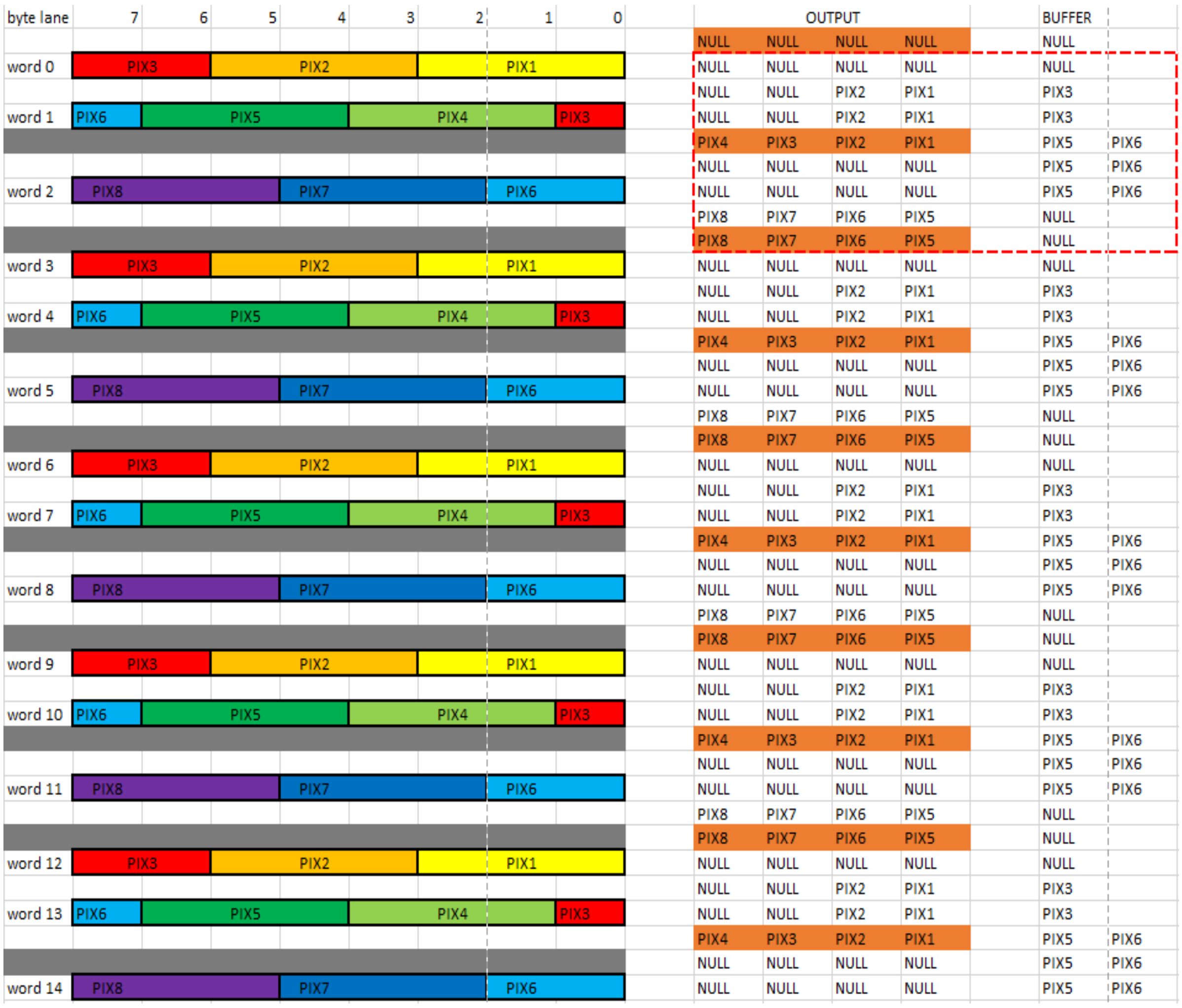


Figure 4.32: Components of the Switch - The PIX4

The final step in the processing pipeline is the PIX4 block, as indicated in Figure 4.33. The PIX4 is mainly responsible for producing groups of 4 pixels every 4 clocks from the data it receives from the Decoder, after traversing through the FIFO. This would have been a relatively straightforward task if there was a whole amount of pixels transported every clock. On the contrary, as described in Section 4.3.1, there are approximately 2.66 pixels being transported every clock tick through the buffer, or 64 bits in terms of data. The effective bandwidth of the module is 7.2 Gbits, making it slightly faster than the 7 Gbits limit that is imposed on the Ethernet fibre per specification. That means that, theoretically, no matter the load on the fibre and given a large enough FIFO, the PIX4 block should be outputting data faster than they can arrive at its input. Considering that the Ethernet core is usually not operating at 100% capacity, this difference should be significantly wider in most situations. This theory is supported by the simulations detailed in Section 4.2, showing that even under extreme situations no buffer overflow takes place and the system recovers normal operation within a reasonable amount of time.

Unfortunately, the irregular alignment of pixel data within the channel, as described in Section 4.3.1, reflects in an equally irregular behaviour at the PIX4 block. The typical operation of the module is showcased in Figure 4.33, broken down in cycles. The left side corresponds to the pixel data available at the output of the FIFO whereas the right side coincides with the output vector and internal buffer of the PIX4 block. The orange marked lined on the output vector columns denotes the the cycles when the module produces an output, after which the vector is emptied. Every line corresponds to a single cycle of the 300 MHz clock, hence the gaps between inputs on the left side. Furthermore, the block requires 96 bits of pixel data in order to form and forward a group, which occurs every second and third transmission if the 32 leftover data from the second packet are buffered. This means that if the module requested data immediately whenever those became available in the FIFO, it would be overwhelmed. Thus, every 4th cycle, the module does not request new data, instead performing internal processing on the ones already available to it. This behaviour is denoted by the grey lines in the figure. Finally, the internal buffer is used to temporarily store fragmentary pixels, which are transferred to the output vector once completed. Following this practice, the PIX4 exhibits a repetition of its operational cycle every 8 clock cycles, which is marked by a red outline in Figure 4.33 and further showcased in Figure 4.34. This cycle continues until an active `tlast` is asserted, triggering a termination procedure. During that time, the block retains its last output steady for 4 cycles so that it can be safely sampled by the time-division MUX, after which it returns to an idle state, polling for new inputs. It should be noted that should PIX4 identify a BGR stream through `colourspace`, then it simply swaps the R and B channel values during the aforementioned output cycles, since the fragmented data has already been collected and grouped.



OUTPUT				BUFFER	
NULL	NULL	NULL	NULL	NULL	
NULL	NULL	NULL	NULL	NULL	
NULL	NULL	PIX2	PIX1	PIX3	
NULL	NULL	PIX2	PIX1	PIX3	
PIX4	PIX3	PIX2	PIX1	PIX5	PIX6
NULL	NULL	NULL	NULL	PIX5	PIX6
NULL	NULL	NULL	NULL	PIX5	PIX6
PIX8	PIX7	PIX6	PIX5	NULL	
PIX8	PIX7	PIX6	PIX5	NULL	
NULL	NULL	NULL	NULL	NULL	
NULL	NULL	PIX2	PIX1	PIX3	
NULL	NULL	PIX2	PIX1	PIX3	
PIX4	PIX3	PIX2	PIX1	PIX5	PIX6
NULL	NULL	NULL	NULL	PIX5	PIX6
NULL	NULL	NULL	NULL	PIX5	PIX6
PIX8	PIX7	PIX6	PIX5	NULL	
PIX8	PIX7	PIX6	PIX5	NULL	
NULL	NULL	NULL	NULL	NULL	
NULL	NULL	PIX2	PIX1	PIX3	
NULL	NULL	PIX2	PIX1	PIX3	
PIX4	PIX3	PIX2	PIX1	PIX5	PIX6
NULL	NULL	NULL	NULL	PIX5	PIX6
NULL	NULL	NULL	NULL	PIX5	PIX6
PIX8	PIX7	PIX6	PIX5	NULL	
PIX8	PIX7	PIX6	PIX5	NULL	
NULL	NULL	NULL	NULL	NULL	
NULL	NULL	PIX2	PIX1	PIX3	
NULL	NULL	PIX2	PIX1	PIX3	
PIX4	PIX3	PIX2	PIX1	PIX5	PIX6
NULL	NULL	NULL	NULL	PIX5	PIX6
NULL	NULL	NULL	NULL	PIX5	PIX6
PIX8	PIX7	PIX6	PIX5	NULL	
PIX8	PIX7	PIX6	PIX5	NULL	
NULL	NULL	NULL	NULL	NULL	
NULL	NULL	PIX2	PIX1	PIX3	
NULL	NULL	PIX2	PIX1	PIX3	
PIX4	PIX3	PIX2	PIX1	PIX5	PIX6
NULL	NULL	NULL	NULL	PIX5	PIX6
NULL	NULL	NULL	NULL	PIX5	PIX6

Figure 4.33: Unfolded operational cycles of the PIX4 block

An additional function of the PIX4 block, which was briefly mentioned in Section 4.4.1, is the reconstruction of broken or missing packets under the directive of the Decoder. To incorporate that functionality, the block is equipped with a "Pad Mode". As described in Section 4.4.2, any transmission containing structural errors is accompanied by the `error` signal. Upon encountering `error`, the block immediately engages its Pad Mode, continuing its operation beyond the `tlast` mark. However, under this mode, no new data is requested during the "data request" cycles. Instead, the module replaces the nonexistent data with a set of predefined values and signals the scalers to process them with a '0' coefficient, effectively ignoring them. The Pad Mode is automatically disengaged when the final piece of missing data has been reconstructed, returning to an idle state. Finally, as was the case upon the termination of the transmission, PIX4 is responsible for keeping all outputs steady for exactly 4 clocks so they can be sampled correctly. This applies to the synchronisation signals `vsync` and `stream_id` as well.

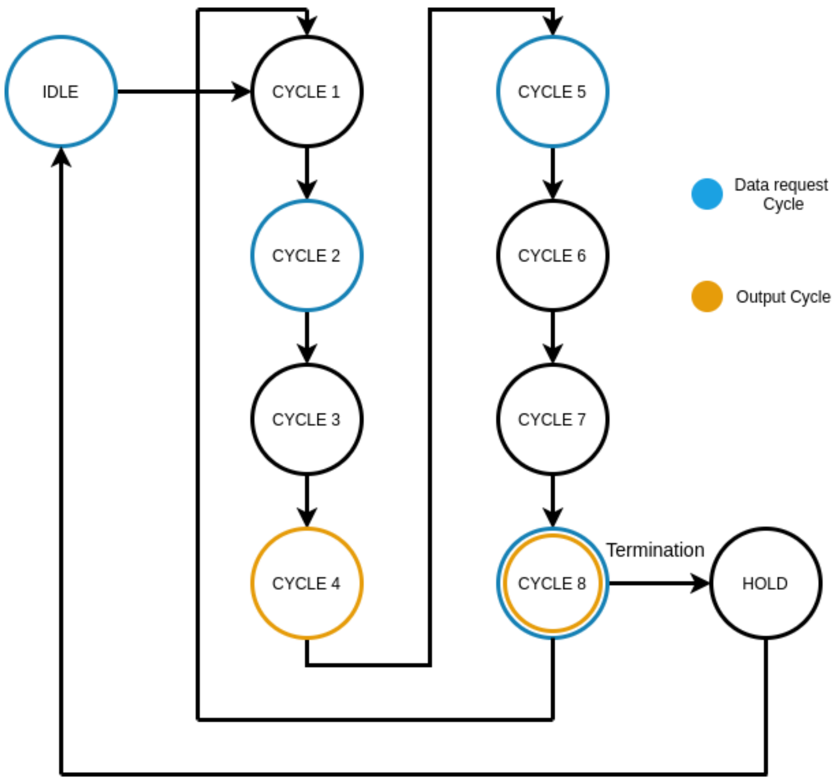


Figure 4.34: Finite state machine of the PIX4 block

4.7. Error handling

Since all stations of the processing pipeline have been clarified and detailed, it is important to elaborate on the subject of possible errors that can ensue in the system and how they are managed by it. Although the subject was touched upon briefly during the previous sections, there was only ever enough information provided so as to cover each respective issue that was examined at the time. Figure 4.35 roughly depicts the algorithm which the system follows in case of an error. To begin with, the errors present in the system can be divided into two broad categories, the first of which is jitter-induced errors. Jitter can originate in two locations in the surrounding networks' topology, which has been presented in Figure 4.11. Those correspond to the two points of possible congestion, namely the external network switch and the conversion box. Both of them additionally stem from the same cause which is contention for the same output slot. In this system, jitter can disrupt the periodic supply of input data coming from the Ethernet cores, causing them to arrive later than intended and while the PIX4 block is still in processing. At worst, the packets can coincide entirely, resulting in a much higher load for the PIX4. This case has been extensively analysed during Section 4.2.4. Jitter-related errors are generally resolved through buffering, provided that the system is outfitted with a large enough buffer. As described before, a single URAM block in the FIFO design is more than capable of withstanding this, and as such these errors are considered solved.

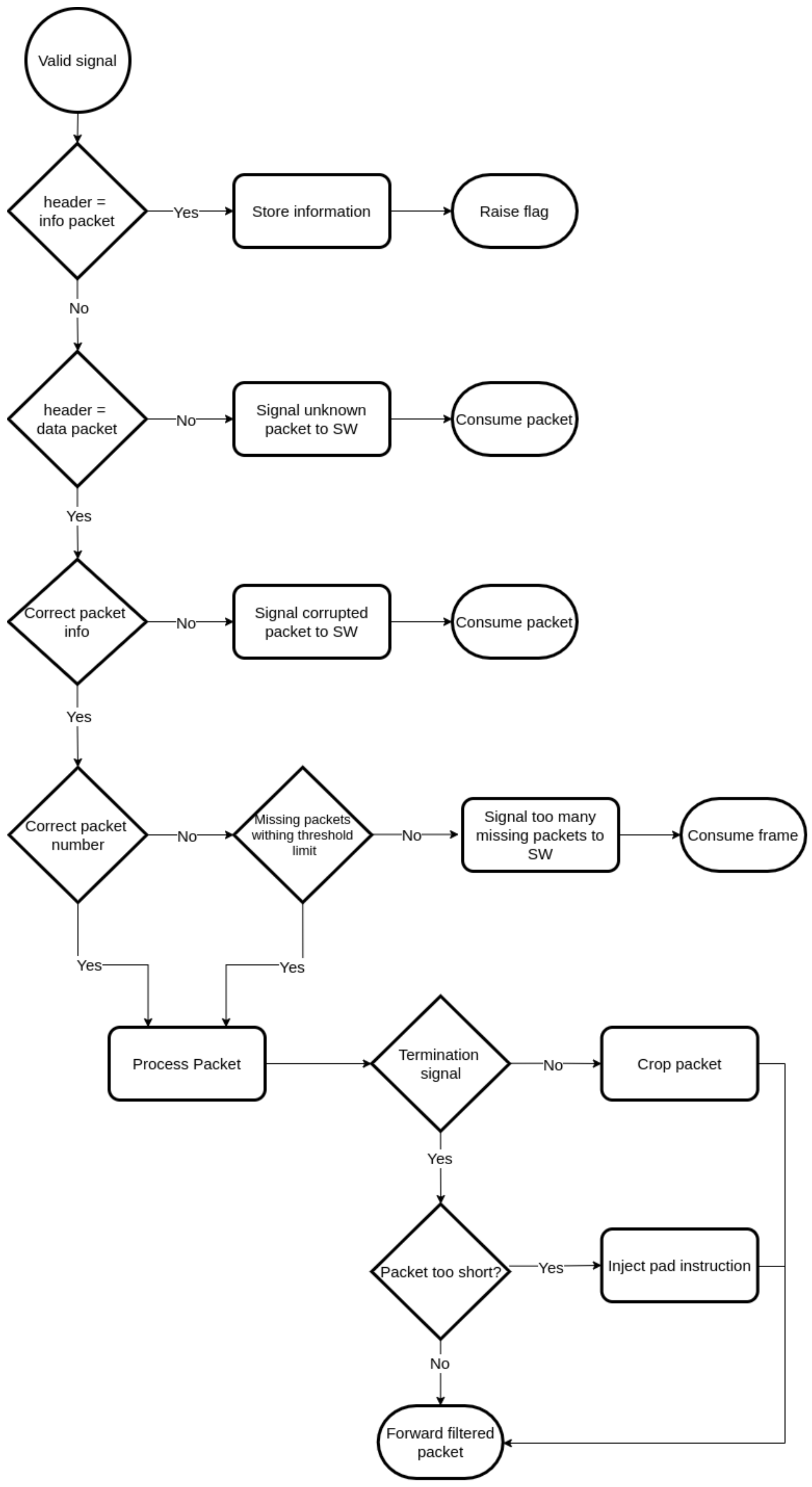


Figure 4.35: Flowchart of the system's error handling algorithm

The second category corresponds to packet errors, which constitute the various errors that can occur on the data itself during transmission. An assumption is made that all data are considered equal and alterations can potentially be found in any containing field, e.g. the Ethernet headers, packet headers or pixel data. This leads to a sizeable amount of probable packet errors, however, not all of them prove equally detrimental to the system. For example, an error in the VLAN tag or VLAN ID field of the Ethernet header will result in the stream not being accepted for further processing and nothing appearing on the output screen. In contrast, a packet being of shorter length than expected might result in severe malfunctions at the downscaler which is not aware of this fact, has no indication of packet boundaries and processes a given stream's line in a single stride. Even worse, such an error can have cascading effects in every other stream that the same scaler is responsible for processing. Fortunately, all of the destructive errors have a common thematic of interfering with a given packets' structural integrity and are found in the following cases:

- Packets that are bigger than their predefined size.
- Packets that are smaller than their predefined size.
- Missing packets.

Because of their severity and the hazard they pose to the system, those errors are identified and rectified directly by the processing pipeline, before they arrive at the scalers. For every incoming packet, during the assertion of `tlast`, the Decoder performs a comparison of the packets current size and its intended size. If the packet is found too short, the Decoder will instruct the PIX4 block to pad the missing data with dummy values, as explained in Section 4.4.2. In the inverse case, i.e. when a packet has exceeded its intended size with no indication of `tlast`, the Decoder will simply crop the packet at that point, signal the auxiliary software about the occurrence and forward it. While both might result in brief visual glitches on the output screen, they are usually too fast to be perceived by the human eye. More importantly, if they are handled this way they will not be destructive to the system, allowing for normal operation to resume for the following packets. On the implementation side, the decoder can only detect missing packets and not with missing pixels. As such, the amount of data that the PIX4 block is directed to reconstruct (PP) are expressed in bytes, rather than whole pixels. Packets that are missing entirely, rather than being fragmented, are also handled in a similar manner in terms of regeneration. They are simply thought of by the decoder as "short" packets containing no data. They differ in two particular areas however, the first of which is their identification method. The Decoder relies on the packet headers, which contain their corresponding sequence number, in order to realise that one, or potentially more, is missing. This fact leads to their second distinction which is that they can only be found out when the next packet arrives. As such, they breed situations similar to those of jitter-induced errors and need to be handled in an analogous manner, through buffering. The worst-case scenarios as well as their unlikely combination with jitter have been analysed in sections 4.2.4 and 4.2.4, respectively. Furthermore, there an additional type of errors that is handled directly by the pipeline simply for efficiency reasons, although its effects are less severe. Those are CRC errors that can be corrected on the spot.

Other than the two main scenarios discussed above, there is a plethora of potential errors that can happen, some of which are represented in Figure 4.35, while many of the minor ones have been omitted to prevent cluttering. As can be seen, the response to most lesser errors that do not thwart the operation of the system is to increment a dedicated hardware counter and notify the assisting software via a register. Some additional aspects concerning the way that missing packets are managed should be clarified. At the time of writing, the system based its decision via a simple comparison to a set threshold. Should there be more than 4 consecutive missing packets, the entire frame was dropped and the system waited until the next `vsync`. This implementation is very primitive and quite naive since it allows for continuous occurrences of this behaviour, which might overload the buffer and would have visible results on the output image. One way to mitigate this is through the use of a more sophisticated buffer control algorithm. Another, more elegant approach is discussed in future work, during Section 6. Finally, it should be mentioned that the decoder keeps an additional timeout counter for each stream which signals a stream disconnect in case it detects no activity for a certain amount of time.

A note should be made here on how likely those errors actually are to transpire in the first place. As per specification, the maximum allowable Bit Error Rate (BER) for 10 Gb Ethernet connections, such as the ones used in the current work, is 10^{-12}. Assuming a possibility of error in every packet, this corresponds to a single error every 20000 frames or every 300 seconds. The recommended and most cost effective BER for long distance communication is 10^{-13}, when distances of 100 km with repeaters in 2 km segments are concerned. For the current project, where long network segments are not present and distances do not cross 30 m by default, a BER of 10^{-18} is achievable, corresponding to a single error every 4.5 years. For the sake of a pessimistic analysis, a bad network with a BER of 10^{-12} is assumed. Since a stream is subjected to two network segments before arriving at the Ethernet core, the error rate doubles to $2 * 10^{-12}$, or 1 error every 150 seconds. Another assumption is made that every one of the aforementioned cases is encapsulated by a bit error and falls under this probability distribution. Furthermore, all errors are considered unrelated incidents and as such no conditional probabilities are involved in the case of consecutive events. While this second assumption is appropriate for most types of errors, it is not entirely accurate for the case of missing packets, which tend to happen in sequence and thus correlate to each other. Additionally, the BER can be considered as an approximate estimate of the bit error probability, which is accurate for a long time interval and a high number of bit errors. With the aforementioned assumptions in place, the probability of sequential errors is reduced to a multiplication of their individual probabilities. This renders the probability of the event of four packets arriving simultaneously due to jitter while each carrying 4 missing packets, which was examined in Chapter 4.2.4, to be $\frac{1}{8} * 10^{-12}$. That said, this is meant to serve as a rough estimation of the potential error likelihood with the intention of providing some intuitive feeling on the subject, rather than an in-depth analysis. As previously mentioned, in a typical scenario these numbers are improved by a factor of a million. Finally, the above analysis corresponds to a single fibre. Should more fibres be present in the network, the combined error rate is multiplied by their respective amount. In summary, errors in the system are improbable but not impossible, and the same is true for multiple consecutive instances.

4.8. Optimisations

By this point, the entire functionality of the original approach to the Switch architecture should be entirely unambiguous. Although the processing pipeline, which was acutely detailed in the previous sections, remained consistent until the completion of the project, some further optimisations were made before arriving at the ultimate version of the proposed design. The final version of the Switch architecture can be examined in Figure 4.36. The most immediately apparent difference to the original approach, as seen in Figure 4.1, is the absence of the time-division multiplexer. While the multiplexer served as an excellent initial step towards achieving the intended functionality of the Switch, it was ultimately concluded that there was a more efficient approach to forwarding pixel groups. The purpose of the multiplexer and the accompanying delay line was to provide each downscaler with 4 distinct opportunities to fetch the data from the processing pipelines. This enabled the customised multicasting ability, i.e. providing each steam the option to map to any scaler up to 4 times. The idea behind this implementation was that by serving an equal time-slice to each pipeline it was possible to derive a fair, predictable behaviour from the otherwise asynchronous streams. However, given that the PIX4 blocks were not synchronous with the time-division multiplexer, the only way to ascertain the correct sampling of the data was to keep each output steady for at least 4 clocks, as was showcased in Section 4.6.

This 4-cycle stabilisation of the output data from the PIX4 blocks now meant that the scalers had 4 distinct opportunities to fetch them by default. This lead to time-division multiplexer acting like a simple synchroniser to compensate for the inherent asynchronous nature of the streams. Thus, it was decided to incorporate this functionality directly into the PIX4 block and eliminate the multiplexer entirely. This was achieved by forcing the aforementioned blocks to have a synchronous IDLE state, leading to the synchronisation of all streams. Once again, if the number of pixels on a given stream is not a multiple of 4, the data is padded and the excess information is stripped off in the downscaler. Granted this change did also force the following stream selection algorithm to adjust, it can be argued that the revised algorithm is a simplification rather than a complication. This final optimisation additionally led to the overall reduction of resource usage and design complexity of the Switch module. Finally, the 4 bits of control data referred to in Figure 4.36 correspond to the 2-bit `stream_id`, the `tvalid` (used to distinguish idle from valid data) and the `vsync`. This combination of control signals additionally allows for an artificial "frame abort" signal to be encoded as `vsync`='1' with `tvalid`='0'.

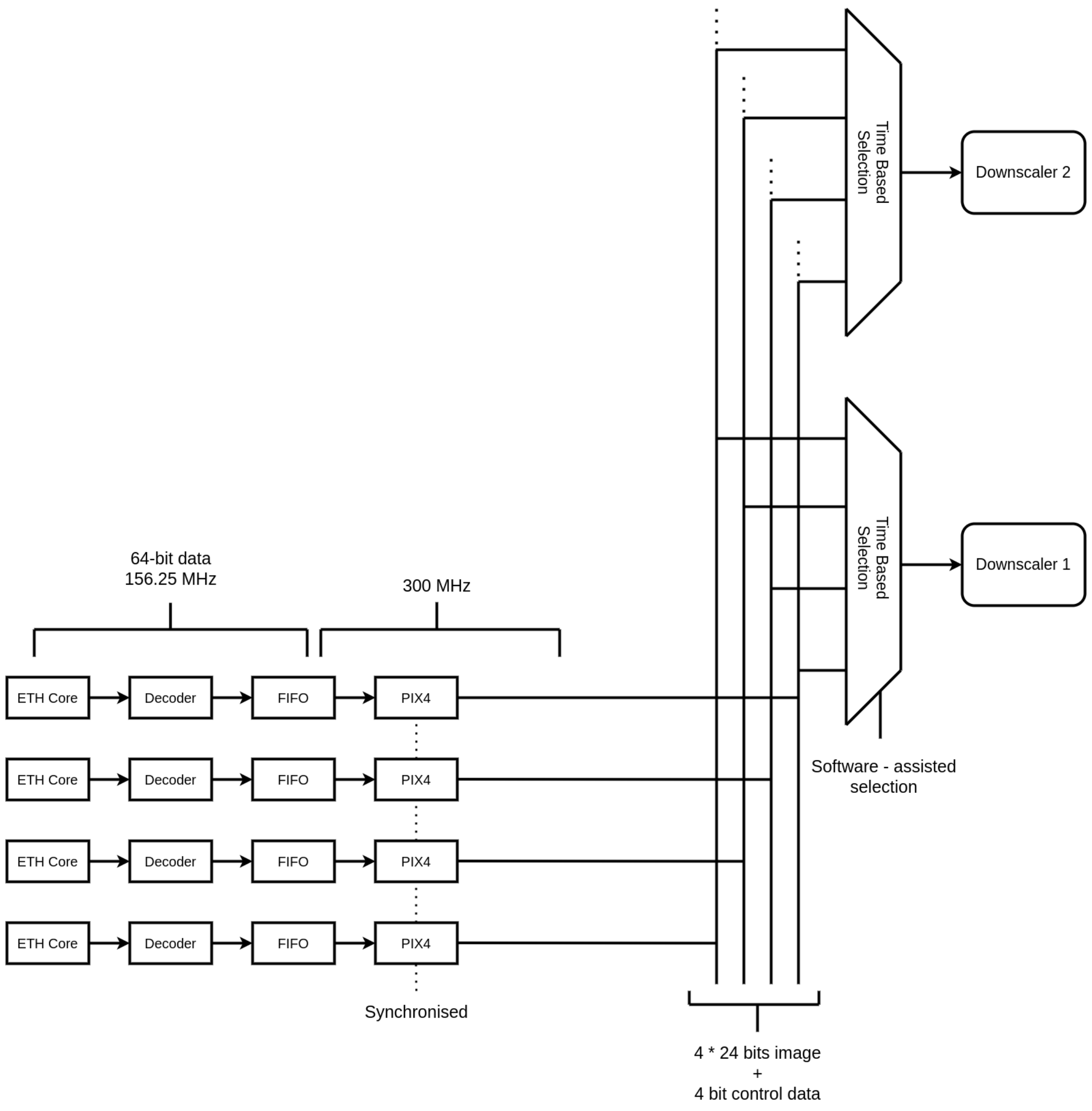


Figure 4.36: Final architecture of the Switch

5 Results and analysis

This chapter presents and further analyses the results ultimately obtained by the final architecture of the Switch. The presented statistics were obtained on a Xilinx ZYNQ Ultrascale+ FPGA device, with a part identifier of xczu7ev-ffvc1156-2-e. The software tool used for the simulation, synthesis and implementation of the module was Vivado version 2019.1.3. Due to the lack of physical availability of the FPGA board itself, all measurements and subsequent reports were produced through Vivado itself. This includes quantifiable metrics on resources, area, power and timing behaviour, whereas the performance of the module was investigated manually.

5.1. Synthesis

Figures 5.1 and 5.2 present the generated netlist, produced as a result of synthesis through Vivado. In both figures, the highlighted datapath corresponds to the processing pipeline, as previously illustrated in Figure 4.3, with every box mirroring a different component. The section previous to the highlighted part in Figure 5.1 is necessary for the correct functionality of the emulated Ethernet core, presented in Figure 4.22, and will not be a part of the design once integrated with the compositor engine. While the synthesis tool does provide outputs for resource utilisation, timing and energy efficiency, they will not be presented here. This decision was taken due to the fact that these reports are provided through the sole input of the netlist and some preliminary constraints, rendering them more of "educated guesses" with not actual FPGA devices involved. Instead, the corresponding and more accurate post-implementation reports will be presented in the sections below. The presented netlist was subsequently provided as input to the tool along with the desired device part in order to attain an actual hardware implementation of the Switch module on the FPGA fabric.

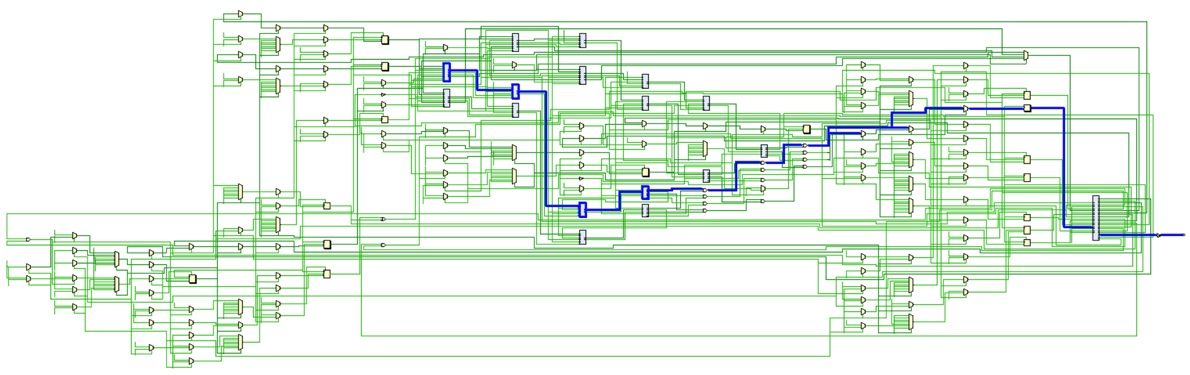


Figure 5.1: Switch netlist with the data processing pipeline highlighted

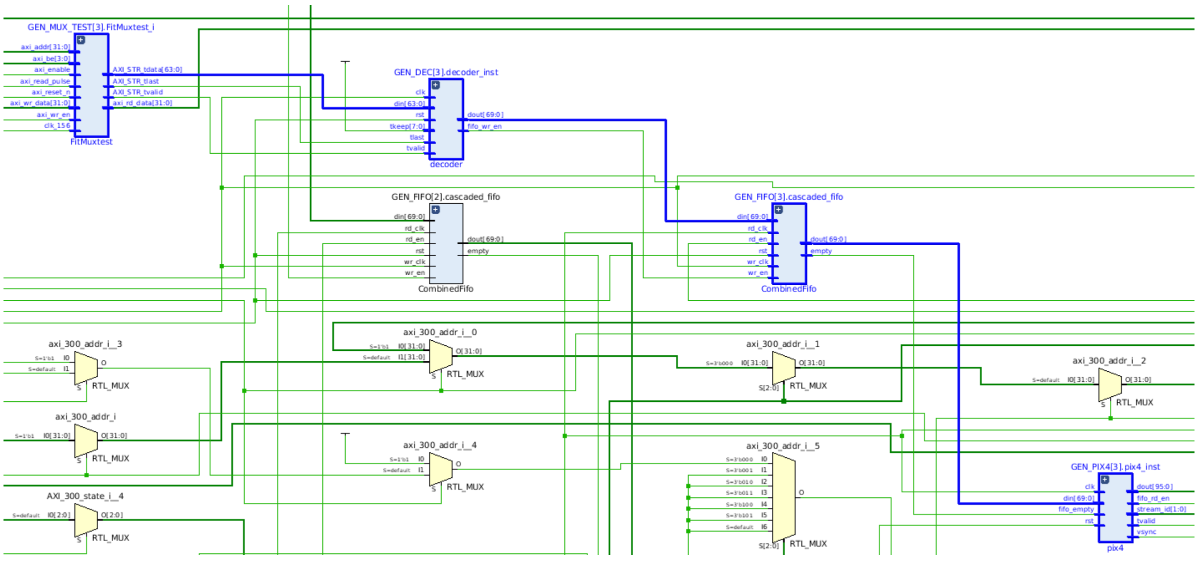
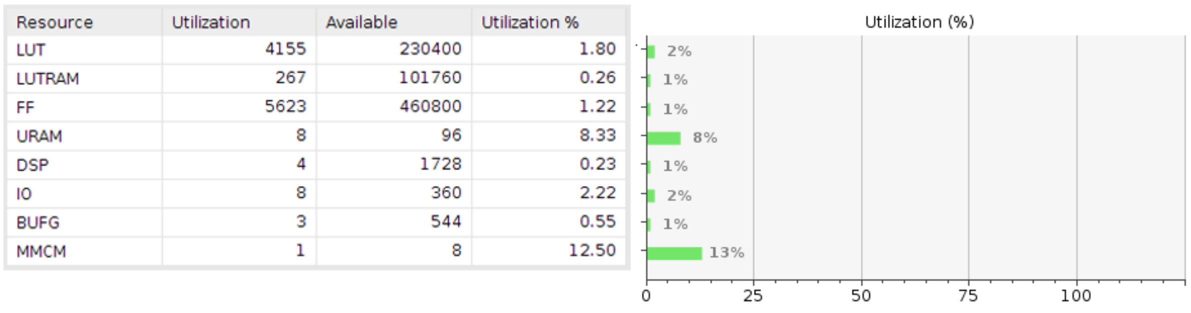


Figure 5.2: Switch netlist of the data processing pipeline

5.2. Resource usage

Resource utilisation and resource reduction constitute the main driving directives of the entire FitOptiVis project. The same concept applies to the design of the switch module, especially memory elements are concerned, a scarce resource on the FPGA. As such, this work is very critical on resource consumption. Figure 5.3 showcases the produced hardware implementation of the module after the netlist is subjected to the place-and-route procedure, whereas Figure 5.4 presents the summary of resource utilisation after said procedure. However, this summary includes resources consumed by the emulated Ethernet cores and the PLL used for producing the driving clocks, both of which are only present to ascertain timing closure and do not constitute part of the Switch's design. Even with those artificial additions, however, it can be seen that design uses very little resources, with logic utilisation remaining below 2%.



Resource	Utilization	Available	Utilization %
LUT	4155	230400	1.80
LUTRAM	267	101760	0.26
FF	5623	460800	1.22
URAM	8	96	8.33
DSP	4	1728	0.23
IO	8	360	2.22
BUFG	3	544	0.55
MMCM	1	8	12.50

Figure 5.4: Design resource summary

Figure 5.5 depicts the elaborated resource utilisation. From this figure we can derive the exact resources allocated to each structural component of the Switch. In terms of logic, each decoder instantiation takes up 167 LUTs, each FIFO 233 LUTs (40 of which are reserved for memory) and PIX4 blocks average at 250 LUTs. The only memory resources utilised are those of the synchronous part of the FIFO, with each buffer reserving 2 URAM blocks. This comes in contrast with what was described in Section 4.5, where it was argued that a single URAM block was enough to accommodate the vast majority of possible cases. Since the external dedicated error handler has not been implemented at the time or writing, this decision to include an extra block of memory was made in favour of an all-encompassing design that incorporated every corner case. Additionally, this pessimistic allocation provides a worst-case scenario on the overall resource utilisation of the project, allowing for improvements at an if-need-be basis. Even through this approach, only 8% of the available URAMs was used and none of the BRAMs, crossing an excellent achievement. Table 5.1 lists the estimated memory

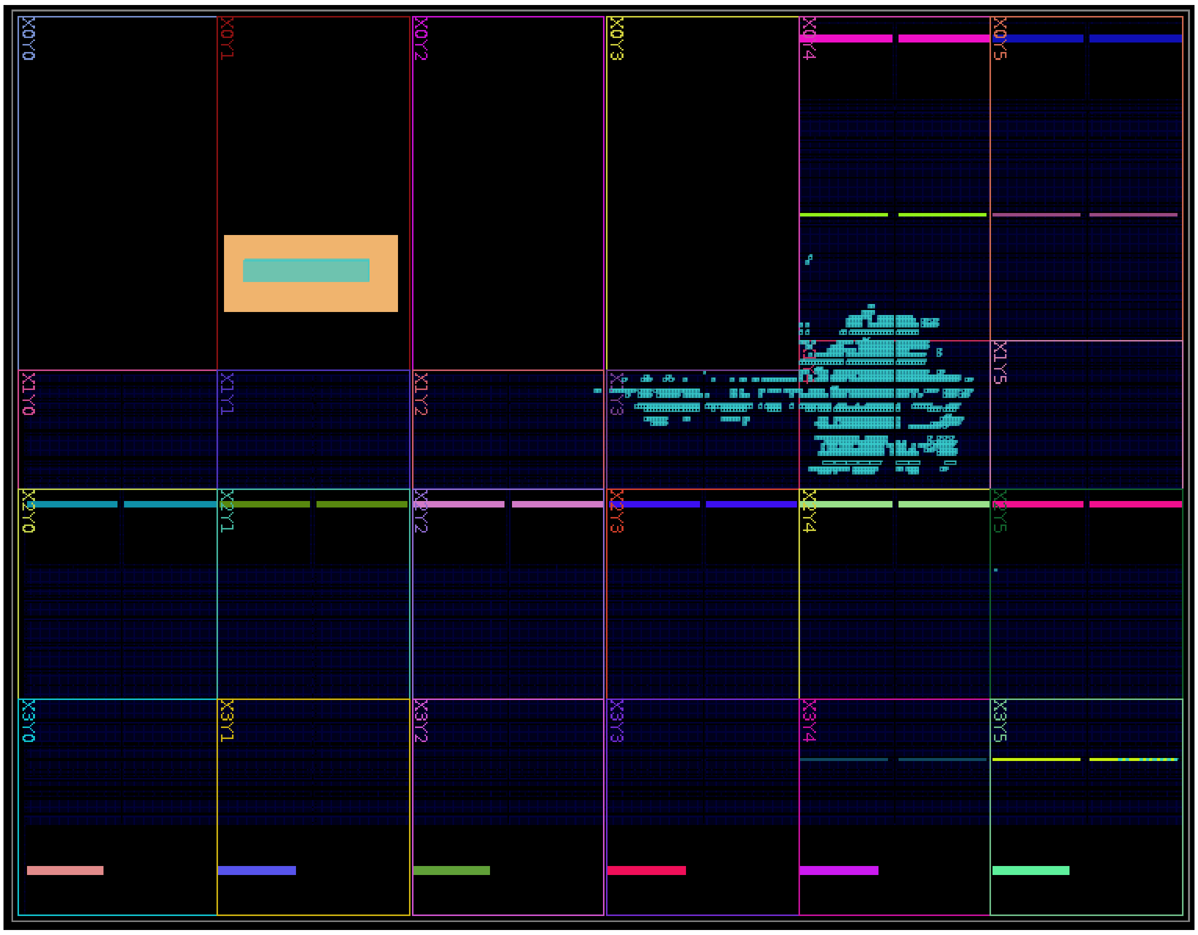
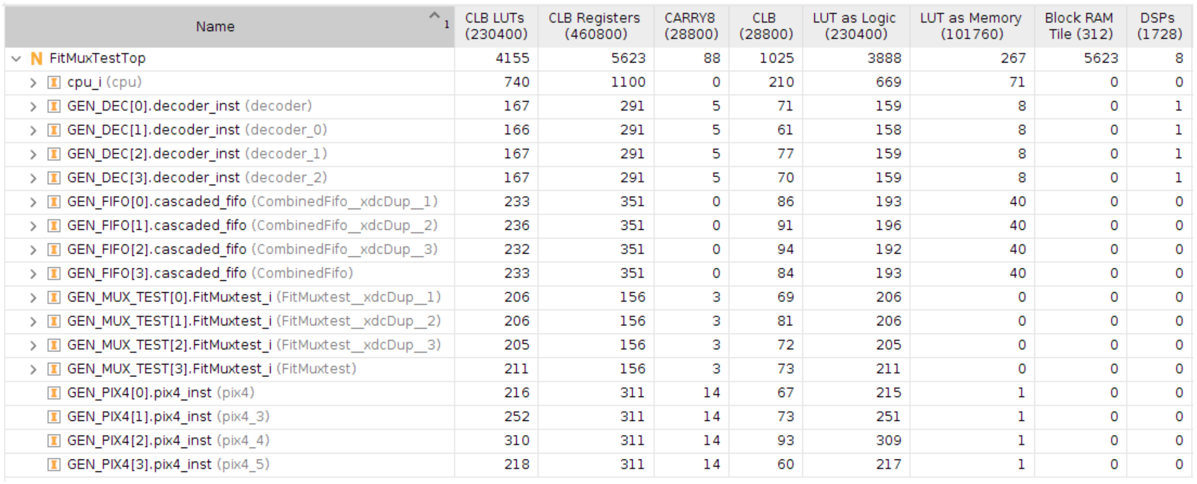


Figure 5.3: Design implementation on the FPGA device

resources allocated to the different components of the compositor as of the time of writing. As can be seen, the Switch accounts for a very small portion of the overall design resource allocation, while the opposite was estimated during its conception. This eliminates a major obstacle in the systems resource management problem and possibly allows the entire design to be ported into even smaller FPGAs.



Name	CLB LUTs (230400)	CLB Registers (460800)	CARRY8 (28800)	CLB (28800)	LUT as Logic (230400)	LUT as Memory (101760)	Block RAM Tile (312)	DSPs (1728)
FitMuxTestTop	4155	5623	88	1025	3888	267	5623	8
cpu_i (cpu)	740	1100	0	210	669	71	0	0
GEN_DEC[0].decoder_inst (decoder)	167	291	5	71	159	8	0	1
GEN_DEC[1].decoder_inst (decoder_0)	166	291	5	61	158	8	0	1
GEN_DEC[2].decoder_inst (decoder_1)	167	291	5	77	159	8	0	1
GEN_DEC[3].decoder_inst (decoder_2)	167	291	5	70	159	8	0	1
GEN_FIFO[0].cascaded_fifo (CombinedFifo__xdcDup__1)	233	351	0	86	193	40	0	0
GEN_FIFO[1].cascaded_fifo (CombinedFifo__xdcDup__2)	236	351	0	91	196	40	0	0
GEN_FIFO[2].cascaded_fifo (CombinedFifo__xdcDup__3)	232	351	0	94	192	40	0	0
GEN_FIFO[3].cascaded_fifo (CombinedFifo)	233	351	0	84	193	40	0	0
GEN_MUX_TEST[0].FitMuxtest_i (FitMuxtest__xdcDup__1)	206	156	3	69	206	0	0	0
GEN_MUX_TEST[1].FitMuxtest_i (FitMuxtest__xdcDup__2)	206	156	3	81	206	0	0	0
GEN_MUX_TEST[2].FitMuxtest_i (FitMuxtest__xdcDup__3)	205	156	3	72	205	0	0	0
GEN_MUX_TEST[3].FitMuxtest_i (FitMuxtest)	211	156	3	73	211	0	0	0
GEN_PIX4[0].pix4_inst (pix4)	216	311	14	67	215	1	0	0
GEN_PIX4[1].pix4_inst (pix4_3)	252	311	14	73	251	1	0	0
GEN_PIX4[2].pix4_inst (pix4_4)	310	311	14	93	309	1	0	0
GEN_PIX4[3].pix4_inst (pix4_5)	218	311	14	60	217	1	0	0

Figure 5.5: Design resource usage elaboration

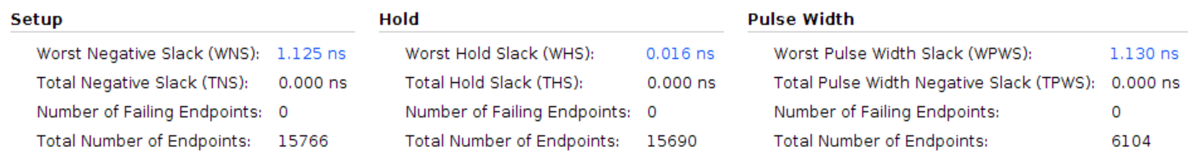
Memory Resource Utilisation	BRAM	URAM
The Switch (4 pipeline instantiations)	0	8
Downscaler (7 instantiations)	42	35
DRAM interface	8	0
Upscaler (4 instantiations)	76	8
Total	126 (40%)	51 (53%)

Table 5.1: Overall memory resource utilisation of the compositor

5.3. Scalability

As stated during the introduction in Chapter 1, scalability was a main concern for the current design. While the switch had to follow some initial requirements, it needed to be expansible as to accommodate for more inputs, more scalers and an increased variety of image resolutions. This meant that the design would have to inevitably scale upwards to incorporate the additional functionality, which comes at the cost of increased hardware resources. The resulting Switch design, nevertheless, manages to be extremely scalable at a minimal cost. To begin with, the final design can provide support for virtually any number of downscalers at no additional hardware cost. Furthermore, it removes the original requirements of 4 different scaling factors and 8 replications per stream, allowing for an unrestricted amount of both when given enough scalers. Added hardware resources are only required if the amount of input Ethernet fibres are increased, in which case the scaling is almost linear, which means that twice the amount of fibres would result in twice the amount of logic. In essence every additional input fibre necessitates its own processing pipeline, which translates to 650 LUTs and 2 URAMs according to Figure 5.5.

5.4. Timing



Setup

Worst Negative Slack (WNS):	1.125 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	15766

Hold

Worst Hold Slack (WHS):	0.016 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	15690

Pulse Width

Worst Pulse Width Slack (WPWS):	1.130 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	6104

Figure 5.6: Design timing summary

Figure 5.6 depicts the post-implementation design timing summary as provided by Vivado. As can be seen, the design manages to meet all timing constraints, achieving a Worst Negative Slack (WNS) time of 1.125 ns. It needs to be noted that this timing report corresponds to the implementation of the Switch alone on the FPGA and does not account for the rest of the compositor. Being the sole implemented module allows the tool to optimally place it on the FPGA fabric and minimise the required wiring, especially from and to URAM blocks which exist near the centre of the device. Integrating the rest of the compositor engine might, and most probably will, significantly alter the Switch's placement which will in some way reflect on its timing behaviour. Nevertheless, with the design exhibiting more than 33% of the timing requirement as negative slack, it is not expected to be a bottleneck in the overall timing closure of the system.

The end-to-end latency of the Switch can be calculated per contained component as follows:

- **Decoder** - The Decoder imposes a strict, deterministic latency of 2 clock periods which are used for data synchronisation. Being in the 156.25 MHz domain this translates to 12.8 ns.

- **FIFO** - The FIFO, as detailed previously, is comprised of two individual components, each inducing the design with a different latency. The synchronous part features a set latency of 4 156.25 MHz clock periods or 25.6 ns. The asynchronous component, on the other hand, is less deterministic. It exhibits a constant latency of of 10 clock periods from the 300 MHz domain which is introduced due to the synchroniser flip-flop pipeline at its output which is present to account for potential metastability issues that might arise from the asynchronous behaviour. In addition to that, a 0-6 ns variable delay is induced from the potential phase difference between its two driving clocks. Overall the FIFO shows a latency of 33.34 - 39.34 ns.

- **PIX4** - The PIX4 block operates within the 300 MHz domain of the FPGA. Depending on its operational cycle, shown in Section 4.6, it can feature a latency of 1-3 clock cycles or 3.334 - 10 ns. There is a single exception for when the block begins operation from the IDLE state, in which case the delay is increased to 16.67 ns.

After traversing through the processing pipeline, the grouped data are simply held steady for 4 clock cycles to ensure safe sampling. This translates to at most 3 extra cycles, or 10 ns, until the data reach a downscaler. As such, the total worst-case end-to-end latency of the Switch amounts to 104.41 ns, while the average case is approximately 88.08 ns. From the reported times, about 40% is attributed to the synchronisation mechanism. However, even during the worst-case scenario, these figures are negligible, accounting for merely 0.01% of the overall compositor engine's latency budget.

The numbers reported above only hold true in the case of an empty FIFO, however. An extra delay is added for every queued 64-bit element in the FIFO at the time of a new inputs arrival. The time associated with this delay is related to how quickly the FIFO is drained, which corresponds to the PIX4 processing rate. As such, an average of 13.334 ns of end-to-end latency is added per 96 bits of data (4 pixels) in the FIFO at the time that the input arrives, rounded up. Figure 5.7 illustrates how the average latency increases in response to the FIFO occupancy. Figure 5.8 showcases the degree of this additional latency for a FIFO consisting of a single URAM block, featuring a maximum capacity of 286720 bits. As can be seen, the added latency exhibits a near linear increase and attains substantial values for high percentages of buffer occupancy, reaching values as high as 40 us.

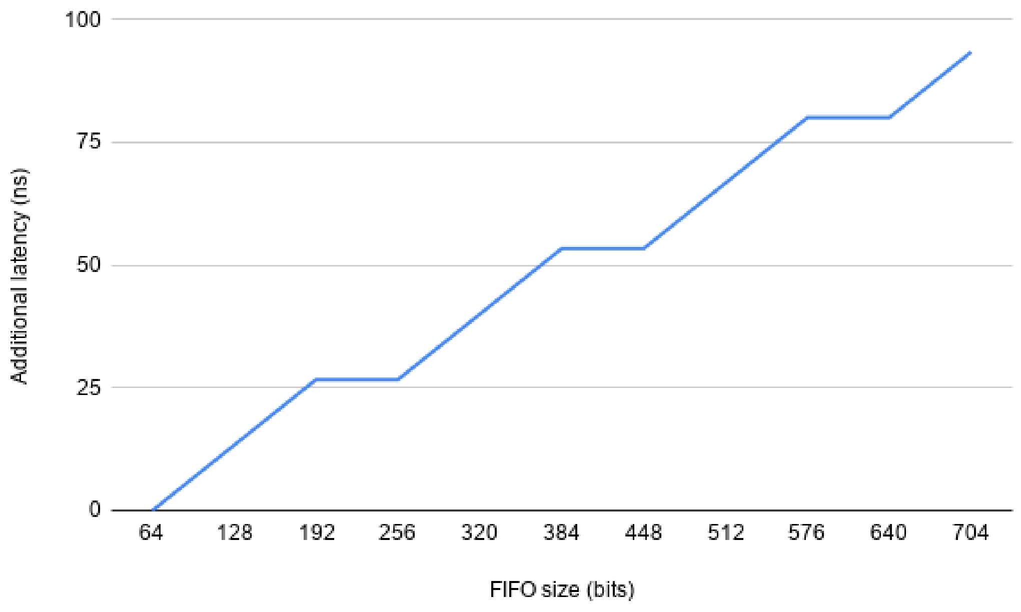


Figure 5.7: Latency vs FIFO contents for small data sizes

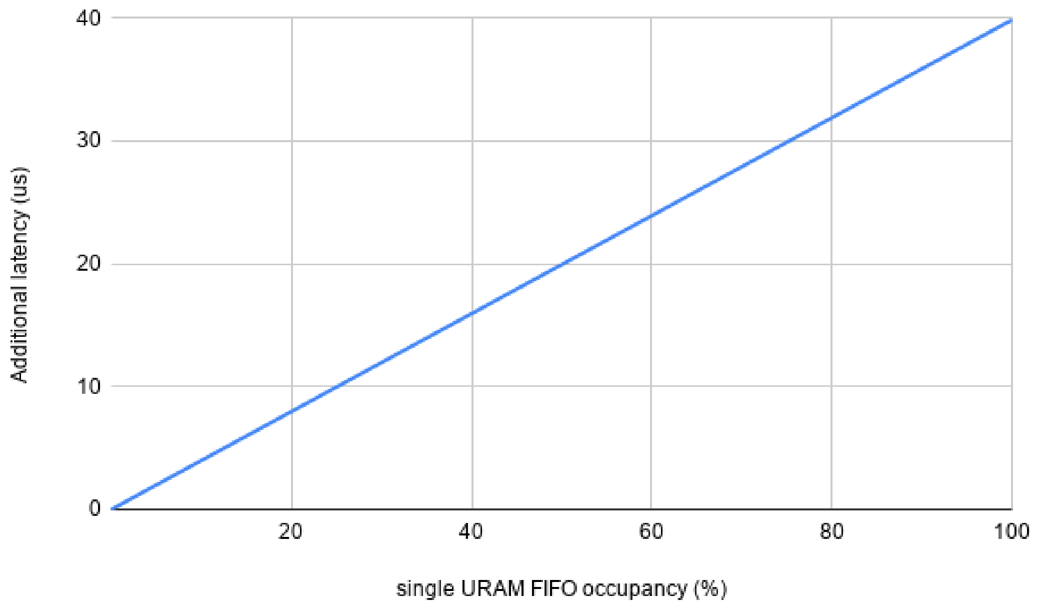
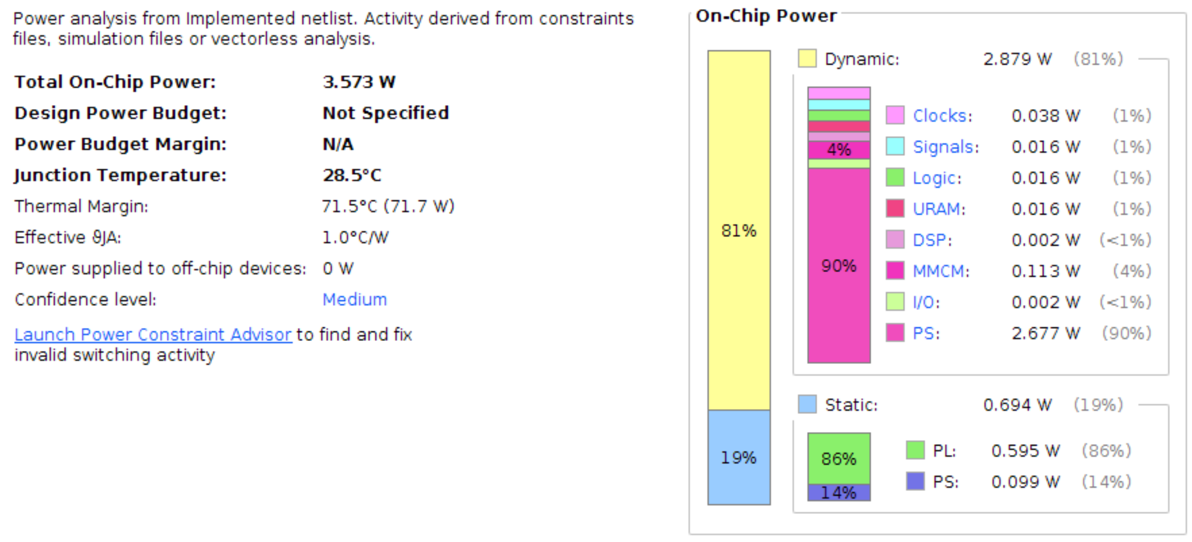


Figure 5.8: Latency vs FIFO occupancy for big data sizes

5.5. Energy efficiency

Figure 5.9 showcases the resulting power analysis from the implemented Switch module. As previously, this includes the extra hardware needed to emulate the Ethernet cores. It should be noted that this is only a rough estimation and should be viewed as an approximate value. In order for the assisting software to provide an accurate report, a large number of constraints need to be clarified, which has not been the case here. Even then, the only precise way to measure the actual power efficiency of the module would have been on the FPGA device itself, which given the unfortunate circumstances, was unavailable for close inspection. That said, even with a very rough estimation, the power efficiency is incomparable with that of a CPU or GPU based solution, remaining well below 60W in the total system when compared to an average of 300W needed for the alternative.


Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**3.573 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**28.5°C**
Thermal Margin:	71.5°C (71.7 W)
Effective ϑJA:	1.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	2.879 W	(81%)
Clocks:	0.038 W	(1%)
Signals:	0.016 W	(1%)
Logic:	0.016 W	(1%)
URAM:	0.016 W	(1%)
DSP:	0.002 W	(<1%)
MMCM:	0.113 W	(4%)
I/O:	0.002 W	(<1%)
PS:	2.677 W	(90%)
Static:	0.694 W	(19%)
PL:	0.595 W	(86%)
PS:	0.099 W	(14%)

Figure 5.9: Design power summary

5.6. Discussion

The most important achievement of the design was the elimination of any but a single buffer, which was the cornerstone of the current project. This versatile buffering scheme allowed for the simplification of the Switch hardware when compared to other solutions like the crossbar, discussed in Section 3.3, which would require one FIFO per input and output port, resulting in a tremendous waste of memory resources. Additionally, those buffers would each have to accommodate for at least a single line of a 4K stream at minimum, since there is no way of knowing the mapping of the streams beforehand. It is apparent that the classic memory management of a network switch would not perform efficiently in the current case, potentially leading to a reduction in message sizes. In stark contrast, this design's single self-adapting FIFO can automatically scale and adjust to any combination of incoming input load, whether it be a single 4K or multiple 720p streams, on the same hardware with no waste of memory. Moreover, the single buffer paradigm allowed for the massive scalability featured in the Switch module as well as the elimination of every other potential bottleneck which led to a simple design, granting the ability to be adequately analysed through a simulation model, as was showcased in Section 4.2.

6

Conclusions and future work

6.1. Conclusions

Initially, the concept of a network switch type hardware module was seen as a possible risk factor to the overall design of the compositor engine. It was estimated that such a module would require significant memory and logic resources allocated to it in order to support the complex required functionality, which was further supported when existing commercial solutions were investigated. The goal of the current project investigation was to examine possible reductions in both the established memory management schemes as well as the complexity of existing switching paradigms. The primary development directions did not yield satisfactory results either, instead proving too expensive, too slow or too energy consuming, further solidifying the pessimistic belief of the original estimation. Furthermore, the custom multicasting ability remained a consistent problem throughout the exploration phase, as it excessively complex and, for the greatest part, not supported by existing architectures. However, the ambitious hardware architecture showcased in Section 4.1, managed to ultimately pave the way towards providing a feasible solution to the proposed problem.

In conclusion, the developed Switch module did not only efficiently meet every requirement stated in Section 1.2.2 and thoroughly answer every question posed in Section 1.2.3, but even managed to exceed many of them. The hardware implementation ultimately does adhere to the required switching functionality, as showcased in Figure 4.4, supporting any possible mapping configuration, including the custom multicasting capability. On that note, the module far exceeded the original requirement of 4 replications on screen per stream, instead allowing for an unconstrained amount. Additionally, the overall latency added to the compositor pipeline was merely 0.01% (100 ns) of the total allocated budget on the average case, increasing to 4% (40 us) during the worst-case scenario, as seen in Section 5.4 . This excessively fast switching times additionally include the frequency domain crossing between the Ethernet fibre inputs and the FPGA device while introducing no disturbance to the remaining processing pipeline and imposing minimal buffering of data. Furthermore, as previously discussed in Section 5.2, the project came to fruition under an astonishingly low amount of resource utilisation, only requiring a single URAM block per input fibre and a marginal amount of LUTs to be allocated in the FPGA under normal circumstances. This elegant, single buffer paradigm enabled the near-linear scalability that the Switch ultimately exhibits, allowing for both possible portability to smaller FPGA devices as well as massive expansibility of the design. More importantly, these achieved functionalities come at virtually no additional cost since the design was chosen to be developed directly integrated with the rest of the compositor on the FPGA fabric, rendering it the most inexpensive of the examined solutions by a tremendous margin.

6.2. Future work

While the final developed architecture more than sufficiently satisfies the original set of requirements and constraints imposed to the project, it is by no means a perfected solution. On the contrary, there are several areas which could be improved, further pushing the boundaries of the state of the art in high frequency switching. A non exhaustive summary of such proposed improvements is discussed in this section, in hopes of providing some possible direction towards the future work of this project.

To begin with, in order for the Switch to be equipped to handle 4K image resolution streams, there needs to be a decoupling of the line boundary from the packet boundary. As it stands, the Ethernet Jumbo Frame can only support payloads of up to 9000 bytes, making it impossible to accommodate a full line of 3840 pixels which corresponds to 11520 bytes. This means that the line has to be split and transported between two different packets, which in turn allows room for the interleaving of packets from other streams belonging to the same fibre in between the aforementioned line fragments. Since, currently, the sole indicator of termination is the `tlast` signal which denotes both a line and a packet boundary, there is no supported way to differentiate between the two. As such, a line separation and identification mechanism would need to be implemented, possibly involving a sort of `tlast_line` signal.

Secondly, as previously mentioned in Section 4.7, the current implementation by which the module handles subsequent events of missing packets is rather naive, allowing for multiple occurrences of its threshold case. Thus, it is proposed that this mechanism be geared towards a more streamlined approach, instead having the FIFO drop incoming frames depending on its current capacity rather than asserting a set threshold of missing packets. This would allow for a better, self-adjusting buffer control that is additionally more in line with the rest of the design choices taken during the rest of this work.

Furthermore, on the topic of error handling, the extreme worst-case scenario examined in Section 4.2.4 currently poses as a bottleneck towards the memory utilisation of the design. As was described previously, in order to account for this single, highly unlikely case, a second URAM block is needed per input fibre, essentially doubling the resource utilisation of the entire design. If chosen to be ignored, however, its occurrence could lead to severe failure of the entire compositor engine. It is, thus, proposed that this case be identified and handled separately through software assistance and omitting the second URAM block, vastly reducing the already low resource utilisation as well as the average FIFO occupancy. The case needs to be simply identified at the input, signalling the rest of the corresponding frame to be skipped until the next `vsync`.

In addition to the above, the current throughput bottleneck of the Switch architecture is the PIX4 block operating at 7.2 Gb. This can be increased in two possible ways, however, both attached with their own distinct setbacks and challenges. On the one hand, the block could technically be replaced by a "PIX8" block. Namely, the block could be altered to bundle groups of 8 pixels which can effectively be provided every 6 clock cycles. The effective bandwidth of the block would then increase to 9.6 Gb, which is also the theoretical maximum assuming no overclocking. This is possible due to the downscalers operating in strokes of 128 pixels, which is divisible by 8. On the downside, this solution would require some modification at the input of the downscalers which includes additional memory resources. On the other hand, the same maximum throughput can also be achieved through the PIX4 block providing groups of 4 pixels every 3 clocks, kept stable for 3 clock cycles. Granted, this modification does violate one of the original requirements, restricting the mapping of any given stream to 3 times to a single downscaler instead of 4.

Finally, if the design needs to migrate to 100 Gb Ethernet inputs, the system would be unable to handle the input load through pure widening. As such, for this particular case, it would be recommended to investigate the creation of artificial channels inside the FPGA fabric and proceed to a static allocation. It is currently theorised that such a solution would essentially lead to a version very similar to the current design, only distributed.

Bibliography

[1] *Royal philips,* https://www.philips.nl/, accessed: 2021-01-06.

[2] Z. Al-Ars, T. Basten, A. de Beer, M. Geilen, D. Goswami, P. Jääskeläinen, J. Kadlec, M. M. de Alejandro, F. Palumbo, G. Peeren, L. Pomante, F. van der Linden, J. Saarinen, T. Säntti, C. Sau, and M. K. Zedda, *The fitoptivis ecsel project: Highly efficient distributed embedded image/video processing in cyber-physical systems,* in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19 (Association for Computing Machinery, New York, NY, USA, 2019) p. 333–338.

[3] *The basic structure of fpgas,* https://ckyrkou.medium.com/what-are-fpgas-c9121ac2a7ae, accessed: 2020-12-25.

[4] *Xilinx fpga overview,* https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html, accessed: 2020-12-25.

[5] S. Mittal, *A survey of fpga-based accelerators for convolutional neural networks,* Neural Computing and Applications (2020), 10.1007/s00521-018-3761-1.

[6] P. A. Simpson, *FPGA Design, Best Practices for Team Based Reuse, 2nd edition* (Switzerland: Springer International Publishing AG, 2015) iSBN: 978-3-319-17924-7.

[7] *Fpga basics: Architecture, applications and uses,* https://www.arrow.com/en/research-and-events/articles/fpga-basics-architecture-applications-and-uses (), accessed: 2020-12-25.

[8] J. Hoozemans, R. Heij, J. van Straten, and Z. Al-Ars, *Vliw-based fpga computation fabric with streaming memory hierarchy for medical imaging applications,* in *Applied Reconfigurable Computing (ARC)* (2017).

[9] G. Smaragdos, G. Chatzikonstantis, R. Kukreja, H. Sidiropoulos, D. Rodopoulos, I. Sourdis, Z. Al-Ars, C. Kachris, D. Soudris, C. I. D. Zeeuw, and C. Strydis, *BrainFrame: a node-level heterogeneous accelerator platform for neuron simulations,* Journal of Neural Engineering **14** (2017), 10.1088/1741-2552/aa7fc5.

[10] E. Houtgast, V. Sima, and Z. Al-Ars, *High performance streaming smith-waterman implementation with implicit synchronization on intel fpga using opencl,* in *IEEE International Conference on Bioinformatics and Bioengineering (BIBE)* (2017) pp. 492–496.

[11] P. Bhosale, M. Staring, Z. Al-Ars, and F. F. Berendsen, *GPU-based stochastic-gradient optimization for non-rigid medical image registration in time-critical applications,* in *Medical Imaging 2018: Image Processing*, Vol. 10574, edited by E. D. Angelini and B. A. Landman, International Society for Optics and Photonics (SPIE, 2018) pp. 185 – 191.

[12] N. Ahmed, J. Lévy, S. Ren, H. Mushtaq, K. Bertels, and Z. Al-Ars, *Gasal2: a gpu accelerated sequence alignment library for high-throughput ngs data,* in *BMC Bioinformatics*, Vol. 20 (2019).

[13] S. Ren, N. Ahmed, K. Bertels, and Z. Al-Ars, *Gpu accelerated sequence alignment with traceback for gatk haplotypecaller,* in *BMC Genomics*, Vol. 20 (2019).

[14] *Why use an fpga instead of a cpu or gpu?* https://blog.esciencecenter.nl/why-use-an-fpga-instead-of-a-cpu-or-gpu-b234cd4f309c (), accessed: 2020-12-25.

[15] *Fpga advantages and common applications,* https://hardwarebee.com/fpga-advantages-common-applications-today/ (), accessed: 2021-01-06.

[16] *Publications and presentations,* https://web.archive.org/web/20100821182813/http://www.bdti.com/articles/info_eet0207fpga.html (), accessed: 2020-12-25.

[17] M. LaPedus, *Xilinx aims 65-nm fpgas at dsp applications,* EETimes., accessed: 2020-12-25.

[18] *Fpga implementation step-by-step,* https://digitalsystemdesign.in/fpga-implementation-step-by-step/, accessed: 2021-01-06.

[19] *A gentle introduction to hardware accelerated data processing,* https://hackernoon.com/a-gentle-introduction-to-hardware-accelerated-data-processing-81ac79c2105, accessed: 2020-12-27.

[20] *Hardware acceleration,* https://www.omnisci.com/technical-glossary/hardware-acceleration (), accessed: 2020-12-27.

[21] *Network switch,* https://en.wikipedia.org/wiki/Network_switch, accessed: 2021-01-06.

[22] T. Sridhar, *Layer 2 and layer 3 switch evolution,* The Internet Protocol Journal (1998).

[23] *What is a network switch, and how does it work?* https://www.networkworld.com/article/3584876/what-is-a-network-switch-and-how-does-it-work.html, accessed: 2021-01-06.

[24] *Switch overview,* https://www.grotto-networking.com/BBSwitchArch.html (), accessed: 2020-12-27.

[25] *What's inside a router,* https://www.net.t-labs.tu-berlin.de/teaching/computer_networking/04.06.html (), accessed: 2020-12-27.

[26] *Noc features, processors of the future,* https://itigic.com/network-on-a-chip-features-processors-of-future/, accessed: 2021-01-06.

[27] *Introduction to fpga acceleration,* https://www.stemmer-imaging.com/en/technical-tips/introduction-to-fpga-acceleration/, accessed: 2021-01-06.

[28] *Fpga vs gpu for machine learning applications,* https://www.aldec.com/en/company/blog/167--fpgas-vs-gpus-for-machine-learning-applications-which-one-is-better (), accessed: 2020-12-29.

[29] e. a. Cong, Jason, *Understanding performance differences of fpgas and gpus,* Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM (2018), 10.1109/FCCM.2018.00023.

[30] e. a. Ovtcharov, Kalin, *Accelerating deep convolutional neural networks using specialized hardware.* Microsoft Research Whitepaper 2.11 (2015).

[31] e. a. Nurvitadhi, Eriko, *Can fpgas beat gpus in accelerating next-generation deep neural networks?* Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM (2017), 10.1145/3020078.3021740.

[32] C. Murphy and Y. Fu, *Xilinx all programmable devices: A superior platform for compute-intensive systems,* .

[33] *Fpga vs gpu,* https://haltian.com/resource/fpga-vs-gpu/ (), accessed: 2020-12-29.

[34] *Ethernet gigabit jumbo frame,* https://oringnet.com/en-global/tech/detail/15, accessed: 2021-01-06.

[35] G. Kahn, *The semantics of simple language for parallel programming,* Proc. IFIP Congress on Information Processing. North-Holland. (1974).

[36] *'xilinx axi reference guide',* https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf, accessed: 2021-01-06.

[37] *'ultrascale architecture memory resources',* https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf, accessed: 2021-01-06.

[38] *'ultrascale architecture libraries guide',* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug974-vivado-ultrascale-libraries.pdf, accessed: 2021-01-06.