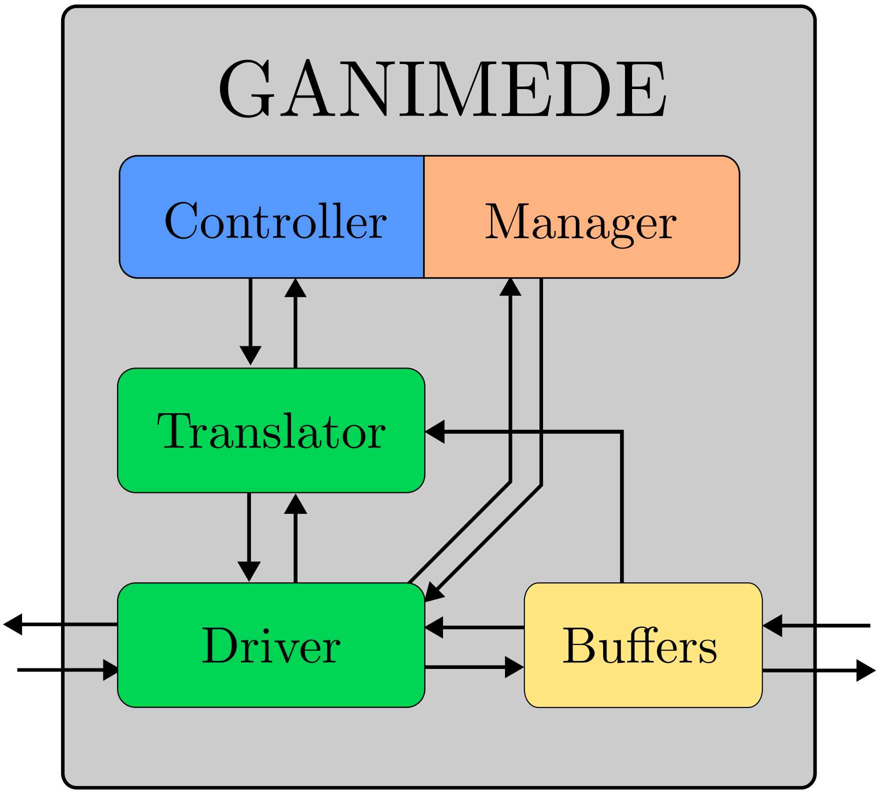
CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG



Optimizing Stream Engines for use in eFPGAs on Radiation Hardened SoCs

Master's thesis in Computer science and engineering

Adam Magnusson
Erik Örtenberg

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

MASTER'S THESIS 2025

Optimizing Stream Engines for use in eFPGAs on Radiation Hardened SoCs

Adam Magnusson
Erik Örtenberg

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Optimizing Stream Engines for use in eFPGAs on Radiation Hardened SoCs

Adam Magnusson

Erik Örtenberg

Supervisor: Ionnis Sourdis, CSE
Co-Supervisor: Magnus Östgren, CSE
Company Supervisor: Martin Rönnbäck, Frontgrade Gaisler
Company Co-Supervisor: Jan Andersson, Frontgrade Gaisler
Examiner: Pedro Petersen Moura Trancoso, CSE

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A block diagram of the proposed topology GANIMEDE developed in this thesis work

Typeset in LaTeX
Gothenburg, Sweden 2025

Optimizing Stream Engines for use in eFPGAs on Radiation Hardened SoCs

Adam Magnusson, Erik Örtenberg
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Systems on a Chip (SoCs) are becoming increasingly common for use in most computational domains as heterogeneous hardware architectures prove themselves very efficient and powerful. The space domain is one such example and poses a plethora of challenging design constraints, which become even more pronounced in the context of radiation hardened embedded Field Programmable Gate Arrays (eFPGAs). eFPGAs lend themselves to supporting powerful hardware accelerators (HAs), where data is streamed in by the use of a stream engine. Due to the relatively small amount of programmable logic in eFPGAs, the stream engine supporting the HA must be made as resource efficient as possible. Though, to the best of the authors knowledge, there is no previous work exploring resource optimized stream engines for use on eFPGA.

In this thesis, we implement a performance and resource efficient stream engine for eFPGAs. The proposed stream engine – named GANIMEDE – achieved a communication link utilization of 94.5% accounted for protocol overheads while only occupying 2.1% of the total available resources provided by the targeted eFPGA. In addition, this thesis offers a discussion on desirable properties in stream engines and presents parts of supported protocols that should be implemented on the SoC instead of the eFPGA.

Keywords: Radiation Hardend SoC, eFPGA, Space Application, Stream Engine, Dataflow Hardware Accelerators, Resource Utilization, Optimization.

Acknowledgements

We would like to thank Jan and Martin at Frontgrade Gaisler for supporting us during the project. Your help is much appreciated. Also at Gaisler, We would like to thank Juan Pedro for providing us with tools and support for getting synthesis up and running.

We would also like to thank our Co-supervisor Magnus for being a valuable and insightful discussion partner throughout the project, while also working on his PhD.

Lastly, our peer reviewers Nor and Erik have been invaluable, providing continuous feedback throughout our work. Your help in providing a fresh perspective on our work has helped us a great deal. Thank you.

Adam Magnusson and Erik Örtenberg, Gothenburg, 2025-08-12

Contents

Word List

ASIC - Application Specific Integrated Circuit. An IC made for a specific application. This covers most all ICs as the only reasonable alternative is to deploy to FPGA. ASICs are fast and space efficient compared to FPGAs but there is a large start-up cost to set up manufacturing of an ASIC design. Thus they are only economically viable if produced in large quantities.

DFF - D flip-flop. A digital storage cell storing 1 bit of data.

DMA - Direct Memory Access. DMA modules allow for communication between memory and other modules on an SoC without micromanagement from the CPU. While this communication is often initiated by the CPU it does not need to waste its time feeding data to other modules, thanks to the DMA structure.

FE - Functional Element. A hardware resource on an FPGA. Mostly used to refer to LUTs, DFFs and digital signal processors.

FIFO - First In First Out. A way to order data inside a data queue. A FIFO is what most would consider a normal queue, where the first to enter the queue will also be the first to exit it.

FPGA - Field Programmable Gate Array. A programmable fabric which can implement any hardware function by the use of lookup tables and crossbar switching logic. Usually RTL code developed in a HDL is used to synthesize a bit stream which is the compiled hardware, ready to be deployed on a target FPGA.

FSM - Finite State Machine. An abstract way to encode a program or algorithm into a series of steps with well defined transitions between each step. The steps are called *states*, and the *state machine* is considered *finite* if the number of states present in the state machine is finite.

HA - Hardware Accelerator. A hardware implementation of a computation made with the purpose of speeding it up. While a CPU is versatile and can perform practically any task, some tasks are done often enough or are demanding enough to warrant a fast and efficient hardware implementation.

HDL - Hardware Description Language. A programming language which describes the behavior of digital and analog circuits. The most famous ones are VHDL and Verilog, which both have an emphasis on digital logic but have extensions for analog circuitry.

IC - Integrated Circuit. A semiconductor chip which incorporates multiple electronic components to produce a desired behavior. The complexity of ICs can vary greatly from simple logic gates to entire processor systems.

IP Core - Intellectual Property Core. A catch all term for licensed reusable hardware modules, both proprietary and open source ones. Closed source IP cores usually have a non-disclosure agreement in their license, while open source usually use one of the free or viral licenses such as MIT or GPL.

LUT - Lookup Table. The backbone of FPGAs. In essence it is a small memory cell where the input signals, often 4 or 6 of them, are used as an index for the output. This means that a 4-input LUT can emulate any binary function taking up to 4 bits of input and yielding 1 bit of output. Put many of them together and any boolean function can be implemented.

RTL - Register Transfer Level. The domain which most HDLs focus on describing. The goal of RTL is to describe combinatorial digital logic and its interaction with sequential hardware features, such as flip flops and latches. Combinatorial and sequential logic together describe the function of most ICs.

SoC - System on a Chip. A type of IC which incorporates different modules such as CPUs, memory and network interfaces on the same chip. Being on the same chip means eliminates a lot of latency since the modules are much closer together than they would be on a circuit board.

1

Introduction

Designing hardware for space applications presents a unique set of challenges including radiation effects, lack of surrounding infrastructure and limited cooling solutions just to name a few. With the rising demand of on board data processing on satellites, the industry turns to hardware accelerators (HAs) to meet these demands while overcoming the aforementioned challenges [1], [2]. As hardware acceleration becomes more prominent in space applications, it incentivizes the inclusion of Field Programmable Gate Arrays (FPGAs) in space missions. This allows for speedups in certain tasks well suited for hardware acceleration and enables replacement of broken mission critical systems which could extend the operational mission time

Traditionally, commercial of the shelf FPGAs have been deployed for applications in low earth orbit where radiation effect do not affect the digital logic too much. Going further into space does, however, require the FPGA to be radiation hardened (rad-hard) in order to operate normally in the face of hazardous radiation. Rad-hard FPGAs use various techniques to resist radiation at the cost of using more area, leaving less resources for the user compared to their non-rad-hard counterparts. As rad-hard FPGAs became more common during the early 2010s, the demand for them in the space industry grew considerably [3], [4]. This demand is being met by recent advances in technology, as Frontgrade Gaisler, Microchip and NanoXplore are developing processing systems with rad-hard FPGAs more closely interwoven with the system architecture with FPGA systems on a chip (SoCs)[5]–[7].

While having a rad-hard FPGA closely connected to the SoC provides a good starting point for HAs in space, one problem still remains. Namely, moving data to and from the accelerator. Usually, this is done by implementing a direct memory access (DMA) controller that handles the communication between the hardware accelerator and a memory subsystem by transferring chunks of data back and forth. However, hardware acceleration is most often implemented as a dataflow structure, which incentivizes DMA implementations where data is streamed in and out of the HA [8]. This behavior can be facilitated by a stream engine, whose purpose it is to manage streams of data between memory and an application [9]. As alluded to in the previous paragraph, implementing a stream engine for use in a rad-hard FPGA must be optimized for resource utilization rather than purely performance. Therefore, balancing the implicit tradeoff between performance and resource utilization while prioritizing reducing resource utilization is of utmost importance.

1.1 Problem Statement

Rad-hard FPGAs and eFPGAs are becoming commonplace in spacecraft hardware, which promotes the use of dataflow oriented HAs [4]. These HAs often perform best when delivered streams of data, rather than discrete parts of memory. To the best of the authors knowledge, no resource optimized stream engine for use in spacecraft SoCs exist which justifies this thesis.

1.2 Thesis Objectives

This paper aims to develop a stream engine, called GANIMEDE, to efficiently communicate data to and from the rad-hard embedded FPGA (eFPGA) contained within FrontGrade Gaisler's GR765 SoC. GANIMEDE will be optimized with regards to the constraints of space-grade hardware and developed using the GR765 as a target. The results presented in this paper will be gathered partly from a case study of the CCSDS hyperspectral image compression standard, and partly from synthesis results. The CCSDS hyperspectral image compression standard will be run as the hardware accelerator SHyLoC composed of the CCSDS 123.0-B-1 and CCSDS 121.0-B-2 [10]. The metrics gathered are the hardware resource utilization, the throughput needed to saturate the targeted IP-core and the maximum unhindered throughput through GANIMEDE. The goal of the paper is to present an optimized stream engine for use in rad-hard space grade hardware, and in the process, provide an evaluation of the GR765 design specification w.r.t. the eFPGA. In addition to achieving the developmental goals stated previously, this project also aims to answer the following questions.

What are desirable properties in a stream engine optimized for spacecraft SoCs? As previously mentioned, a case study on SHyLoC will be performed. From this study it should be possible to determine what kind of functionality is desirable or maybe even necessary for a stream engine to be useful in a resource constrained environment. Therefore a set of properties which keep resource utilization low while providing high throughput must be determined.

Which parts of the implemented protocol should be on eFPGA versus in the SoC? Implementing hardware on the SoC as an application specific integrated circuit (ASIC) will in general yield better performance, less area utilization and less power consumption at the cost of versatility[11] additional FPGA IO pin usage. Different HAs make use of data differently however, which means that it is infeasible to have a general design data management solution in ASIC as some HAs desire streams while others demand memory system access. Thus, at least some communication logic will have to be implemented in FPGA but it is desirable to have as much of the design in ASIC as possible since it saves both power and area while also increasing performance. This raises the question of what parts of the interface are general enough to be on the SoC instead of the eFPGA.

What is required to saturate SHyLoC with data using GANIMEDE? Ideally GANIMEDE would be able to provide data to the SHyLoC IP core as fast as it can be consumed. To determine how well GANIMEDE performs, throughput measurements will be run, whereafter the throughputs of GANIMEDE and an ideal throughput will be compared. The goal is to determine what parameters and version of GANIMEDE is needed to saturate SHyLoC while minimizing resource utilization.

How small can GANIMEDE be made without compromising performance? Since rad-hard FPGAs typically have a relatively limited amount of resources, the design of GANIMEDE should preferably be as small as possible to leave room for HAs and reduce power draw. Thus, efforts will be made to minimize the design while maintaining the highest possible throughput.

1.3 Ethical Considerations

This thesis aims to provide a compact yet efficient stream engine as a contribution to the shift toward on-board computation on extraterrestrial missions. Running computations locally on the spacecraft or probe enables more real time applications to run on these systems. With multiple countries looking into the potential of space as a military domain [12], real time applications in space will likely include the likes of targeting systems for space weaponry. The technological implications of this political development are rather unfortunate. The aim of this thesis is not to contribute to the militarization of space, but rather the exploration and discovery of space as a frontier of science. Therefore, the value of on board computing in cases such as hyperspectral imaging are considered to outweigh the harm present in potential space warfare.

By continuing work on space grade equipment, this work indirectly encourages the continued exploration of other planets. This is a cause for concern, as interplanetary exploration may annihilate potential alien ecosystems or bring back harmful biological materials to Earth. Historically, human exploration has brought with it contamination in forms of disease and invasive species. Consequently, measures were put into place early on in space exploration to minimize the risks of contamination [13]. The developments in planetary protection are still ongoing and the hope is that the lessons learned from historical incidents are enough to put the necessary precautions in place to eliminate this issue.

1.4 Thesis Outline

The remainder of the thesis is organized as follows. Chapter 2 presents background of the field, the context in which GANIMEDE is built as well as some related works. Chapter 3 outlines the limitations of the thesis, that is to say what will not be investigated. Furthermore, Chapter 3 explains the methods to be used for developing and evaluating GANIMEDE. Chapter 4 explains the architecture of GANIMEDEs components as well as improvements to both performance and resource utilization. Chapter 5 goes over the results, some of which provide conclusive answers to the

"How small can GANIMEDE be made without compromising performance?" and "What are desirable properties in a stream engine optimized for spacecraft SoCs?" objectives. Some other insights not relevant to the research questions are also discussed in this Chapter. Chapter 6 provides a discussion about remaining objectives and the work overall, as well as a conclusion.

2

Background and Related Works

This chapter details some background information regarding direct memory access (DMA) structures and how they relate to the thesis work followed by a short description of what field programmable gate arrays (FPGAs) are. After that, some common methods for dealing with radiation effects in digital logic are discussed and how they relate to the thesis case study in the Section regarding the target system GR765. Then a description of the case studys computational workload SHyLoC is given. Following the background information, a section is dedicated to related works in the field and how this work differs from them.

2.1 Direct Memory Access

Moving data to and from computational devices is often a large contributor of execution time in different algorithms due to latency and throughput issues [14]. In traditional CPU design, cache hierarchies are used to mitigate the effects of data transfer latency on execution time by a plethora of techniques [15]. One important aspect is that this way of accessing the memory is implicit rather than explicit in the sense that the entire memory space is assumed to be present in a memory adjacent to the CPU.

The tradeoff for this abstraction is lower latency in exchange for more area on chip spent on caches. This tradeoff makes sense in CPUs, as much of the infrastructure built around modern processing systems and operating systems depend on the cache hierarchy to function efficiently. For separate computational devices, such as HAs, a cache hierarchy is often unnecessarily cumbersome and too resource intensive compared to what is required of the computational load. The alternative to using caches which implicitly fetch data from memory is, unsurprisingly, to fetch data explicitly.

Accessing data explicitly requires a protocol in order to make use of hardware resources efficiently. One way to implement the protocol used to fetch the data is by the use of DMA controllers. DMA controllers are able to receive memory instructions from the CPU and generate memory accesses on communication link the DMA is connected to. The accesses generated by the DMA controller are issued without CPU intervention thereafter. DMA controllers generally move data from one address to another, for example from one address on a hard drive to another address in RAM. This is referred to as a memory-to-memory transaction.

2.1.1 Memory Access Patterns

When accessing hundreds to millions of words in memory, it is usually done in a regular pattern with a distance between each read or written word. The distance between words, also known as stride, is usually measured in word size, i.e. a stride of 1 means that the next word to read is one word size ahead in memory. A stride of 1 is commonly referred to as the unit stride and is the access pattern of, for example for-each loops or equivalent structures in software as seen in Listing1.

A non-unit stride is also common in cases such as matrix operations when fetching a column of the matrix. Since matrices usually are stored row by row in memory, fetching a column out of this data structure requires indexing the memory with a side length stride.

```c
int stride = x;
for (int i = 0; i < n; i += stride) {
    int y = arr[i];
    /*
      Some operations using y
    */
}
```

Listing 1: C code example of array accesses using strides defined by `x`. The stride defines at what intervals values are read from memory. If `x = 1` it is a unit stride. If `x = 2` it is called a stride of 2, and so on.

2.1.2 Stream Engines

Stream engines are a special case of DMA in which the transfers are memory-to-application rather than memory-to-memory. The difference is illustrated in Figure 2.1, where the DMA controller needs 4 bus transactions after receiving a request as opposed to the 2 transactions issued with the stream engine example.

When supporting high performance HAs there are three reasons why stream engines are preferred over DMA controllers. First, high performance HAs tend to need to fetch long continuous data to well defined FIFO ports. This allows for some simplifications of the stream engines control and index logic which is not viable in DMA controllers. Second, since the HAs FIFO ports does not have an address space, there is no need to provide both a read and write address for each instruction, as is needed in a DMA controller. This is what reduces the total number of bus transactions compared to the DMA controller case, as the sender of the first request is also the intended recipient.

These hardware accelerators tend to be fully dataflow driven, with potentially deep pipelines [9]. This is the intended target of GANIMEDE, which is why a stream engine design was chosen.

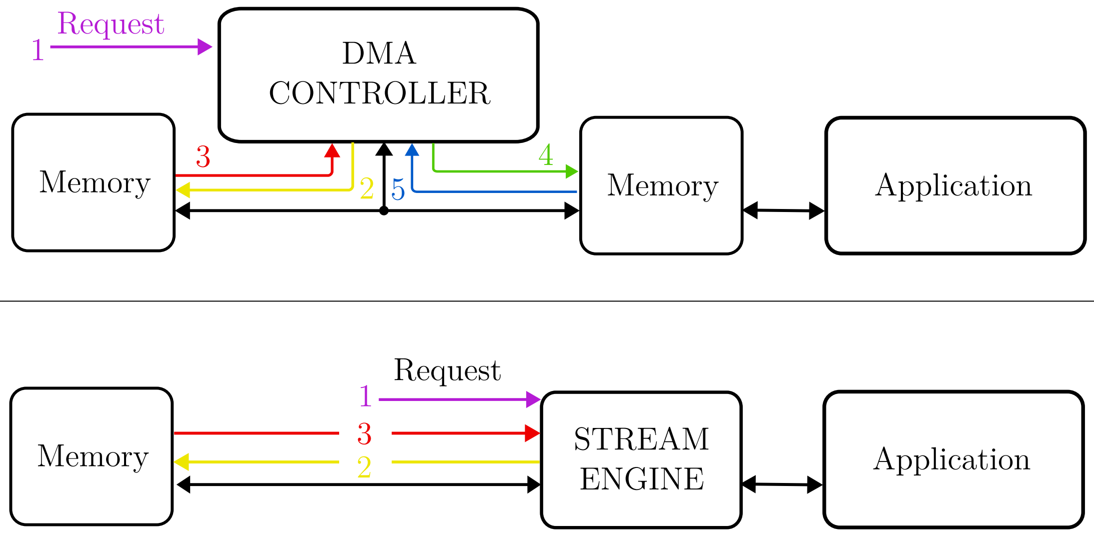


Figure 2.1: A comparison between DMA engines and Stream engines. The colored arrows represent bus transactions occurring as a result of the memory access sent by the request. A total of 4 bus transactions are issued in the DMA controller example, while only 2 are issued in the stream engine example.

2.2 Reconfigurable Technologies

Digital logic written in a HDL, while testable and possible to develop with the use of simulators, will at some point need to be deployed to a targeted technology to become truly useful. Normally this is done in the form of ASICs which are, by design, unchangeable. This encourages their design to be generic and programmable by software instead of specialized for one specific task. This causes programmable ASICs – such as processor systems – to be extremely performant and efficient *generic* computational devices, but not even remotely close to as efficient as a *dedicated* computational device could be. The reason dedicated ASICs do not exist for every use case is mainly due to non-recurring engineering (NRE) costs associated with designing and manufacturing ASICs and their long time to market. In other words, they do not always make financial sense. Field Programmable Gate Arrays are an alternative approach of implementing HDL code in hardware. They allow for dedicated computational devices while avoiding many of the costs associated with ASIC development. This is due to FPGAs being programmable which significantly lowers the NRE costs and time to market. Most modern FPGAs are reprogrammable, which allows for multiple interchangeable dedicated hardware implementations to run on the FPGA. The way FPGAs are implemented is that instead of etching the logic gates which define the desired boolean function, logically equivalent lookup Tables (LUTs) and D flip-flops (DFFs) are used. The LUTs are programmable such that any boolean function with the same number of inputs and outputs as the LUT can be implemented. Together LUTs, which implement combinatorial boolean functions, and DFFs, which capture the stateful (sequential) functions, model any possible RTL code.

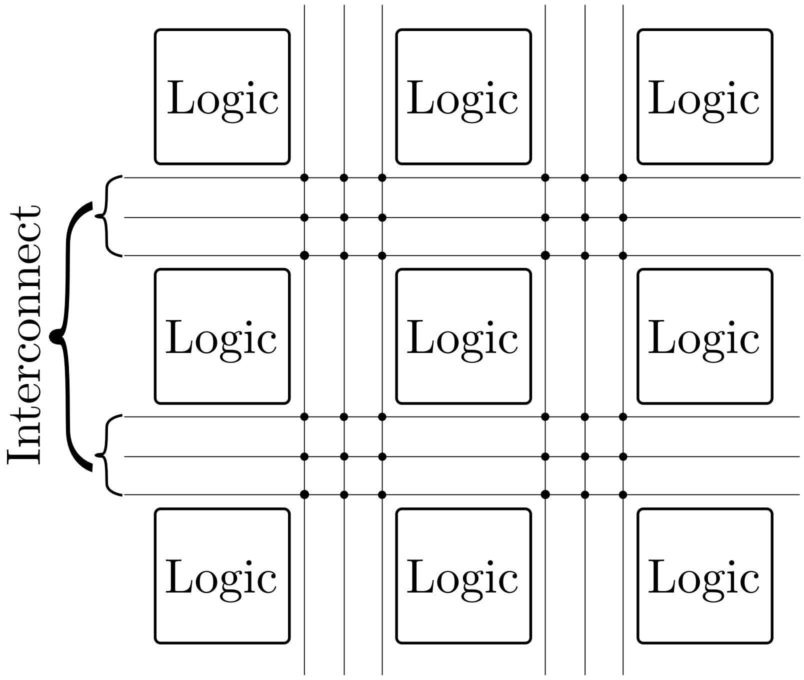


Figure 2.2: Simplified view of island style FPGAs. It can be summarized as a sea of configurable logic gates interconnected with configurable wires. The logic blocks contain one or more pairs of LUTs and DFFs. The interconnect is the reconfigurable wires which allow for the logic blocks to communicate with one another. RAM blocks and carry signals also exist in island style FPGAs, but are not depicted here.

The most common type of FPGA is known as island-style, seen illustrated in Figure 2.2. They are built out of a set of different functional elements (FEs), which together form the avilable hardware resources in an FPGA. The FEs relevant to this thesis are LUTs, DFFs, carry signals between LUTs and RAM blocks. RAM blocks exist scattered across the FPGA to allow for efficient storage of intermediate results. Together, these resources are typically shown as fractions of the utilized resources over the available resources on a given FPGA. Utilization metrics are provided by the tool used to convert the HDL code to an FPGA design.

2.3 Radiation Hardening

The lack of a protective atmosphere and magnetic field leaves ICs deployed in space vulnerable to the vast amounts of radiation present there. There are multiple hazards caused by radiation, such as single event upsets and permanent failure of the affected IC [16], [17]. There are a variety of different methods one can use to mitigate the effects of radiation on ICs which when put together can almost completely preserve the functionality, even in the hazards of space.

For this particular project, the target hardware – the GR765 – will already be radiation hardened and as such will not be accounted for in the development process. It is however a key aspect of developing hardware for space applications and interesting in its own right. The hardening techniques used in the FPGAs developed by NanoXplore, such as the eFPGA in the GR765, are [18]:

- Dual Interlocked storage Cell as memory latches [19]
- Error Detection and Correction for BRAM blocks [20]
- Triple Module Redundancy for all other blocks [21]

One notable difference between rad-hard FPGAs and normal FPGAs is the maximal hardware resources available from vendors. Comparing the largest available rad-hard FPGAs from Microchip and NanoXplore (PolarFire and NG-ULTRA respectively) with the largest non rad-hard AMD FPGA (Virtex UltraScale+ VU19P) there is about an order of magnitude difference in the LUTs and a factor 2 difference in RAM blocks, both in favor of the AMD FPGA [6], [7], [22]. This emphasizes the importance of optimizing resource utilization when developing solutions, such as hardware accelerators, targeting rad-hard FPGAs.

2.4 Target System

The target system for this project is Frontgrade Gaisler's upcoming rad-hard 8-core SoC known as the GR765[5]. While it does have all the features one would normally expect of an 8-core SoC, there is one specific feature that will be relevant for this thesis, namely the aforementioned eFPGA. The eFPGA is connected to the rest of the SoC through an interface called SoCBridge. The SoCBridge is in turn connected to an AMBA AHB bus which is the main system IO bus.

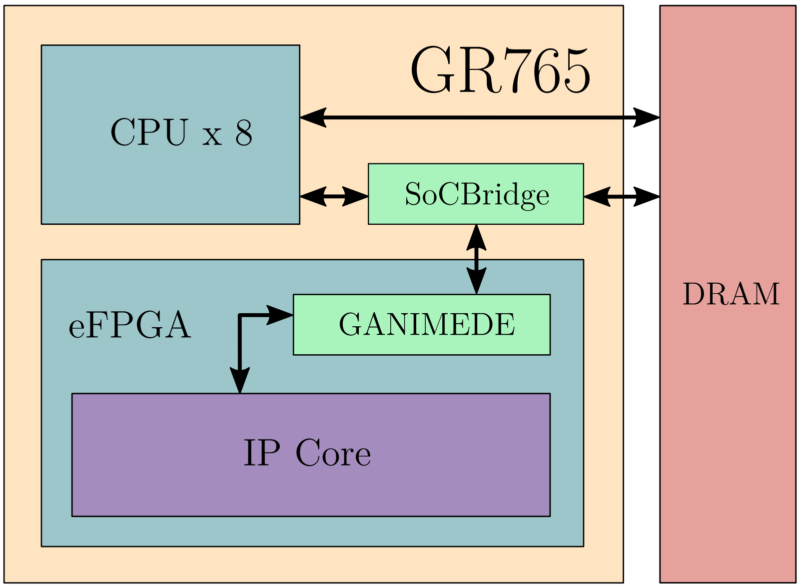


Figure 2.3: Simplified diagram of GR765 containing only the parts relevant to this thesis.

While there are many modules present on a modern SoC, only a subset of the GR765's design is relevant for this thesis. A diagram showing a simplified version of the target system for this project with all irrelevant parts removed can be seen in Figure 2.3. Note that the GANIMEDE and IP Core blocks are deployed to the reconfigurable eFPGA and not the rest of the SoC. The CPUs and DRAM operate as they would in any other system for the purposes of this project, which leaves the SoCBridge and eFPGA as the components of interest in this configuration.

2.4.1 eFPGA

The aforementioned eFPGA is a rad-hard model called eFPGA28RH-30K developed by NanoXplore. While having a CPU and an FPGA on the same circuit board is fairly commonplace, having both on the same SoC is not. The eFPGA's available resources can be seen in Table 2.1. The available resources are few compared to commercial off the shelf FPGAs partially due to being rad-hard and partially due to being embedded. Comparatively, AMD's currently commercially available FPGAs range from a few thousand LUTs to a few million with most being a few hundred thousand LUTs[23].

Table 2.1: Available hardware resources on the target eFPGA

Resource	**Available**
DFF	30720
4-LUT	32640
Carry	7680
RAM blocks (48Kib)	40
Embedded DSP	80
I/O	192 Inputs / 192 Outputs

2.4.2 SoCBridge

GANIMEDE needs a communication protocol in order to accomplish anything useful as a stream engine. In this thesis the communication protocol chosen is the SoCBridge protocol, and will be described in detail in this Section as its implementation details are important to understanding architectural decisions presented and discussed in later chapters.

The SoCBridge is a component developed at Frontgrade Gaisler to provide a simple expansion interface for external chips or systems. In the GR765 it is also used to interface with the eFPGA as seen in Figure 2.3, since it is still distinct from the processor system despite being on the same chip. To provide the promised simplicity, it offers its own protocol which is easier to interface with than the system AHB bus it is connected to.

The SoCBridge module interfaces two AMBA AHB buses with Frontgrade Gaisler's serial SoCBridge communication protocol [24]. An AMBA APB bus is used to configure the run time behavior of the primary SoCBridge [25]. Furthermore, an additional module named SoCBridge PARIF can be used which converts the serial communication to a parallel interface.

This thesis will only detail the SoCBridge protocol using the PARIF module. The resulting communication interface consists of 10 bits in both directions, 8 bits of data, 1 bit of parity for the data and 1 bit for the clock signal. One key aspect of the SoCBridge protocol is that it is asymmetrical – one primary side and one secondary side. The primary side is responsible for setting the clock frequency of the communication, while the secondary side consequently follows the frequency set by the primary side. A data transfer example is shown in Figure 2.4.

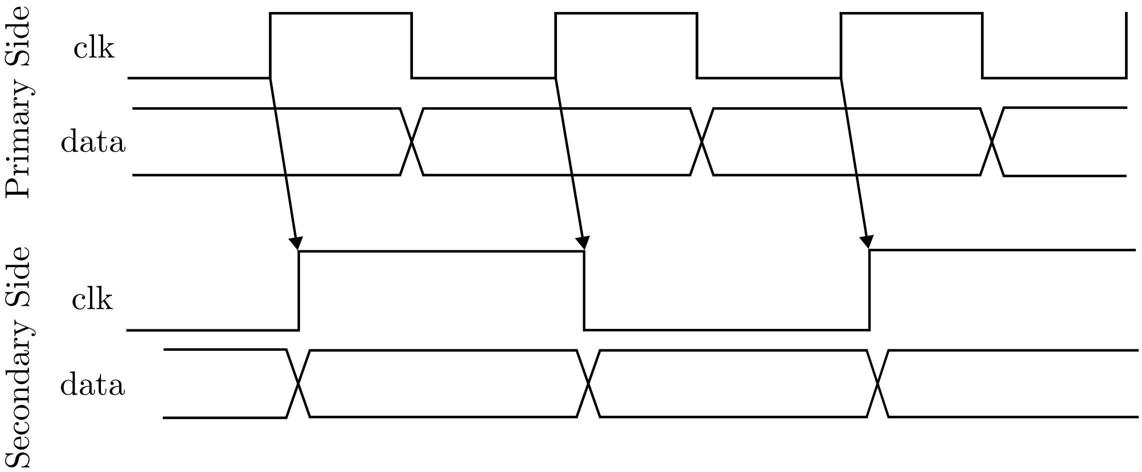


Figure 2.4: Timing diagram of a transfer over the SoCBridge protocol. The primary side clock controls the secondary sides clock. Data is captured into registers on the primary side on the falling edge of the primary side clock. On the secondary side, data is captured on any edge of the secondary side clock.

While the exact specification of the protocol can be found in Frontgrade Gaislers public IP core documentation, some important features and certain limitations will be discussed in the following paragraphs [26].

Read and writes are allowed with or without an address supplied. If no address is supplied, the most recently supplied address is used as a base for a unit stride access. Thus, each subsequent non-addressed command will access the byte following the last accessed byte in memory. A summary of the SoCBridge commands can be seen in Table 2.2

Table 2.2: Commands specified in the SoCBridge protocol. x_1 represents bits from the size parameter. The size of the packet is determined by 2^{x_1}

Cmd	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Addressed**	**Data**	**Response**
Idle	0	0	0	0	0	0	0	0	No	No	No
Write	1	0	0	0	0	x_1	x_1	x_1	Yes	Yes	Write ack
Write	1	0	1	0	0	x_1	x_1	x_1	No	Yes	Write ack
Read	1	1	0	0	0	x_1	x_1	x_1	Yes	No	Read response
Read	1	1	1	0	0	x_1	x_1	x_1	No	No	Read response

Responses are sent after a command is received. The response contains information regarding what command caused it to be sent, such as whether it was addressed or if it was a read or a write. The size of the sent command is also returned in the response. A summary of the SoCBridge responses can be seen in Table 2.3

Table 2.3: Responses specified in the SoCBridge protocol. x_1 represents bits from the size parameter. The size of the packet is determined by 2^{x_1} x_2 indicates if the sent command was an addressed or non-addressed command.

Response	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Addressed**	**Data**
Write ack	0	0	x_2	0	1	$x_{1,2}$	$x_{1,2}$	$x_{1,2}$	No	No
Read response	0	1	x_2	0	0	$x_{1,2}$	$x_{1,2}$	$x_{1,2}$	No	Yes

In the primary SoCBridge only one address may be stored at a time, which means that reads and writes are not mutually exclusive with regards to the last addressed byte. In listing 2, a sequence of commands sent from GANIMEDE is shown in which one might expect the non-addressed read on line 3 to fetch 4 bytes from address 0x1004. However, this is not what happens. That read will instead fetch 4 bytes from address 0x4004 due to the addressed write on line 2 to address 0x4000. The root cause of this is assumed to be that the SoCBridge only has one address register which is shared between reads and writes. We call this limitation the addressing limitation.

```
1 read  4 bytes, 0x1000
2 write 4 bytes, 0x4000
3 read  4 bytes //Reads 0x4004
4 write 4 bytes //Writes 0x4008
```

Listing 2: Command sequence side effects causes the read on line 3 to continue reading after the written address.

```
1 read  4 bytes, 0x1000
2 write 4 bytes, 0x4000
3 read  4 bytes, 0x1004
4 write 4 bytes, 0x4004
```

Listing 3: Command sequence in which side effects do not affect addresses read or written to

The addressing limitation incurs additional overheads as certain traffic patterns, such as the one in listing 2, would need to replace non-addressed commands with addressed commands. In listing 3 the same pattern as in listing 2 is shown, but the side effects of previous commands are avoided. The overheads consist purely of the packet header which, as shown previously in Table 2.2, is one byte and conditionally an address. For the rest of this thesis the address is assumed to be 4 bytes long. The possible packet header sizes are therefore 1 and 5 bytes for non-addressed and addressed commands respectively.

Another limitation in the SoCBridge is the cyclic nature of its internal incrementing address used during non-addressed command sequences. We call this limitation the increment limitation. When the primary SoCBridge uses the AMBA AHB bus, the address supplied the bus is constructed by combining the base address supplied at the beginning of the command sequence with an 8-bit incrementing register. The size of the register makes a long continuous access with non-addressed commands cyclic due to overflow in the incrementing register. Therefore, when constructing a secondary SoCBridge interface – such as the one in this thesis – extra logic has to be added to ensure large accesses function as expected.

2.5 SHyLoC IP Core

Compression of hyperspectral images is an important and common task which is needed to alleviate the limited bandwidth resources experienced on satellites [27]. Furthermore, as smaller satellites continue to grow in popularity, a demand for low power, low weight yet high performance implementations of hyperspectral image compression is needed [2]. To meet these demands, hardware accelerators targeting these compression algorithms have been developed.

The SHyLoC IP Core is one of these accelerators and follows the CCSDS121.0.B-2 and CCSDS123.0-B-1 image compression standards [10]. In short, the CCSDS123.0-B-1 standard expects a hyperspectral 3D image as input and produces either a compressed image or an intermediate representation of the original image. This intermedeiate representation can then be fed into an implementation of the CCSDS121.0-B-2 standard which compresses the intermediate representation. In either case, the compression is lossless and will produce the original image given that the correct options are used when decompressing the image.

For the purpose of this thesis, only the input and output of the SHyLoC IP Core are of interest. SHyLoC expects a FIFO data stream interface, using a ready-valid synchronization structure, and outputs a similar FIFO data stream. While it is possible with configuration of SHyLoC to vary the width of the FIFO input data stream, only a data width of 16 bits will be considered in this thesis. The output FIFO stream is 32 bits wide.

2.6 Related Works

Stream-Dataflow Acceleration is a paper in which the authors define a general data streaming architecture which is optimized for high computational intensity applications [9]. The idea is to enable software compilers to insert special instructions to start the in- and output streams of data, similarly to SIMD compilers, and allow the data to flow through a dataflow description of the computational task. The streaming of data hides the latencies associated with accessing data, as the input and output streams overlap, thus reducing overall execution time.

While this work is indeed very interesting and quite similar to our work, it is not exactly what this thesis aims to explore. Our work is targeted towards software decoupled execution and low resource utilization. Furthermore, Stream-Dataflow Acceleration targets a CGRA instead of FPGA, which also differentiates their work from ours.

A High-Performance, Energy-Efficient Modular DMA Engine Architecture is a very recent paper where the authors develop a state of the art DMA Engine, which they call iDMA [28]. iDMA formalizes and gives an architectural description of what a modular and high parametric DMA Engine is made of. They define 3 parts, the front-, mid- and back-end. The front-end is responsible for controlling the DMA Engine and the back-end implements the network ports through which the DMA Engine moves data. The mid-end translates the requests provided to the front-end into accesses which the back-end can understand.

iDMA is very similar in idea regarding architecture and organization of hardware modules, but it describes a DMA Engine - something that moves data between memory and memory. This is in contrast to our work, where data is moved between memory and application.

3

Methods and Limitations

The goal of this thesis is to develop a stream engine optimized for space-grade hardware. The following Sections will describe the development plan, method and limitations in this work.

3.1 Limitations

Due to various constraints set by factors such as target hardware, time constraints and available tools, certain limitations are applied to the project. These limitations restrict some results to being mere heuristics or approximations of the actual metrics.

One such limitation which has already been mentioned is that the target SoC – the GR765 – is still in development and thus, does not exist as of yet. This limits the accuracy and attainability of metrics as measurements will have to be performed on mock instances of the system or derived from heuristics. To remedy this, one can instantiate a similar system, such as a LEON3 soft core with the additional entities needed to test GANIMEDE. It was deemed not relevant enough to do this as the GR765's datapath is completely different to that of a LEON3 SoC.

Another important metric is power which many hardware development tools provide in a report based on analysis of the synthesized design describing the calculated power draw in various states and conditions. However, the tools for NanoXplore's FPGAs do not provide this kind of power analysis. Since power calculations can not be obtained through NanoXplore's tools nor from the GR765 itself, power will not be considered in this thesis.

Latency is yet another metric which will not be taken into account. Building on the fact that the target system does not exist, any tests performed would need to have a mock latency added artificially in the test environment. Since the focus of the work is building a minimized interface with maximal throughput, the external memory latency is not that important and will be disregarded in this thesis.

Furthermore, this study only examines the use of one HA, the SHyLoC IP core. Thus, the varying specifications of other HAs – which will undoubtedly set different requirements on GANIMEDE – will not be taken into account.

3.2 Methodology

To answer the questions presented in Section 1.2, a development plan is needed. Below is the plan summarized into few bullet points ordered in chronological order.

1. Develop a single issue GANIMEDE as specified in Section 4.1.
2. Measure the optimal throughput and resource utilization of the single issue GANIMEDE.
3. Develop the performance optimized GANIMEDE as specified in Section 3.2.1 and 4.2.
4. Measure the optimal throughput and resource utilization of the performance optimized GANIMEDE.
5. Identify entities of the performance optimized GANIMEDE with high resource utilization.
6. Minimize the resource utilization of the identified entities in the performance optimized GANIMEDE, creating the resource utilization optimized GANIMEDE.
7. Measure the optimal throughput and resource utilization of the resource optimized GANIMEDE version.
8. Measure the run time of the single issue, performance optimized and resource utilization optimized versions serving the SHyLoC IP Core and compare against the ideal run time of a hyperspectral image.

To evaluate GANIMEDE by measuring throughput and resource utilization, different comparisons between baselines or calculated ideal cases must be made. Since the GR765 does not yet exist and no other directly comparable results could be found by the authors, the methodology must be centered around self comparison and calculated ideal cases rather than real world comparisons. In the following Sections the aforementioned methodology as well as the practical measurement setup will be discussed.

3.2.1 Throughput Considerations

The goal of optimizing for throughput is to achieve the highest possible utilization of the provided link. As one objective of the thesis is to achieve the maximal throughput possible, the following method will be adopted:

1. Measure the unhindered bidirectional throughput of GANIMEDE.
2. Identify a bottleneck in GANIMEDE.
3. Implement solutions which reduce the impact of or entirely remove the bottleneck.
4. Repeat steps 1-3 until no new bottleneck is identified or the ideal bidirectional throughput is achieved.

Since the target hardware does not yet exist, only simulation results will be collected for the throughput.

In the case study performed by this thesis, the link's communication protocol is the SoCBridge protocol. Since two bytes in total can travel through the wires per cycle (one byte in each direction per cycle) the maximal idealized throughput will be two bytes per cycle. When considering protocol overheads however, different maximal throughputs can be calculated for differing packet sizes and protocol behavior. Listed in Table 3.1 are the idealized throughputs for the different packet sizes available in the SoCBridge protocol. The reason for providing both the addressed and the non-addressed ideal throughputs is due to the addressing limitation explained in Section 2.4.2.

The ideal throughput is calculated with the formula $\frac{2N}{2H+N}$ where N is the packet size and H is the header size assuming each byte takes one cycle to transmit. The formula models the perfect, no down time, transfer of data with absolute minimal protocol overheads. The non-addressed commands models a perfect transfer with minimal headers ($H = 1$) and the addressed commands models the worst case size ($H = 5$) brought on by the addressing limitation. The utilization is calculated by dividing the throughput by the maximal idealized throughput.

Table 3.1: Ideal throughputs for addressed and non-addressed commands in the SoCBridge protocol.

Packet Size	Addressed		Non-addressed	
(B/P)	Throughput (B/C)	Utilization	Throughput (B/C)	Utilization
128	1.8551	0.92755	1.9692	0.98460
64	1.7297	0.86485	1.9394	0.96970
32	1.5238	0.76190	1.8824	0.94120
16	1.2308	0.61540	1.7778	0.88890
8	0.8889	0.44445	1.6000	0.80000
4	0.5714	0.28570	1.3333	0.66666

To empirically measure the maximal throughput of different versions of GANIMEDE, a simultaneous and unhindered read and write of 256KiB each will be run. The measurement will be the time in clock cycles from the first read instruction beginning transmission to the final write instruction receiving an acknowledgment. The total data transmission of 512KiB will be divided by the total number of clock cycles measured to produce the throughput of that particular GANIMEDE version. The test will be run for every packet size listed in Table 3.1.

3.2.2 Resource Utilization Considerations

As mentioned in Section 1.1, a major goal of this project is to minimize the resource utilization of the design. In order to accomplish this, one first needs to know where resources are being wasted. This is done mainly by looking at the synthesis reports generated by NanoXplore's tools. In these reports, resource utilization is shown per module in the design in the form of LUTs, DFFs, RAM as well as various other resources which are not relevant for this design. This is not a perfect metric as different components vary in complexity and resource utilization will likely vary accordingly, but it does provide a useful starting point in that it shows which components in the design use the most resources.

To gain further clues about where unnecessary resources are being used, NanoXplore's integrated development environment Impulse is used. In Impulse, there is a "floorplan" tab in which one can view all allocated resources on the FPGA. For certain instances, Impulse can provide the exact line in the source code responsible for the allocation of that instance. A LUT may, for example, be marked with `component_x|add_L515` which means that the line 515 in the source code for `component_x` contains an addition which caused this LUT to be allocated.

Since Impulse mainly shows where arithmetic operations occur, any other minimizing action has to be derived from the source code itself. To realize minimization in the source code, a variety of strategies are used. These strategies are presented in Section 4.3.

3.2.3 SHyLoC measurements

To measure the run time of SHyLoC an extra bit of glue logic is needed because of the different bit widths in SHyLoC's and the SoCBridge's interfaces. The SoCBridge is 8 bits wide in both directions whereas the SHyLoC IP Core has a 16 bit wide input and a 32 bit wide output. To remedy this, two buffers, a serializer and a de-serializer has to be introduced to connect GANIMEDE and SHyLoC as shown in Figure 3.1.

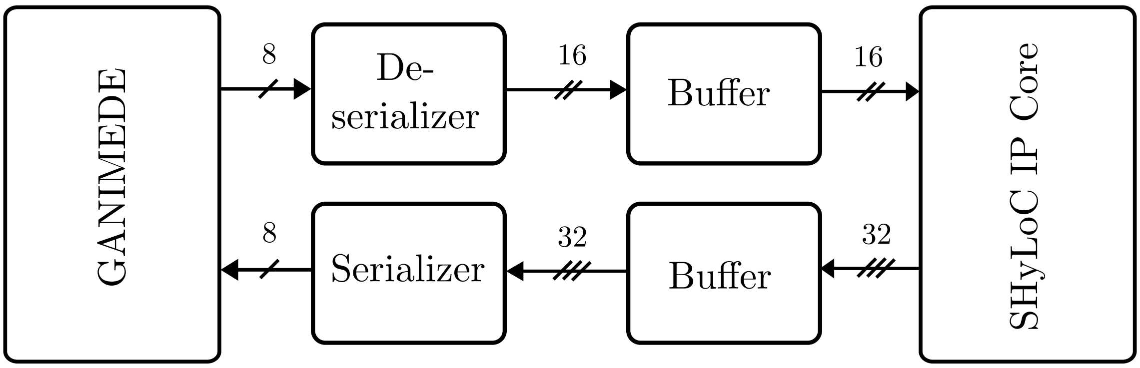


Figure 3.1: A block diagram depicting the connection between SHyLoC and GANIMEDE.

The serializer, de-serializer and buffers all implement a FIFO interface, and are on the same clock. This can lead to bottlenecks in cases where the buffers are too small since the serializer will only consume one 32 bit word every 4 cycles. Thus, the buffers will have to be sized accordingly to avoid this issue. To gather the empirical throughputs of every GANIMEDE version, the same strategy as detailed in the end of Section 3.2.1 will be used.

3.2.4 Experimental Setup

The setup in which all tests will be run is constructed as a VHDL testbench. The testbench will then be simulated in Aldec's Riviera simulator. In the testbench there is a primary SoCBridge connected to a simulation-only AMBA AHB and AMBA APB bus. An AMBA AHB test RAM is also connected to the AHB bus. GANIMEDE will be instantiated and connected to the primary SoCBridge.

The simulation-only AMBA buses allow for simple VHDL procedures to be run, in which AMBA commands are sent through test requesters on the specified bus. This is what replaces the CPU in the testbench. In Figure 3.2 an overview of the described system can be seen.

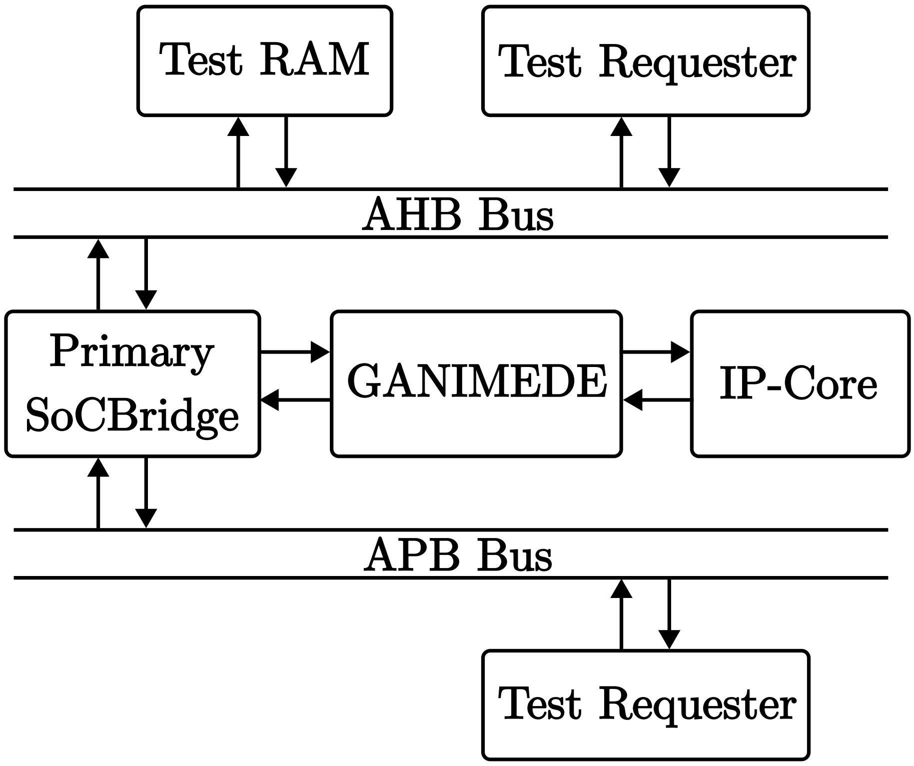


Figure 3.2: Block diagram of the experimental setup. A primary SoCBridge is connected to GANIMEDE, where an IP core may optionally be slotted in.

All tests use one read and one write instruction, which are sent from the AHB test requester consecutively. The read arrives first, followed by the write instruction before any commands are issued by GANIMEDE.

The period between sending the instructions and GANIMEDE fully receiving them are fully ignored in the tests. Instead, the first time measurement occurs when the driver's TX FSM enters a non-idle state. The last time measurement occurs when the driver's RX FSM enters the a non-idle state for the last time during the test.

4

Architecture

Our proposed stream engine GANIMEDE is composed of 4 discrete entities, the *driver*, the *translator*, the *control unit* and the *management unit*. The management and control unit work together to form the stream dispatch system, in which instructions are received, stored and issued to the functional parts of GANIMEDE. The driver implements the underlying communication protocol, which in this case is the SoCBridge protocol, through which data and GANIMEDE instructions flow. Last but not least is the translator, which is responsible for subdividing the protocol agnostic GANIMEDE instructions into commands which the driver can understand.

As outlined by Section 3.2, the coming three Sections will describe the actions taken to achieve the single issue, performance optimized and resource utilization optimized versions of GANIMEDE. The single issue version will serve as our baseline model of a stream engine for this system so as to enable comparison between the different versions.

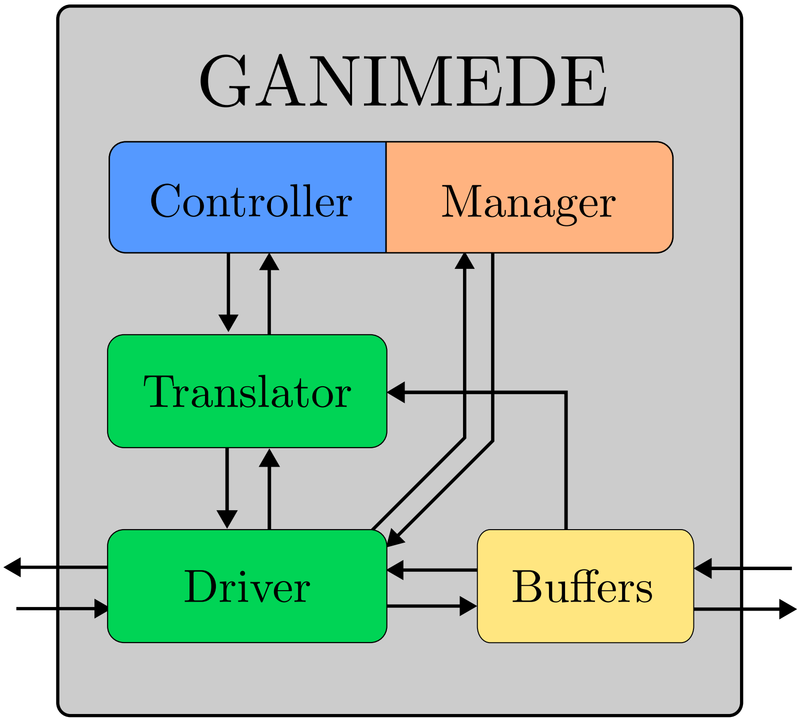


Figure 4.1: Block diagram of GANIMEDE's topology.

4.1 Single Issue GANIMEDE

This version of GANIMEDE can receive one read and one write instruction from the CPU at a time, which then executes to completion before it is able to receive further instructions. It supports one driver which means it can communicate using one external protocol, in this case Frontgrade Gaislers SoCBridge protocol [26]. Data is transferred between GANIMEDEs internal buffers and the internal IP core through FIFO interfaces to enable streaming with minimal overheads. An overview of the architecture can be seen in Figure 4.1. In the following subsections, each entity of GANIMEDE will be described.

4.1.1 Management unit

The management unit stores the instructions waiting to be issued to GANIMEDE. For this purpose it is essentially a small RAM cell with the necessary interfaces to communicate with the controller and driver. The instructions are written to GANIMEDE by the CPU through a memory mapped interface which the primary SoCBridge provides.

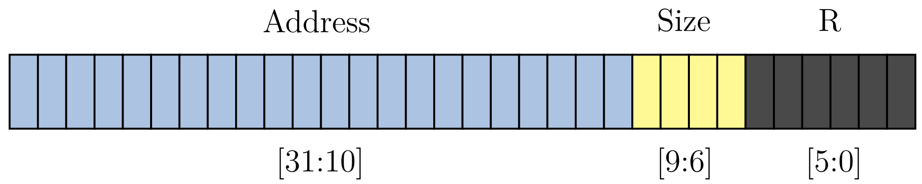


Figure 4.2: Instruction word used by the management unit where *R* means reserved, *Size* describes the number of bytes to read/write and *Address* symbolizes the base address to operate at.

An instruction in GANIMEDE contains an address and a number representing the number of bytes to read or write. The formatting of an instruction word can be seen in Figure 4.2. One striking detail is the amount of reserved bits present. Initially, the instruction word was meant to hold three command bits as well which would allow further control over the way GANIMEDE would handle the instruction. Due to time constraints this was scrapped but the features it would provide are further described in Section 6.3.1 under "Different kinds of access types". Of the remaining three reserved bits, only one currently serves a function and exclusively in the minimized version of GANIMEDE, which is further explained in Section 4.3.4.

In the instruction word, the address is truncated to 22 bits despite GANIMEDE using a 32-bit datapath[1]. Thus, the 32-bit address that GANIMEDE infers from the instruction word has the 10 lowest bits padded with zeroes. This limits GANIMEDEs precision to operating on data aligned to KiBs in memory. This KiB alignment is also reflected in the 4-bit size parameter in the instruction word. This number is parsed by the management unit as the exponent in the formula $n = 2^x \times 2^{10}$, where

[1]The GR765 actually has a 64-bit datapath but the testbench used in this project was 32-bit. This can easily be changed for other datapath widths.

n is the actual number of bytes to read/write and x is the unsigned integer stored in the instruction word. Thus, GANIMEDE will read/write 2^x KiBs of data where $x \in [0, 15]$.

If an instruction is present in the manager's memory, it tries to dispatch it to the controller. If there is both a read and write instruction, the read instruction is always dispatched first, and then the write.

4.1.2 Control Unit

The control unit can be seen as a subsystem for the management unit. It is in essence the interface between the management unit and the driver. When the control unit receives an instruction from the manager, the instruction is presented to the translator. When the translator has received the instruction, it signals the controller which then tells the management unit to clear the issued instruction.

Essentially, any number of translators can be connected to the control unit. While not investigated in this thesis, supporting more than one translator-driver pair could lead to other very interesting case studies. One example is supporting a sensor data stream entering through one interface, being processed by the application and finally being written to system memory. More discussion about this topic can be found in Section 6.3.

4.1.3 Driver Implementation

The driver implemented in this case study is that of the SoCBridge [26]. As a reminder, the SoCBridge has 8 bits of data, one bit of parity and one clock signal. The communication is bidirectional, meaning that data can be sent and received simultaneously. Since the driver accepts incoming messages and independently generates outgoing messages, some method of mutual exclusion must be implemented to eliminate race conditions. This was accomplished with the use of FSMs, as simple static priority arbiters can easily be integrated into the decision making logic residing in the FSMs.

Implementing the FSMs for the SoCBridge protocol was done in two separate but functionally similar FSMs where one handles the receiving link and the other handles the transmitting link. Both FSMs consist of the same states and transitions, but differ in what each state represent and how the FSMs affect each other. In Figure 4.3, an overview of the different states and transitions are shown. To describe the purpose of the states, Table 4.1 is provided to summarize the function of both the receiving and transmitting FSM at that given state.

While a description of how every state in both FSMs work would be much too lengthy, the `HEADER` state will be described in more detail due to its complexity. The `HEADER` state has 5 outgoing state transitions which trigger on different conditions. In the transmitting FSM, the conditions are based on the internal state of the driver and in the receiving FSM, the conditions are based on the first byte of the received packet. If it is an an addressed command `ADDR1` will be the next state, and in the

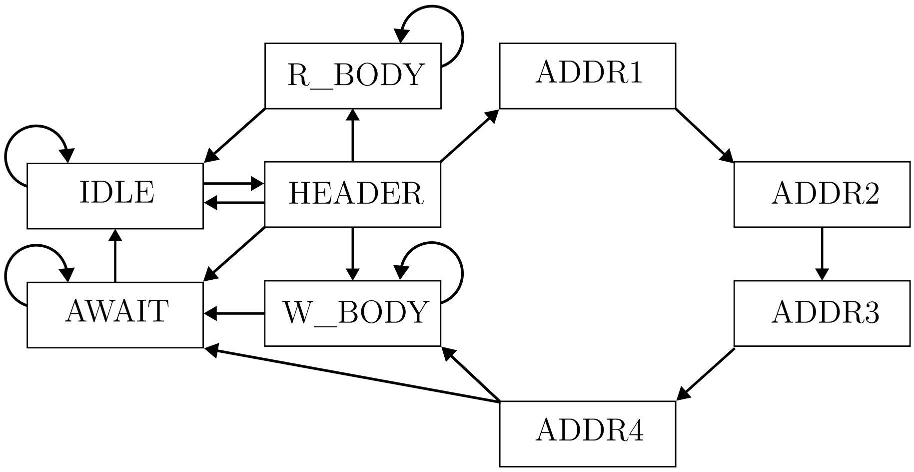


Figure 4.3: SoCBridge state transition diagram for both the receiveing (RX) and transmitting (TX) end.

Table 4.1: Description of states contained within the TX/RX FSMs in the SoCBridge driver.

State	**Receiving FSM (RX)**	**Transmitting FSM (TX)**
IDLE	Await new incoming data	Await internal stimulus
HEADER	Received header, classify packet type	Transmit correct header, decide appropriate next state
R_BODY	Receive data body for outgoing read	Transmit data body for incoming read
W_BODY	Receive data body for incoming write	Transmit data body for outgoing write
AWAIT	Await successful completion of response transmission	Await successful completion of response reception
ADDR1-4	Receive address bytes for memory mapped read/write	Transmit address bytes for memory mapped read/write

case of an non-addressed command either `AWAIT` or `W_BODY` will be the next state for a read or write command respectively. If it is a read response `R_BODY` will be the next state. If none of the above conditions are true, the next state will always be `IDLE`. The attentive reader will notice that the write acknowledgment response is not accounted for explicitly, but rather implicitly through the last catch-all case which returns to `IDLE`. Since this is the intended behavior after receiving or sending a write acknowledgment, it makes no functional difference to explicitly specify this case.

4.1.4 Translator Implementation

The translator is the connective layer between large, protocol agnostic memory accesses provided by the controller and commands which the driver can understand. The way the translator's and driver's FSMs communicate is by inspecting the others current state, similar to snooping. Both FSM are programmed in such a way to make this snooping work like synchronization barriers to ensure no part of the memory access is lost between the translator and driver FSMs.

The FSM controlling the translator, which can be seen in Figure 4.4 is divided into 4 states where all except one of them awaits a certain stimuli before continuing to the next state.

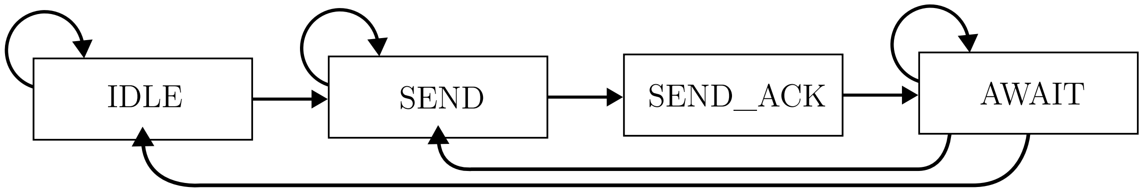


Figure 4.4: Translator state transition diagram. Describes the procedure of sending a part of a message through the translators associated driver. The controller dictates when the translator wakes up from the idle state.

The `IDLE` state awaits a memory access from the controller, which upon reception progresses the FSM into the next state.

The `SEND` state indicates that a memory access is present in the translator state and that an instruction has to be issued in order to progress. Progression into the next state occurs when the drivers transmitting FSM enters the `HEADER` state to send the corresponding header.

The `SEND_ACK` state is a transitory state which updates the translators state to reflect that an instruction is being sent, namely the memory pointer to where the next instruction should be targeted and the remaining bytes to transfer.

The `AWAIT` state is the idle state of the translator during a memory access. To transition away from this state the response of the previous instruction must be fully received. The state transition may either be to `IDLE` or `SEND` depending on whether the memory access is complete or not.

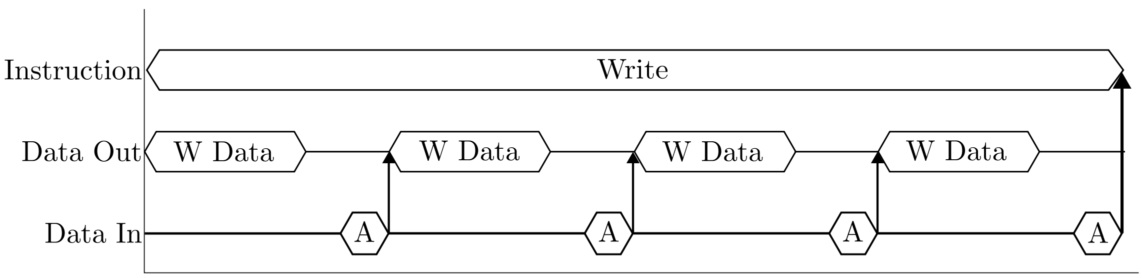


Figure 4.5: Data transfers during a write command. ”Write” is a write command with data sent from GANIMEDE and the ”A” is the write acknowledgment.

In the translator one of two memory accesses can be handled at a time, either a read or a write. A write memory access will cause the driver to send an appropriate number of write transactions, each awaiting their respective acknowledgments before the translator issues the next command. This behavior is shown in Figure 4.5.

Similarly to write memory accesses, reads function by the translator issuing an instruction and awaiting its completion before sending another command. The behavior of read memory accesses is shown in Figure 4.6

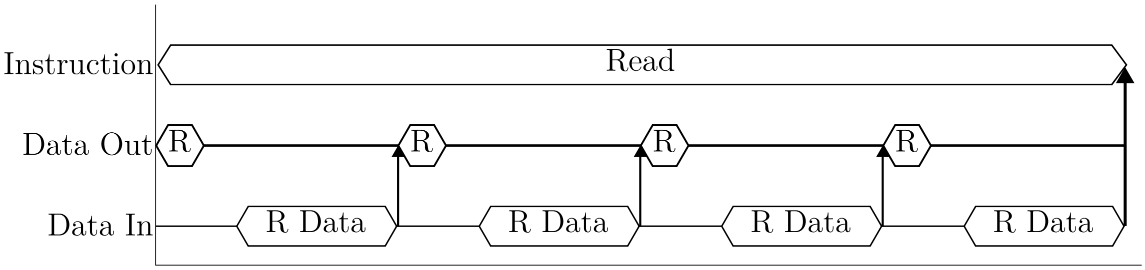


Figure 4.6: Data transfers during a read command. ”R” is a read request sent from GANIMEDE and the ”R Data” is the read response containing the requested data.

The translator also takes available buffer space into account when issuing commands such that the receiving read buffer is never full and that the write buffer has enough data for one command.

Lastly, the translator also provides a signal which tells the driver FSM whether or not to include the address in the command or not. This is done to reduce protocol overheads, allowing for burst-like behavior in the case when the address in not needed is the header. The translator implements logic to handle the increment limitation (described in Section 2.4.2) which becomes present during the bursts.

4.1.5 Buffers

Since the SoCBridge protocol does not provide any kind of back pressure handling, two FIFO buffers are placed between the IP core and driver as a mitigative measure. These buffers implement block RAM cells as storage which allow them to store large amounts of data without using using too many resources. The implementation as it stands does however require that the read and write ports on the RAM can run on separate clocks since the SoCBridge – and by extension the SoCBridge driver – run on a separate clock to the rest of the eFPGA.

4.2 Performance Optimized GANIMEDE

To close the gap in performance between the basic version of GANIMEDE and the ideal throughputs shown in Table 3.1, 3 performance optimizations were theorized. Due to time constraints and the limitations set out in Section 3.1, FSM timing improvements and IO pipelining were the only implemented performance optimizations. The other performance optimization will be described as it would have been implemented given more time.

4.2.1 FSM timing improvements Implementation

The idea of this optimization is simple: try to hide latencies whenever possible. One example of unnecessary latency is in the case of sending a read request only after receiving the entire previous message. This is simply not needed as the SoCBridge protocol ensures that the data is continuous when any read response is sent. Therefore it is possible in the driver's TX and RX FSMs to simply not stay in the `AWAIT` state until the entire message is received, and just send the next read command as soon as the first byte of the read response arrives. For read instructions, this has the added benefit of allowing uninterrupted continuous reads to completely hide their protocol overhead, as the entire packet can usually be sent before the read response ends.

Further improvements, such as adding edges between the penultimate states and the `HEADER` state, were also added. This removes the need to return to `IDLE` before entering the `HEADER` state if the need should arise.

In Figure 4.7, the beginning of the arrows indicate when the improved driver FSMs knows it is safe to progress to the next command, and the end of the arrow is when the next command is actually sent. As will be explained in the next Section, this property is required to implement IO pipelining.

4.2.2 IO Pipelining Implementation

The goal of IO pipelining is to let data continuously flow from the input, through the target IP core and finally to the output in a continuous stream. Furthermore, IO pipelining can improve the throughput through the communication channel by a factor of 2, since now both directions of the channel can be used simultaneously. During an IO pipelined transfer, the same commands used to execute reads and writes are interleaved, as seen in Figure 4.7, in order to emulate 2 completely separate streams.

To implement this optimization two translators are used, one for reads and one for writes. This allows both translators' next state logic to be simplified slightly, as now only one type of command can cause transitions in each of the translators' FSMs. Special attention now has to be paid to the arbitration of the transmitting channel in the driver as command responses, reads and writes, all may request the channel at the same time.

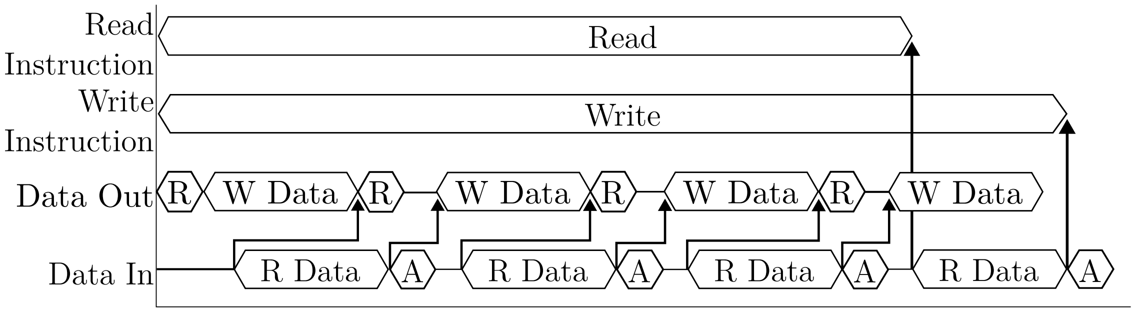


Figure 4.7: Data transfers during a write command. By adding another translator, illustrated as “Read Instruction” and “Write Instruction”, both instructions can be carried out simultaneously.

The arbitration is a implemented as a fixed priority arbiter chained into a round robin arbiter. The fixed priority arbiter decides whether a command response or command should be sent, where command responses take priority. The round robin arbiter decides whether the next command sent should be a read or a write and tries to alternate evenly between the two. There is however a bias towards reads, which means that reads will always be preferred if no other command was sent previously.

As was mentioned in Section 4.2.1, the improved timings in the driver FSMs are needed in order to implement IO pipelining. While the previous implementation was easy to analyze and think about, it prohibits the communication link from being used while waiting for responses. If only the FSM improvements mentioned previously were implemented, a primitive version of the IO pipeline improvement would be possible. In this version, a read response and a write command could operate at the same time, but not truly concurrently in the sense that the first byte of the read response must arrive before the write can commence. This allows *some* performance to be gained, but it is far from perfect.

To further improve performance, the driver was modified to enable communication between itself and the translator. The modification was to replace the `TX_AWAIT` state in the FSM with two boolean flags, `read_in_flight` and `write_in_flight`, in the internal state of the driver. These flags fulfill the same function as provided by the `TX_AWAIT` FSM state, with the improvement that the transmitting FSM is not hindered from sending new packets while waiting for the associated response.

4.2.3 Theorized Multi Packet-In-Flight Implementation

To improve further after the IO pipelining optimization, more than one read and write packet in flight is needed. The reason is that the remaining unutilized throughput on the communication link is due to no commands being present at the primary SoCBridge. Adding support for this requires flow control. Since the primary SoCBridge does not provide any back-pressure signaling, GANIMEDE must implement some flow control mechanism. To allow more than one read and write packets in flight the `read_in_flight` and `write_in_flight` boolean flags are converted to integers. Non-addressed accesses are unused due to the addressing limitation.

Converting the boolean flags entails introducing a generic parameter to control the maximal number of in-flight packets at any given time. Furthermore the translators FSMs needs to make sure that it is considered safe, given the current number of in-flight packets and the generic parameter, to issue another command to the transmitting FSM in the driver.

4.3 Resource Utilization Minimized GANIMEDE

As stated in one of the research questions in Section 1.2 and in the introduction of this thesis, resource utilization is especially important in developing GANIMEDE. An effort has been made to categorize the resource utilization reductions into different reduction strategies, which will be explained while also briefly summarizing their respective impacts on GANIMEDE.

4.3.1 Technology Specific Mapping

FPGAs generally come equipped with commonly used instances such as RAM blocks, register files and digital signal processors among others. When RTL code requires such functionality it can be mapped to one of these technology specific instances instead of using a functionally equivalent amount of LUTs and DFFs. This can save a significant amount of resources as is the case with GANIMEDEs buffers. No attempt was ever made to implement the buffers as anything but technology mapped RAM and thus it is unknown how many resources would have been used to synthesize the memory required. RAM blocks were technology mapped immediately as the cost of LUTs and DFFs to instantiate a RAM inside the eFPGA would be too great. No embedded DSP resources were used to save as many as possible for the targeted IP core.

4.3.2 Arithmetic reduction

There were a number of ways found to reduce the resource allocation for use in arithmetic. Firstly, by providing a range of values an integer signal can assume like in Listing4, the VHDL compiler knows not to allocate more resources for that signal than is necessary. For example, there is no need to give an integer signal 32 bits if it will never assume a value outside the interval [0, 255]. By extension, there is no reason to use a 32-bit adder for two signals which are only 8 bits. Since this effect cascades to everywhere the signal in question is used, it proved to be an efficient technique.

```vhdl
signal int_good : integer range 0 to 255; -- Do this
signal int_bad  : integer;                -- Not this
```

Listing 4: Two definitions of integer signals in VHDL. `int_bad` will likely generate a 32-bit signal while `int_good` will not generate a signal larger than 8 bits, thus reducing resource utilization.

Another way to reduce the arithmetic utilization is to move all constants of a comparative expression to one side of the operator. This way, the expression with all the constants can be evaluated at compile time, reducing the number of operations to be performed at runtime and thus the utilization. An example of this can be seen in Listing5. This technique saves on arithmetic unit allocation but its effects are somewhat reduced if integers already have ranges and it can be overshadowed by expression reduction described in Section 4.3.4.

```vhdl
if some_value = some_constant - 1 then -- Do this
    -- Do stuff
end if;

if some_value + 1 = some_constant then -- Not this
    -- Do stuff
end if;
```

Listing 5: Two if-statements in VHDL. The first can be compiled as one function since all constants are one one side of the equals sign. The second will first have to calculate `some_value` + 1, then compare.

Lastly, In cases with complex arithmetical expressions it may be beneficial to declare a function or array which has the precomputed results based on the possible inputs, much like a LUT. This does, of course, require the set of inputs and outputs to be relatively small and known beforehand but if that is the case then it may save a sizable number of hardware resources.

```vhdl
type arr is array(0 to 3) of integer;
constant precomputed : arr := (5, 11, 23, 47);
signal some_integer : integer;
...
some_integer <= precomputed(some_integer mod 4); -- Do this

some_integer <= 2**(some_integer mod 4) * 6 - 1; -- Not this
```

Listing 6: Example of using precomputed values for an arithmetically complex expression. Of course, the use of magic numbers is not endorsed but present nonetheless in the name of simplicity.

An example of precomputed results can be seen in Listing6. In the example, the signal `some_integer` is the sole variable in the arithmetic expression. Since the expression uses modular arithmetic, it is guaranteed the the array index will always be in the range [0,3] which greatly reduces the number of possible inputs and outputs of the emulated function and thus resource utilization.

4.3.3 Comparative reduction

In iterative processes, there is often a need to compare two values to keep track of, for example, how many iterations are to be done. The intuitive way to do this in a for loop-like format is with an iterator and some limit to stop at, i.e to keep iterating while $iterator < limit$. The problem lies in the $<$ operator as it is quite expensive to synthesize compared to equality checks [29]. However, given that the iterator has a constant increment value and its reset values are known, $\neq$ can be used instead.

```
proc : process(iterator, limit)
begin
    if iterator /= limit then -- Instead of "iterator < limit"
        iterator <= iterator + 1;
        -- Do stuff
    else
        iterator <= 0;
        -- Stop looping
    end if;
end process;
```

Listing 7: Simplified iterative process using $\neq$ instead of $<$, yielding a more space efficient design.

In listing 7, the iterator only ever increments by 1 and only resets to zero. It is therefore certain it will never exceed the limit since it is set to 0 as soon as they are equal.

4.3.4 Expression reduction

Generally, having fewer expressions will yield lower resource utilization. However, since the resources allocated for an expression depends on the operations performed in the expression, it is sometimes possible to rewrite expressions to use less resources. In some cases it is even possible to reduce the resource utilization while increasing the number of expressions. The key is to minimize the number of unique expressions rather than the total number of expressions.

Sometimes it is not possible to reduce the number of unique expressions any further, but it is still possible to reduce resource utilization further. By reusing resource intensive expressions, resources can be saved at the cost of introducing new cheaper expressions to make it happen.

```vhdl
-- Do this
if a < 2 then
    -- Do stuff
end if;
...
if a < 2 or a = 2 then
    -- Do stuff
end if;
```

```vhdl
-- Not this
if a < 2 then
    -- Do stuff
end if;
...
if a < 3 then
    -- Do stuff
end if;
```

Listing 8: Two processes with logically equivalent conditions. In the left case, $a < 2$ can be reused in the second if-statement so only $a = 2$ needs to be allocated.

In Listing8, the behaviors of both code snippets are equivalent, but the left case uses less hardware resources despite using three arithmetic expressions compared to two. As mentioned in Section 4.3.3 equality checks are less resource intensive than $<$ operations. While the right case needs to allocate resources for two $<$ operations, the left needs to allocate one $<$ and one $=$ operation, which reduces resource utilization.

Another way to reduce the cost of expressions is to designate state variables for information which may otherwise be derived through expensive arithmetic or logic. As mentioned in Section 4.1.1, the management unit's instruction words have 6 reserved bits. The single-issue design has logic checking if the entire instruction word is zeroes as the control unit will clear the whole instruction word when the instruction was completed. For the minimized version of GANIMEDE, one of the reserved bits became a flag signaling whether the instruction is active or not. Setting and checking the one bit is significantly cheaper than operating on the entire word. While this was the most significant example of this technique, other instances of allocating dedicated signals for certain information did also yield positive results.

5

Results and Discussion

The results of this project are presented in three parts. First is raw throughput measurements of the interface without an IP core, also compared to ideal throughputs. Second is the synthesis results, showing the resource utilization of various designs of GANIMEDE. Third is simulation run time performance with the SHyLoC IP core, showing how close GANIMEDE performs to the ideal throughput for various problem image sizes. Tables containing all results can be found in appendix A.

5.1 Throughput results

The unhindered throughput was measured in accordance with the procedure specified in Section 3.2.1 and plotted for each GANIMEDE version as seen in Figure 5.1. In addition to the empirical measurements, the ideal throughputs from Table 3.1 are also plotted to give reference to the reader.

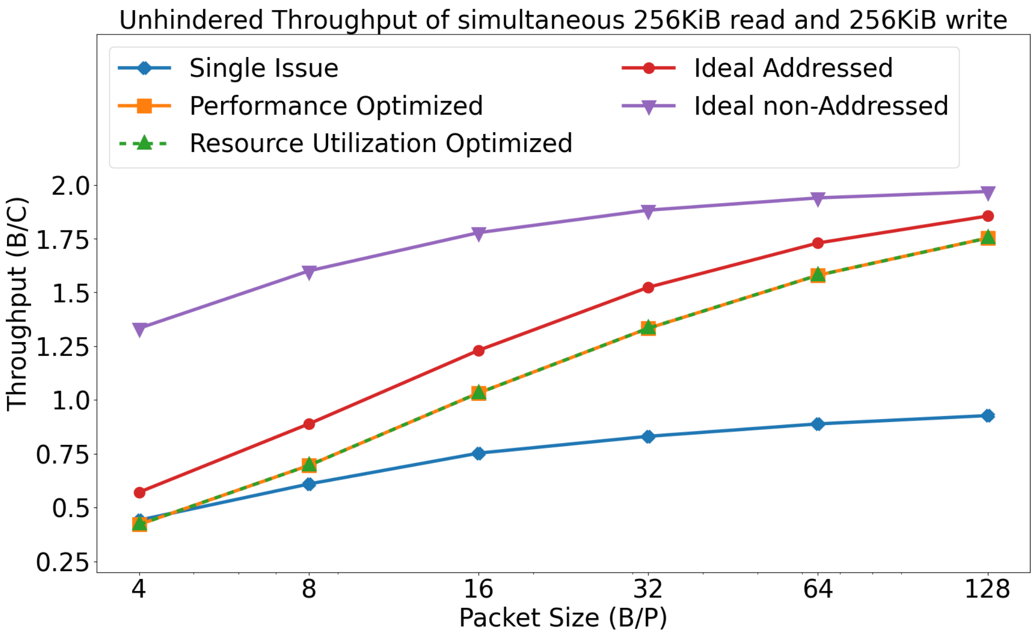


Figure 5.1: Throughput results compared between ideal calculations and empirical tests of different GANIMEDE versions. It is worth noting that the performance optimized and resource utilization optimized lines completely overlap.

The maximal throughput achieved for the three different GANIMEDE versions are found when the packet size is 128 bytes. Both the performance optimized and resource utilization optimized GANIMEDE achieve the same throughput of 1.7527 bytes per cycle, while the single issue GANIMEDE only achieved a maximal throughput of 0.9275 bytes per cycle.

Comparing the maximal throughput with both ideal cases the performance optimized and the resource optimized GANIMEDE achieved 94.5% and 89% of the throughputs seen in the addressed and non-addressed cases respectively. The single issue version's maximal throughput achieved 50% and 47.1% of the throughputs seen in the addressed and non-addressed cases respectively.

Utilization of the maximal throughput of 2 bytes per cycle is 87.6% for the performance optimized and resource utilization optimized GANIMEDE versions and 46.4% for the single issue version.

5.1.1 Discussion of Throughput Results

In the throughput results we compare not only with the ideal throughput with minimal protocol overheads, but also the ideal throughput with the worst case protocol overheads. As presented in Section 3.2.1 and discussed in Section 5.4, the addressing limitations imposed by the SoCBridge hinders GANIMEDE from issuing only one addressed access followed by an indefinite number of non-addressed accesses. When implementing IO pipelining in the way we have done, a natural consequence is that every accesses must be addressed. In Figure 4.7 we show the access patterns that IO pipelining implements, which are not compatible with the current feature set of the primary SoCBridge. Therefore, we compare GANIMEDE with the ideal addressed throughput in addition to the maximal throughput achievable with minimal header size.

The overheads which are improvable are in the time between finishing a write and receiving the start of the next read. This inefficiency appears due to solving a special case of the addressing limitation. The special case occurs in the command sequence "read, read, write", in which GANIMEDE sends an addressed read followed by a non-addressed read and lastly an addressed write. Due to the addressing limitation, the address of the write command overrides the stored address in the primary SoCBridge before the non-addressed read can complete. This exhibits a write after read (WAR) hazard in the primary SoCBridge, despite an otherwise correct command sequence. Since it is IP core dependent when a write will become possible, we solved this by enforcing a stricter requirement for sending writes. All previous conditions apply, with the addition that a read may not be in flight when sending the write command. This avoids the WAR hazard since no non-addressed read is pending when the write is sent. The consequence of this is that the write command will finish transmission multiple cycles after the read response has fully arrived, which is the main contributor to the removable overhead. One other way to avoid the WAR hazard is to disallow non-addressed commands all together. Doing so permanently prohibits the throughput from going above the ideal addressed throughput, even in cases where non-addressed commands could be sent safely in the current GANIMEDE implementation.

One very important limitation we set on this work in Section 3.1 is that of latency. Latency actually drastically affects the current version of GANIMEDE, as both the read and write instructions hinge on receiving responses to continue sending commands. Thus, the throughput would plummet if the latency between read request and response increased to more cycles than the transmission time of one write packet. To remedy this we believe that implementing a multi packet-in-flight optimization would be in order, which we describe in Section 4.2.3 and discuss further in Section 6.3.2.

5.2 Synthesis results

The measured resource utilization of all GANIMEDE versions are presented in Figure 5.2. The smallest number of FEs used was achieved by the single issue GANIMEDE, with a total of 1502 FEs, just about smaller than the resource optimized GANIMEDE with 1538 FEs.

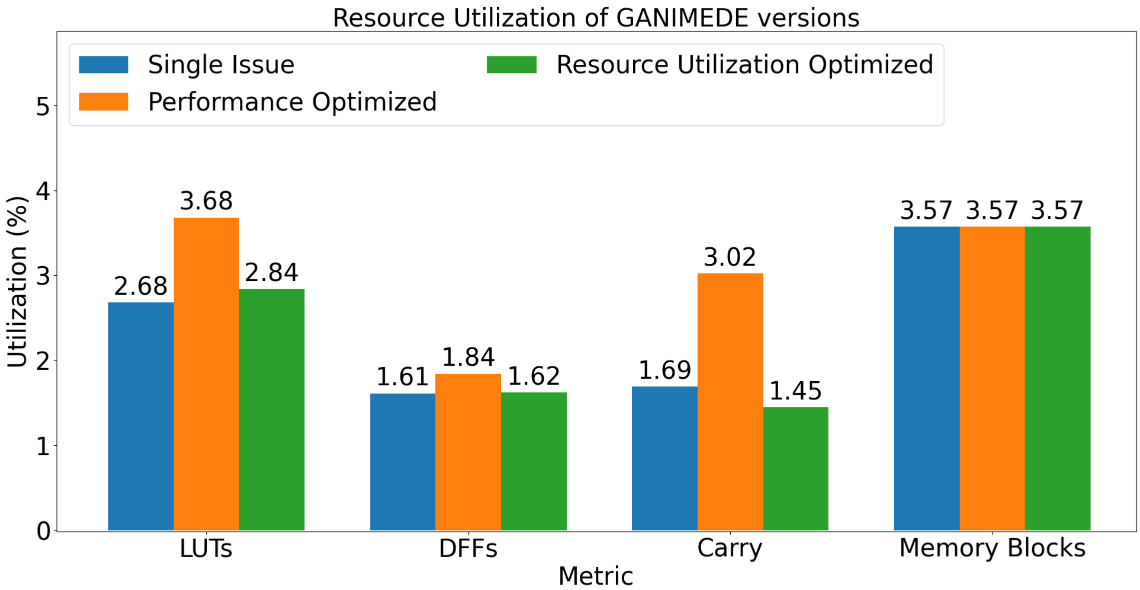


Figure 5.2: Resource utilization results of GANIMEDE versions targeting the GR765 eFPGA.

Comparing the resource optimized GANIMEDE with the performance optimized version, a reduction of 22.8%, 11.6% and 52.2% in LUTs, DFFs and Carries repsectively was achieved. Comparing the resource optimized GANIMEDE with the single issue version the LUT and DFF count increased by 5.9% and 0.6% respectively, while the Carry count decreased by 14.6%.

5.2.1 Discussion of Synthesis Results

An important thing to note about the synthesis results is that they are just that, synthesis results. Due to issues with the toolchain, GANIMEDE could not make it past the place step and thus the presented resource utilization is not final. This problem arose because the tool could not allocate certain resources connected to the technology mapped RAM blocks. Removing the buffers or targeting another one of NanoXplore's FPGAs resolved the problem but may not necessarily reflect the utilization of all of GANIMEDE on the actual target FPGA. Therefore, we deemed the synthesis results from the eFPGA28RH-30K to be the most honest. From our experience with the NanoXplore toolchain, the utilization after place and route was lower than after synthesis which likely makes the results presented in the report somewhat pessimistic.

5.3 SHyLoC results

The hyperspectral image compressed in this test has the ideal throughput of 0.4419 B/C when running the testbench provided by Frontgrade Gaisler. In Figure 5.3 the ideal throughput is visualized as a horizontal blue line. No version of GANIMEDE achieved the maximal throughput, but almost all of them achieved throughputs very close to the maximal throuhgput. The highest throughput achieved with a very small margin was by the resource utilization optimized GANIMEDE with a packet size of 16 bytes. The measured throughput was 0.4385 B/C, which is 99.2% of the ideal maximal throughput. More interestingly, the lowest throughputs for each GANIMEDE version was still more than 95% of the ideal, maximal throughput.

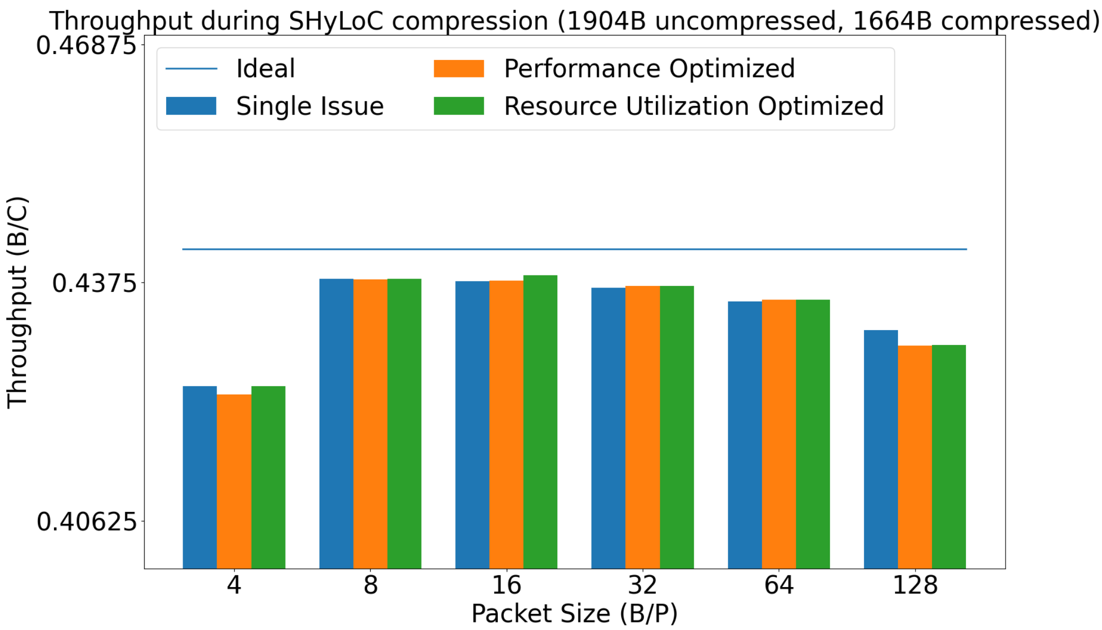


Figure 5.3: Throughput results in SHyLoC compared between ideal calculations and empirical tests of different GANIMEDE versions. The blue line is the unhindered maximal throughput for this image and is situated at a throughput of 0.4419 B/C

5.3.1 Discussion of SHyLoC Results

From the throughput results for SHyLoC presented in Figure 5.3, all designs achieved close to ideal throughput. While the highest throughput was achieved by the minimized design with a packet size of 16, the margin with which it surpassed the others is too slim to matter. Since all designs performed almost identically with the SHyLoC IP core, it did not demand enough data to truly stress test GANIMEDE. This combined with the synthesis results in Figure 5.2 speaks for the single issue design yielding the lowest resource utilization while still saturating the SHyLoC IP core. Worth noting is also that the single issue design is not optimized for resource utilization which means that it could be a very space efficient solution while still saturating SHyLoC if it were minimized.

5.4 SoCBridge Limitations

During the design process, we occasionally found ourselves limited by the SoCBridge interface GANIMEDE communicates with. We discovered three limitations during development, two of which are described in Section 2.4.2 since they affect the decisions made during the design phase of GANIMEDE. Below the three limitations are discussed further. First, the addressing limitation prohibits efficient use of the communication link during IO pipelining. This is the main limiter to increasing performance in GANIMEDE. If this limitation did not exist, the ideal throughput for both the performance optimized and resource utilization optimized GANIMEDE versions would be the ideal non-addressed throughput seen in Figure 5.1. Since those versions of GANIMEDE currently closely follow the throughput of the ideal addressed throughput, we predict that they would follow the ideal non-addressed throughput given that the addressing limitation remedy is removed from GANIMEDE. Adding separate address registers for reads and writes in the primary SoCBridge would solve this issue since that would remove the root cause of the addressing limitation.

Second, streaming long continuous data regardless if it is IO pipelined or not is not as efficient as it could be due to the increment limitation. The implications for data streaming applications is further overheads, since every time an instruction has handled 256 bytes of data a new address must be provided in the next command. Revising the formula for ideal throughput presented in Section 2.4.2 to include the overhead caused by the increment limitation yields the equation:

$$\frac{2N}{2(1 + 4\frac{N}{256}) + N}$$

The $4\frac{N}{256}$ term encodes the number of extra bytes sent "per packet" to account for the increment limitation. H is also replaced with 1 as this equation is only applicable in the non-addressed case. Comparing the utilization of non-addressed case and the increment limited case, we get the relative change shown in Table 5.1. We see that a 2-3% improvement can still be made if this limitation is removed in the SoCBridge.

Lastly, the primary SoCBridge has no support for non-unit stride accesses while

Table 5.1: Idealized utilization for the SoCBridge protocol for non-addressed streams and increment limited non-addressed streams.

Packet Size (B/P)	Inc Limited Utilization (u_l)	Non-addressed Utilization (u_h)	Relative Change ($\frac{u_h}{u_l}$)
128	0.95522	0.98460	+3.1%
64	0.94118	0.96970	+3.0%
32	0.91429	0.94120	+2.9%
16	0.86486	0.88890	+2.8%
8	0.78049	0.80000	+2.5%
4	0.65301	0.66666	+2.1%

there are many applications that would benefit from different sizes of stride. In matrix multiplication for example, reading all elements of a column would mean reading with a stride of the matrix's width. While this was not a problem in our case study it restricts the applicability of using the GR765's eFPGA as an HA. A solution to this issue would be to add support for this in the primary SoCBridge or to implement support for it in GANIMEDE. From a resource utilization perspective, non-unit stride access should be supported in the primary SoCBridge, since that entity is implemented in ASIC rather than the eFPGA fabric.

6

Research Questions and Conclusion

In the following Sections the research questions posed in the introduction are discussed. A conclusion is given which summarizes the most important findings of the thesis, followed by describing a few suggestions for future work.

6.1 Addressing Research Questions

While all research questions will be answered in our conclusions in Section 6.2, some require a bit of background and discussion as they can not simply be derived of the results. The aforementioned background and discussion will be provided in the following Sections.

6.1.1 Desirable properties in a stream engine optimized for spacecraft SoCs

During the development process, a multitude of desirable features were identified. The process of identification varied for the different features. Some were found as solutions to difficulties with testing, others were afterthoughts on how implementation could be simplified.

IO pipelining - For larger problem sizes, IO pipelining allows the HA to run uninterrupted as data can flow in and out simultaneously. This feature was implemented and yielded high throughput, especially at larger packet sizes as seen in Figure 5.1.

Interrupt signaling to CPU - In its current state, GANIMEDE has no way of directly communicating with a CPU by the use of interrupts. Adding this feature would be highly desirable as it adds the final building block of an asynchronous computation - the callback.

Memory mapping pass-through - Many IP cores make use of run time configuration to start computation. If hidden behind the current abstraction layer provided by GANIMEDE, there exists no easy way to access those addresses. Our suggested solution is to add a special pass-through range of addresses inside the management unit which allows for bridging between IP core on the eFPGA and the external system bus.

Instruction queuing and scheduling - Enabling queuing of multiple instructions in a row would allow GANIMEDE to operate on multiple data streams over time without processor intervention. This feature is desirable in cases where the CPU would like to submit work to the eFPGA in batches. Furthermore, dynamic inter-instruction rescheduling could be employed such that the latency between writing the last piece of data and reading first piece of data in the next instruction to be hidden.

Different kinds of accesses types - GANIMEDE currently supports one type of memory access: the unit stride one-shot. It may be desirable to allow for non unit stride accesses, or for an unending continuous access to or from a block of memory, similar to a circular buffer. There exists initial support in the form of reserved bits in the GANIMEDE instruction word.

Choosing features at compile time - If the HA one is interfacing with is the bottleneck for throughput then it is not always necessary for the stream engine to maximize its own throughput. Therefore, it would be beneficial to be able to easily decide which performance optimization to synthesize at compile time using if-generate statements for example. This way, unnecessary throughput can be traded for lower resource utilization.

In addition, due to GANIMEDE residing on the eFPGA, extremely specialized IP-Core specific optimizations may be added to further increase performance. This is one of the major benefits to using FPGAs over ASICs, and should therefore not go underutilized.

6.1.2 What Parts Should Be on SoC instead of FPGA

By conducting this case study, two issues arose which informed the two following resource saving suggestions. Though they come with their own drawbacks, we believe them to be worthwhile improvements.

First, GANIMEDE currently has to keep track of the amount of data present in the read buffer and deny instructions which would cause the buffer to overflow and drop data. This uses unnecessary logic when a simpler solution exists. Preferably, the interface with which GANIMEDE communicates – in this case SoCBridge – would have a backpressure signal to halt data transfer, like the ready signal in a FIFO interface. Letting the protocol implement backpressure signaling on ASIC rather than in FPGA would simplify parts of the driver and translator inside GANIMEDE, thus saving on precious FPGA resources.

Second, when designing GANIMEDE, one issue that arose was how dealing with instructions arriving on the same interface as data. This led to the SoCBridge drivers FSM being fairly complex, needlessly so compared to how it would look if it were to only handle data. It would be quite beneficial to instead have an external deserializer which takes the bytes of data of an instruction to the SoCBridge and turns it into a memory interface, with parallel data and address signals along with a write/read enable signal once data and address have arrived.

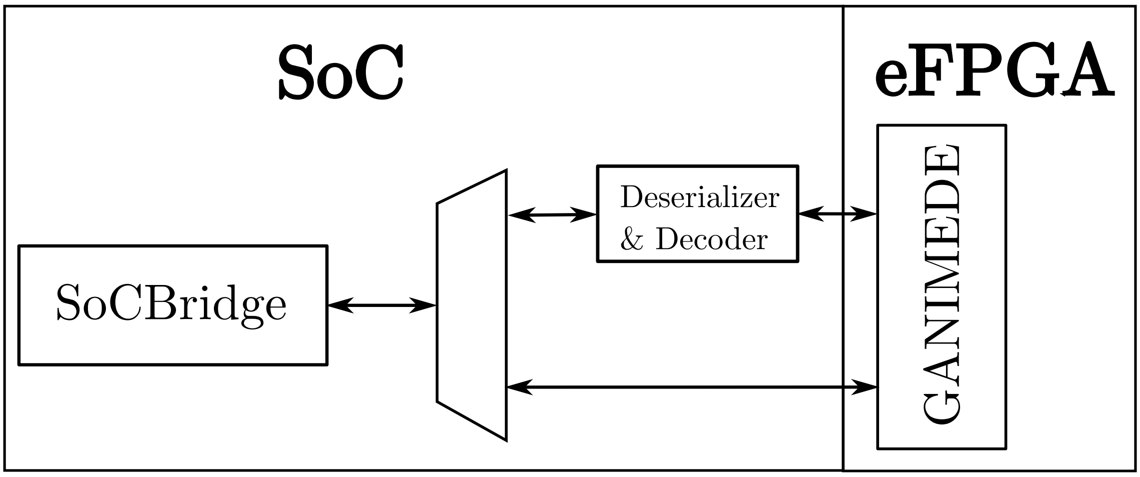


Figure 6.1: Suggested configuration of interface between eFPGA and the SoC.

As seen in Figure 6.1, the proposed layout gives the memory mapped instructions from the CPU a separate path to the data stream. This way, GANIMEDE – or any other interface for that matter – gets a primary and secondary interface, namely the lower and upper link in the image respectively. This greatly simplifies any design residing in the eFPGA as the deserializer and decoder can handle requests from the CPU while only GANIMEDEs own requests are handled through the other link. Simplifying the driver in this way would lead to both a simpler description of what a driver is responsible of and to lower resource utilization. Implementing this comes at the cost of using many more IO pins on the eFPGA.

It should be noted that this model is generally transferable to different protocols, as many bus-like architectures make use of the requester-responder paradigm. To generalize the proposed change, the requester interface would be presented to the eFPGA directly, while the responder interface would partly be implemented already on the SoC.

6.2 Conclusion

In this thesis, a stream engine called GANIMEDE was developed. We provide a detailed description of GANIMEDE, the reasoning behind the design decisions and collection of improvements still possible to add. In an unhindered throughput measurement, the resource optimized GANIMEDE achieved a link utilization of 94.5% given protocol overheads while only occupying 2.2% (1538 functional elements) of the available hardware resources.

The maximal throughput measured during computation within the SHyLoC IP Core was achieved by the resource utilization optimized version of GANIMEDE with a packet size of 16B. From a resource perspective, it is better to utilize the single issue GANIMEDE at 8B per packet as it achieved a very similar throughput. The resource optimized version measured a throughput 99.2% of the ideal whereas the single issue version measured 99.1%.

In a stream engine, it is desirable to support, among other features, IO pipelining, memory mapped pass-through and interrupt signaling. Making use of both the receiving and transmitting side of the communication protocol is paramount to ensuring high utilization.

Protocols which aim to support data stream applications for eFPGAs should implement as much as possible outside of the eFPGA. This entails implementing parts which would go in the eFPGA on the SoC in the ASIC regardless of protocol, as command decoding of requests targeting the eFPGA can be translated into memory-like instructions instead. Implementing this reduces the total utilization as an essential, unremovable part of the protocol is implemented in ASIC rather than FPGA.

6.3 Future Work

After the completion of this thesis we have a few different directions future work on the subject could follow. Below we present three different suggested research projects, one to improve the usability of GANIMEDE, one to improve the throughput of GANIMEDE and one to perform a field survey.

6.3.1 GANIMEDE improvements

If development on GANIMEDE were to continue, many improvements are still of interest to add. Below, a bullet point style list briefly summarizes what we would have done for each improvement. Note that these are in addition to the properties presented in Section 6.1.1

Measure and characterize latency - Latency is not investigated in this thesis, which is one of its flaws as it could impose large performance penalties due to the architecture of GANIMEDE. To resolve the potential performance issues, the theorized multi in-flight packet improvement would be implemented.

Measure runtime and power of GANIMEDE in hardware - Running tests on GANIMEDE in hardware would allow for realistic memory latency and power usage measurements. To run GANIMEDE in hardware we would use an existing FPGA and only report relative measurements, comparing the system with no GANIMEDE instance with the different versions of GANIMEDE. Additionally, if integrated with a SoC, realistic memory bus loads could be imposed on the memory subsystem by the CPU to more precisely evaluate GANIMEDE in its targeted environment.

Investigate and implement more optimizations - As we report and discuss in the previous Sections, GANIMEDE achieves a high throughput given the current circumstances but only about 89% of the available throughput given minimal protocol overheads. This is something we see as improvable, not only by solving the challenges brought on by the SoCBridge limitations, but also by improving GANIMEDE further. We suggest implementing the proposed multi in-flight packet improvement to begin with, but continue following the same methodology presented in Section 3.2.1.

Develop a software library to simplify use of GANIMEDE - As it stands, making use of GANIMEDE in an actual system entails making writes to special addresses in memory in a not-so-user friendly way. A simple yet powerful improvement would be to develop a C library to simplify initialization and use of GANIMEDE.

6.3.2 Multi interface implementation and optimization

One piece of work we see as good next step for the development and exploration of resource optimized stream engines is to investigate the performance impacts and optimization possibilities of using multiple hardware interfaces at the same time.

We imagine the study would identify a set of approaches and evaluate their performance benefits compared to the associated resource utilization for each. Examples of approaches we can think of to investigate are address space mapped interface selection, instruction word indexed interface selection and dynamically scheduled selection.

6.3.3 Resource optimized stream engine survey

To the best of our knowledge, no survey consolidating the field of resource optimized stream engines exists and as such presents a large gap in understanding the unsolved problems in the field. If we found such a work it would have given us a more clear direction and focus in this thesis.

To conduct the survey, both resource optimized DMA controllers and high performance DMA engines in conjunction with stream engine research would be a great start to find relevant research. Similarities between DMA controllers and stream engines, like the one we identified and used in this thesis, could be found and exploited for further performance or resource utilization improvements.

Bibliography

[1] L. Diana and P. Dini, "Review on hardware devices and software techniques enabling neural network inference onboard satellites," *Remote Sensing*, vol. 16, no. 21, 2024, ISSN: 2072-4292. DOI: 10.3390/rs16213957. [Online]. Available: https://www.mdpi.com/2072-4292/16/21/3957.

[2] A. D. George and C. M. Wilson, "Onboard processing with hybrid and reconfigurable computing on small satellites," *Proceedings of the IEEE*, vol. 106, no. 3, pp. 458–470, 2018. DOI: 10.1109/JPROC.2018.2802438.

[3] S. Lopez, T. Vladimirova, C. Gonzalez, J. Resano, D. Mozos, and A. Plaza, "The promise of reconfigurable computing for hyperspectral imaging onboard systems: A review and trends," *Proceedings of the IEEE*, vol. 101, no. 3, pp. 698–722, 2013. DOI: 10.1109/JPROC.2012.2231391.

[4] R. B. Gardenyes, "Trends and patterns in asic and fpga use in space missions and impact in technology roadmaps of the european space agency," Master's Thesis, Delft Univeristy of Technology, 2012.

[5] Frontgrade Gaisler. "GR765 core users manual." (2025), [Online]. Available: https://www.gaisler.com/products/gr765 (visited on 05/26/2025).

[6] Microchip. "Radiation Tolerant PolarFire SoC FPGA." (2020), [Online]. Available: https://www.microchip.com/en-us/products/fpgas-and-plds/radiation-tolerant-fpgas/rt-polarfire-soc (visited on 05/26/2025).

[7] NanoXplore. "NanoXplore NG-Ultra product page." (Unknown), [Online]. Available: https://nanoxplore.com/index.php/product/ng-ultra/ (visited on 05/26/2025).

[8] M. J. Flynn, O. Pell, and O. Mencer, "Dataflow supercomputing," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 1–3. DOI: 10.1109/FPL.2012.6339170.

[9] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 416–429. DOI: 10.1145/3079856.3080255.

[10] L. Santos, A. Gómez, and R. Sarmiento, "Implementation of ccsds standards for lossless multispectral and hyperspectral satellite image compression," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 56, no. 2, pp. 1120–1138, 2020. DOI: 10.1109/TAES.2019.2929971.

[11] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," in *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, ser. FPGA '06, Monterey, California, USA: As-

sociation for Computing Machinery, 2006, pp. 21–30, ISBN: 1595932925. DOI: 10.1145/1117201.1117205. [Online]. Available: https://doi.org/10.1145/1117201.1117205.

[12] A. Sönnichsen, "Chapter 6: Militarization and securitization of outer space," in *A Research Agenda for Space Policy*, Cheltenham, UK: Edward Elgar Publishing, 2021, ISBN: 9781800374737. DOI: 10.4337/9781800374744.00014. [Online]. Available: https://www.elgaronline.com/view/edcoll/9781800374737/9781800374737.00014.xml.

[13] J. D. Rummel, "Planetary exploration in the time of astrobiology: Protecting against biological contamination," *Proceedings of the National Academy of Sciences*, vol. 98, no. 5, pp. 2128–2131, 2001. DOI: 10.1073/pnas.061021398. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.061021398. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.061021398.

[14] W. Dally and B. Towles, *Principles and Practices of Interconnection Network*. Morgan Kaufmann Publishers, 2004, p. 1.

[15] M. Dubois, M. Annavaram, and P. Stenström, *Parallel Computer Organization and Design*. Cambridge University Press, 2012, pp. 193–196.

[16] D. M. Fleetwood, P. S. Winokur, and P. E. Dodd, "An overview of radiation effects on electronics in the space telecommunications environment," *Microelectronics Reliability*, vol. 40, no. 1, pp. 17–26, 2000, ISSN: 0026-2714. DOI: https://doi.org/10.1016/S0026-2714(99)00225-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026271499002255.

[17] G. C. Messenger, "Radiation hardening," 2020. DOI: 10.1036/1097-8542.566850. [Online]. Available: https://www.accessscience.com/content/article/a566850.

[18] Alp Kilic, Kassem Chouayakh, Julia Damal, Benoit Leconte. "Rad hard FPGAs." (2022), [Online]. Available: https://nanoxplore-wiki.atlassian.net/wiki/spaces/NAN/pages/10256387/Rad+Hard+FPGAs (visited on 05/26/2025).

[19] T. Calin, M. Nicolaidis, and R. Velazco, "Upset hardened memory design for submicron cmos technology," *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, pp. 2874–2878, 1996. DOI: 10.1109/23.556880.

[20] F. Faccio, "Design hardening methodologies for asics," in *Radiation Effects on Embedded Systems*, R. VELAZCO, P. FOUILLAT, and R. REIS, Eds. Dordrecht: Springer Netherlands, 2007, pp. 143–160, ISBN: 978-1-4020-5646-8. DOI: 10.1007/978-1-4020-5646-8_7. [Online]. Available: https://doi.org/10.1007/978-1-4020-5646-8_7.

[21] S. Niranjan and J. Frenzel, "A comparison of fault-tolerant state machine architectures for space-borne electronics," *IEEE Transactions on Reliability*, vol. 45, no. 1, pp. 109–113, 1996. DOI: 10.1109/24.488925.

[22] AMD. "Virtex UltraScale+ VU19P." (2020), [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/products/adaptive-socs-and-fpgas/fpga/ultrascale-plus/virtex-ultrascale-plus-vu19p-product-brief.pdf (visited on 05/26/2025).

[23] AMD. "Xilinx FPGAs product page." (2025), [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga.html#overview (visited on 05/28/2025).
[24] ARM. "AMBA AHB protocol specification." (2021), [Online]. Available: https://developer.arm.com/documentation/ihi0033/c/ (visited on 05/26/2025).
[25] ARM. "AMBA APB protocol specification." (2023), [Online]. Available: https://developer.arm.com/documentation/ihi0024/e/?lang=en (visited on 05/26/2025).
[26] Frontgrade Gaisler. "GRLIB IP core users manual." (2024), [Online]. Available: https://www.gaisler.com/grlib-ip-library (visited on 03/25/2024).
[27] S. Lopez, T. Vladimirova, C. Gonzalez, J. Resano, D. Mozos, and A. Plaza, "The promise of reconfigurable computing for hyperspectral imaging onboard systems: A review and trends," *Proceedings of the IEEE*, vol. 101, no. 3, pp. 698–722, 2013. DOI: 10.1109/JPROC.2012.2231391.
[28] T. Benz, M. Rogenmoser, P. Scheffler, *et al.*, "A high-performance, energy-efficient modular dma engine architecture," *IEEE Transactions on Computers*, vol. 73, no. 1, pp. 263–277, 2024. DOI: 10.1109/TC.2023.3329930.
[29] S. D. Automation. "VHDL Reference Manual." (1997), [Online]. Available: https://staff.fysik.su.se/~silver/digsyst/vhdl_ref.pdf (visited on 05/29/2025).

A

Appendix 1: Result Data

Table A.1: Simulating GANIMEDE in an idealized test of a 256KiB read and a 256 KiB write at the same time. Two configurations are measured, one utilizing only IO pipelining and the other utilizing both IO pipelining and tucking. In this test buffer space was always available for reads and data was always present for writes. Throughput is measured as an average, summed for both reads and writes. Zero latency to access the memory.

GANIMEDE throughput test (256 KiB read, 256 KiB write)				
Single Issue	**Packet size (bytes/packet)**	**Throughput (bytes/cycle)**	**Run Time (cycles)**	**Run Time (ns)**
	128	0.9275	565 252	28 262 600
	64	0.8889	589 828	29 491 400
	32	0.8312	630 788	31 539 400
	16	0.7529	696 325	34 816 250
	8	0.6095	860 164	43 008 200
	4	0.4414	1 187 844	59 392 200
IO Pipelining	**Packet size (bytes/packet)**	**Throughput (bytes/cycle)**	**Run Time (cycles)**	**Run Time (ns)**
	128	1.7527	299 138	14 956 900
	64	1.5799	331 841	16 592 050
	32	1.3333	393 219	19 660 950
	16	1.0323	507 907	25 395 250
	8	0.6957	753 666	37 683 300
	4	0.4211	1 245 186	62 259 300
Minimized IO Pipelining	**Packet size (bytes/packet)**	**Throughput (bytes/cycle)**	**Run Time (cycles)**	**Run Time (ns)**
	128	1.7527	299 138	14 956 900
	64	1.5799	331 841	16 592 050
	32	1.3333	393 219	19 660 950
	16	1.0323	507 907	25 395 350
	8	0.6957	753 643	37 682 150
	4	0.4211	1 245 167	62 258 350

Table A.2: Resource utilization from synthesis of GANIMEDEs initial design

Resource	**Available**	**Total used**	**% of available**
4-LUT	32640	874	2.68%
DFF	30720	496	1.61%
Carry	7680	130	1.69%
Memory Blocks (48Kib)	40	2	3.57%

Table A.3: Resource utilization from synthesis of GANIMEDE with all performance optimizations

Resource	**Available**	**Total used**	**% of available**
4-LUT	32640	1200	3.68%
DFF	30720	565	1.84%
Carry	7680	232	3.02%
Memory Blocks (48Kib)	40	2	3.57%

Table A.4: Resource utilization from synthesis of the fully optimized version of GANIMEDE

Resource	**Available**	**Total used**	**% of available**
4-LUT	32640	926	2.84%
DFF	30720	499	1.62%
Carry	7680	111	1.45%
Memory Blocks (48Kib)	40	2	3.57%

Table A.5: Simulating GANIMEDE running SHyLoC with test case 40 (1904B payload, 1664B result)

SHyLoC image compression test (1904B image, 1664B compressed)				
Ideal	**Packet size (bytes/packet)**	**Throughput (bytes/cycle)**	**Run Time (cycles)**	**Run Time (ns)**
	N/A	0.4419	8075	403750
Single Issue	**Packet size (bytes/packet)**	**Throughput (bytes/cycle)**	**Run Time (cycles)**	**Run Time (ns)**
	128	0.4313	8 273	413 650
	64	0.4350	8 203	410 150
	32	0.4368	8 169	408 450
	16	0.4377	8 152	407 600
	8	0.4380	8 146	407 300
	4	0.4239	8 417	420 850
IO Pipelining	**Packet size (bytes/packet)**	**Throughput (bytes/cycle)**	**Run Time (cycles)**	**Run Time (ns)**
	128	0.4292	8 313	415 650
	64	0.4353	8 197	409 850
	32	0.4371	8 163	408 150
	16	0.4378	8 150	407 500
	8	0.4379	8 148	407 400
	4	0.4228	8 438	421 900
Minimized IO Pipelining	**Packet size (bytes/packet)**	**Throughput (bytes/cycle)**	**Run Time (cycles)**	**Run Time (ns)**
	128	0.4293	8 312	415 600
	64	0.4353	8 196	409 800
	32	0.4371	8 162	408 100
	16	0.4385	8 136	406 800
	8	0.4380	8 146	407 300
	4	0.4239	8 418	420 900