Delft University of Technology

Electrical Engineering, Mathematics, and Computer Science
Computer Engineering

Feeding High-Bandwidth Streaming-Based FPGA Accelerators

Thesis by:

Yvo Thomas Bernard Mulder

Advisor:

Prof. Dr. H.P. Hofstee

Committee:

Chair:
Prof. Dr. H.P. Hofstee

Members:
Dr. Ir. Z. Al-Ars
Dr. Ir. R. van Leuken

Feeding High-Bandwidth Streaming-Based FPGA Accelerators

Thesis

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

by

Yvo Thomas Bernard Mulder
born in Utrecht, The Netherlands

to be defended publicly on January 29, 2018 at 15:00.

CE-MS-2018-05
ISBN: 978-94-6186-886-2
Computer Engineering
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4,
2628 CD, Delft
The Netherlands

Abstract

A new class of accelerator interfaces has significant implications on system architecture. An order of magnitude more bandwidth forces us to reconsider FPGA design. OpenCAPI is a new interconnect standard that enables attaching FPGAs coherently to a high-bandwidth, low-latency interface. Keeping up with this bandwidth poses new challenges for the design of accelerators, and the logic feeding them.

This thesis is conducted as part of a group project, where three other master students investigate database operator accelerators. This thesis focuses on the logic to feed the accelerators, by designing a reconfigurable multi-stream buffer architecture. By generalizing across multiple common streaming-like accelerator access patterns, an interface consisting of multiple read ports with a smaller than cache line granularity is desired. At the same time, multiple read ports are allowed to request any stream, including reading across a cache line boundary.

The proposed architecture exploits different memory primitives available on the latest generation of Xilinx FPGAs. By combining a traditional multi-read port approach for data duplication with a second level of buffering, a hierarchy typically found in caches, an architecture is proposed which can supply data from 64 streams to eight read ports without any access pattern restrictions.

A correct-by-construction design methodology was used to simplify the validation of the design and to speedup the implementation phase. At the same time, the design methodology is documented and examples are provided for ease of adoption. With the design methodology, the proposed architecture has been implemented and is accompanied by a validation framework.

Various configurations of the multi-stream buffer have been tested. Configurations up to 64 streams with four read ports meet timing with an AFU request-to-response latency of five cycles. The largest configuration with 64 streams and eight read ports fails timing. Limiting factors are the inherent architecture of FPGAs, where memories are physically located in specific columns. This makes extracting data complex, especially at the target frequencies of 200 MHz and 400 MHz. Wires are scattered across the FPGA and wire delay becomes dominant.

FPGA design at increasing bandwidths requires new design approaches. Synthesis results are no guarantee for the implemented design, and depending on the design size, could indicate a very optimistic operating frequency. Therefore, designing accelerators to keep up with an order of magnitude more bandwidth compared to the current state-of-the-art is complex, and requires carefully thought out accelerator cores, combined with an interface capable of feeding it.

Preface

This thesis report marks the end of this project, on which I have worked for a year. The year started at the IBM Austin Research Lab, after Peter Hofstee invited me to work with him on emerging coherently attached FPGA accelerators. During my six months in Austin, Peter was always ready to help and chat. Also in the period after Austin was Peter always available. Peter, it has been a very pleasant journey and I sincerely hope our paths cross again. I can safely say that you not only have been a tremendous supervisor, but also a good friend.

Because the project is in the field of FPGAs, I had many interesting discussions with Andrew Martin. Andrew is a research staff member and developed a ready-valid design methodology for FPGA design. Andy, I would like to thank you for your support during the design and implementation phase. Your methodology is the missing link for FPGA design.

Dorus, Eric, Jeremy, and Jinho, the three o'clock stretch session must live on, but in different countries and time zones. I enjoyed my time in Austin very much and that is thanks to you.

I would like to thank the Universiteitsfonds for providing funding for the six months I spent in Austin. Without this support, it would have been difficult to have had this experience.

I would like to thank my parents for their life-long support and faith in me.

Finally, I would like to thank Fjóla for always being there for me when I needed her the most and making me a better person every day.

Contents

List of Figures

List of Tables

Listings

Revision Log

The revision log keeps track of all significant changes that have been made to this thesis since the initial public release on the 29th of January 2018. The most recent version of this document can be found at `https://github.com/ytbmulder/msc-thesis`.

Version	Revision Date	Description
v1.0.0	29 January 2018	Initial public release.
v1.0.1	26 April 2018	Addition of this revision log.

Chapter 1

Introduction

In recent years, data centers have been forced to embrace heterogeneous system architectures. As microprocessor technology and design is not able to deliver cost per performance improvements in line with historical rates, deployment of hardware acceleration will become commonplace.

Trends in system architecture show that interconnect bandwidth is significantly increasing in order to provide data centers with the connectivity and ability to attach accelerators for any workload. Traditionally FPGAs (Field Programmable Gate Arrays) lacked bandwidth and programmability and could only be used for the most computationally intensive tasks, but recent advancements target these shortcomings and accelerate adoption. While the change in programming models is an interesting field of study, this thesis focuses on the emerging FPGAs with main memory class bandwidth enabled by such advancements. This change requires re-evaluation of our current design approaches and methodologies for accelerators. It does not only pose a challenge for accelerator design, but also for providing them with data. New interface logic is required to feed the accelerators, and new accelerator cores are needed, at least in those cases where the problem is not trivially parallel.

A common memory access pattern for FPGA accelerators is streaming and examples include content delivery, cryptography and databases. An interface capable of handling multiple streams is desired because workloads exist that inherently use multiple streams or for which multiple single-stream engines are required to work in parallel in order to exploit all of the available bandwidth. Besides that, sustaining throughput is more difficult when using a single stream. Multiple streams can more easily keep the interconnect fully utilized since concurrent requests can be made. Partitioning the compute is left to the accelerator designer.

Previous work such as the Streaming Framework [11, 12] and SNAP (Storage, Networking, and Analytics Programming) Framework [13, 14] for CAPI (Coherent Accelerator Processor Interface) 1.0 will not suffice, since these frameworks are not capable of handling this class of bandwidth, nor the number of streams. The frameworks target a bandwidth that is an order of magnitude smaller compared to OpenCAPI. SNAP also uses the coherent cache present in CAPI 1.0, but not present in OpenCAPI. Therefore a direct port is not trivial.

1.1 Thesis Aim

To combine both efforts of re-evaluating the interface and accelerator engine design, this thesis is part of a larger project, in collaboration with three other master students [15, 16, 17], that focuses on accelerator core design. A harmonized effort is made by studying a re-emerging and inherently multi-stream workload: database operators. Preliminary findings have been presented at the H^2RC 2017 workshop [18].

Database systems have been looking for architectures that achieve a high bandwidth to access the required data. FPGA acceleration was used in the past, but a recent trend is the usage of in-memory databases. In such systems, the database is located in host memory instead of on flash or mechanical storage. Now that interfaces like OpenCAPI are approaching host memory-like bandwidths and have coherent memory access, accelerating database operators using FPGAs by means of low latency memory access becomes relevant again.

The three other master students are also under supervision of Prof. Dr. H. Peter Hofstee. They will study three different multi-stream accelerators for database operators. The studied operators are: Decompress-Filter, Merge-Sort and Hash-Join.

The aim of this thesis is to study the implications of emerging high-bandwidth interconnects for FPGA accelerators, but more importantly their interface. Feeding accelerators with data and keeping up with the increased bandwidth is challenging. Prerequisite are multiple streams and read ports with less than cache line granularity. Providing such an interface is not trivial at this bandwidth. Therefore, the focus is on getting the data to the FPGA. Writing results back to the host is left as future work.

1.2 Thesis Contributions

The aims are to generalize across several common FPGA memory access patterns and to design and implement an interface that can be generally applicable to current and future high-bandwidth interconnects. Supplying a general interface to the FPGA designer will improve adoption and accelerate the design cycle. The contributions made in this thesis can be summarized as follows.

- A study of a new class of accelerator interfaces, and a more detailed overview of OpenCAPI (the first of its kind).
- Re-evaluation of design methodologies for high-bandwidth attached FPGAs.
- Provide documentation and examples for a delay-insensitive design methodology provided by Andrew K. Martin.
- A multi-level buffer architecture proposal and implementation, aware of fixed-size memory resources found on FPGAs, by exploiting features of different memory primitives and state-of-the-art memory resources.
- Improve adoption of high-bandwidth interconnect, with a special interest in OpenCAPI, attached streaming-based accelerators by providing a generalized and reconfigurable interface. This interface supports multiple streams and access patterns in order to be widely used while keeping up with the bandwidth.

1.3 Thesis Organization

Chapter 2 describes technology trends with respect to system level bandwidth requirements and interconnect standards. Chapter 3 takes a brief look at the state-of-the-art interconnect standards and what sets them apart. Chapter 4 characterizes OpenCAPI, the POWER9 processor and future OpenCAPI-compatible Xilinx FPGAs. Chapter 5 takes a look at common accelerator memory access patterns and shows naive buffer designs for full-utilization of the available interconnect bandwidth. Chapter 6 introduces the Delay Insensitive Cell Library which accelerates FPGA design by providing cells with a built-in ready-valid protocol. Chapter 7 combines the previous chapters and motivates the design choices made for a multi-stream buffer and highlights the essential modules of the design. Chapter 8 shows the validation setup, performance, and implementation results, and Chapter 9 concludes the thesis.

Chapter 2

Technology Trends

Data center workloads have increased and diversified over the last decade. These changes are driven by emerging workloads such as artificial intelligence, big data and machine learning, but also by an increased demand of cloud services and a shift of compute from the edge of the network to the data center due to an increase in mobile devices. The following itemization summarizes different classes of workloads and examples [19, 20, 21].

- Analytics: Big Data, High-Frequency Trading, and In-Memory Database.
- Cloud: Search, Virtualization, and Web Servers.
- Communication: Packet Processing and Virtual Switching.
- High Performance Computing: Artificial Intelligence, Genomics and Machine Learning.
- Security: Encryption and Decryption.
- Storage: Compression and Deduplication.

These changes in data center workloads demand not only more and faster resources (such as cooling, network and servers), but also a diversification of compute resources that can be dynamically tailored to the workload in question. Traditionally, servers consist of a fixed set of resources such as compute, memory, storage and I/O and are aggregated into a pool. Workloads are then scheduled on one or multiple pools. This architecture frequently results in under-utilization of resources due to a drastic real-time adjustment in available resources for specific workloads. This results in reduced power efficiency, or performance per Watt [22]. The most important metric when building a new data center is the Total Cost of Ownership (TCO): the cost of purchasing and installing the hardware plus the cost to operate and maintain the data center over time. The electricity costs are roughly 15% of the TCO [23] and power consumption is becoming one of the most import metrics for data center operation and future hardware investments. Systems have to offer performance and power efficiency, while in the past performance was the dominant driver behind new investments. This makes Application Specific Integrated Circuits (ASICs), FPGAs and Graphics Processing Units (GPUs) more and more interesting due to their performance-to-Watt ratio.

2.1 Acceleration in the Data Center

Traditionally, Moore's Law (which predicted exponential growth in the number of transistors on a chip) correlated well with single processor performance due to the increase in operating frequency that accompanied device scaling. The increase in the number of transistors per chip at a constant cost means that transistors become smaller and, if all dimensions are scaled, are able to switch at higher speeds. This explains the constant increase in frequency of processors until roughly 2006.

2.1.1 Dennard Scaling

In 1974, Dennard observed that the necessary current and voltage scale with transistor shrinking. This observation is known as Dennard scaling. Therefore, power consumption is proportional to transistor area. The total power consumed is the sum of dynamic and static power. Dynamic power is consumed by charging capacitors in the circuit and is shown in Equation 2.1. Static power is consumed when the circuit is in quiescence.

$$P = \alpha \times C \times F \times V^2, \text{where} \tag{2.1}$$

- P is the dynamic power,
- α is the percentage of time the circuit switches,
- C is the sum of gate and wiring capacitance,
- F is the frequency at which the circuit operates, and
- V is the operating voltage of the circuit.

However, Equation 2.1 involves simplified assumptions, because in the 1970's sub-threshold leakage was playing a relatively small role with respect to total power consumption. After several decades, sub-threshold leakage constrains further scaling of the threshold voltage and therefore also operating voltage. Due to leakage constraints, gate oxide scaling has also been affected. This prevents voltages from scaling as in the past, and thus starts to play a significant part in the total chip power. These factors limit the operating frequency of circuits [24]. Because voltages no longer scale, shrinking transistors now leads to power density increases. Insufficient cooling capacity resulted in hitting the so-called Power Wall in 2006 that limits processor frequency to around 4 GHz [25].

2.1.2 Homogeneous Multi-Core Systems

Since 2006, after hitting the Power Wall, and well before 2006 for server processors, Central Processing Units (CPUs) started to have multiple cores that work in parallel. Cores can be located within the same package, across multiple sockets, or across multiple systems. However, a workload across multiple homogeneous cores scales only as well as the portion of the workload that can be parallelized. Amdahl's Law [26], shown in Equation 2.2, formulates the theoretical speedup of a workload when a portion of the system is improved and encapsulates this notion. The speedup is limited by the fraction of the task that does not benefit from the improvement.

For example, if a task consists of one portion that can be parallelized and one portion that cannot, eventually the total latency is always bounded by the purely sequential portion.

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}, \text{where} \tag{2.2}$$

- $S_{latency}$ is the theoretical speedup of the workload at a fixed workload size, defined in latency,
- s is the speedup of the portion of the workload that benefits from the improvement, and
- p is the fraction of the execution time that s initially occupied in the workload.

A refinement to Amdahl's Law is Gustafson's Law [27]. This law takes into account that larger problems typically have a smaller sequential component.
The moment the energy to switch a single transistor does no longer decrease in a next process generation, using twice as many transistors to build twice the number of cores, only to have to turn off half of them because there is not sufficient power to feed them, is the end of the homogeneous multi-core era.

2.1.3 Heterogeneous Multi-Core Systems

By applying a simple hardware model to Amdahl's Law, it is shown that a system with asymmetric cores can always achieve a higher maximum speedup than a system with homogeneous cores [28]. Typically, FPGAs and GPUs are used to improve parallel workloads, but in rare cases can also improve sequential workloads.

Arguably the most commonly used heterogeneous system in a nowadays' data center is a CPU-GPU system. In comparison to a CPU, a GPU consists of several thousands of cores operating at a frequency of roughly 1 GHz. Dividing the workload between both compute resources is done manually, but yields a higher speedup compared to parallelization on CPUs only. Programming can be done in various languages but memory transfers between the CPU and GPU have to be invoked manually.

Similar to a homogeneous multi-core architecture found in GPUs, Intel released the Xeon Phi coprocessor. This coprocessor has a maximum core count and frequency of 72 and 1.7 GHz, respectively, placing it in between a typical CPU and GPU [29]. The Xeon Phi can be considered as a GPU stripped from its typical graphics pipeline. Even with all these types of accelerators, there are workloads that do not benefit from the high operating frequency found in CPUs, nor the high core count found in GPUs or the Xeon Phi. Compute-intensive workloads that benefit from custom arithmetic units are an example.

2.1.4 Application Specific Acceleration

Two recent trends in the data center allow for a wider diversification of compute resources by introducing application specific accelerators in the form of ASICs and FPGAs.
One trend is to use fixed-function ASICs, either tightly coupled on-chip or loosely coupled using an interconnect. On-chip accelerators range from compression to cryptographic engines. An example of a loosely coupled accelerator is Google's recent Tensor Processing Unit (TPU) [30].

While fixed-function ASIC accelerators provide a better power efficiency compared to a CPU and free up resources since the CPU is able to off-load work, on-chip accelerators consume valuable chip area while being dedicated to only one function. This renders a fixed-function accelerator useless when the underlying algorithm changes in the (near) future. Besides that, there are multiple time consuming steps in the design cycle of an ASIC that increase the time to market.
Another trend is the adoption of FPGAs. While FPGAs might not offer the same level of power efficiency that ASICs do, their (re)configurable nature allows for flexibility and reusability. According to the same study mentioned earlier [28], reconfigurable or dynamic cores can always achieve a higher maximum speedup than asymmetric configurations. While the design cycle of ASICs can be up to several years, FPGAs can be designed in a fraction of that time because layout and fabrication are not required. However, FPGAs are more difficult to use compared to CPUs and GPUs since both hardware and software have to be developed. FPGAs have not invested as heavily in double-precision floating-point pre-integrated units. Therefore they specifically shine in non-floating point compute-intensive workloads, on parallel workloads because of their inherently parallel architecture, and also in network communication related acceleration. Parallelism can be exploited at different levels of granularity and optimized fixed-function hardware accelerates the compute-intensity. The downside is that an FPGA consists of various pre-defined resources, like memories and DSP slices, that constrain the degrees of freedom of the design. When deployed as a compute accelerator FPGAs have typically been connected using a relatively low-bandwidth and high-latency interconnect, limiting their use for memory-intensive workloads. When deployed as network switches, FPGAs have typically been deployed with very high bandwidth using leadership PHYs (Physical Layers).

2.1.5 FPGA Adoption in the Data Center

FPGAs are becoming more interesting for specific workloads in the data center for various reasons. The (re)configurability allows for reuse and on-the-fly adaptation. In addition, FPGAs are more power efficient than other compute resources. This aspects becomes increasingly important for data center operators to keep the TCO down. Traditional limitations of FPGAs are starting to fade away with recent advancements in interconnect standards and system design.

- Interconnect limitations, such as low bandwidth and high latency, limit FPGAs to parallel and latency-insensitive workloads, because latency of the interconnect is much higher than that of host memory. Upcoming interconnect standards target these limitations.
- Programmability is limited to the use of hardware description languages. Advancements in software frameworks and high-level language to HDL compilers make FPGA acceleration accessible to software engineers. Also manual data movement between the CPU and FPGA is error prone and slow due to copy operations through memory and driver overhead. The typical off-load programming model is being replaced by a shared memory model with advancements in system design. This enables seamless integration of FPGAs with CPUs and lets the FPGA act as a thread-level functional unit instead. Accessing FPGAs for development becomes simpler by using cloud platforms such as Amazon EC2 F1.

One approach is integration of CPUs with FPGAs, like Intel is doing with their Xeon processors [21]. The FPGA is currently on the same package as the CPU and expected to be on the same die in the future. Such advancements enable a high bandwidth interconnect. Another approach is by using off-chip interconnects to attach FPGAs. We are interested in the latter approach, since such advancements in interconnects are also beneficial for other attached devices.

2.2 Interconnect Trends

In this section, the two typical drawbacks of FPGA adoption are explored in more detail. First bandwidth trends are explored, followed by reported improvements in FPGA programmability. Software framework and high-level to HDL compiler advancements are not discussed further in this thesis.

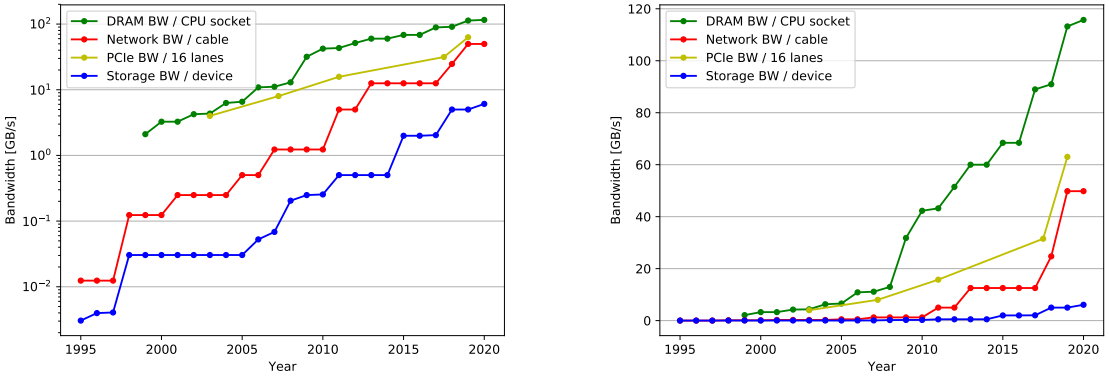
2.2.1 Attached Devices Push Bandwidth Requirements

Besides the adoption of various kinds of accelerators in the data center, the balance of bandwidths at system-level have changed significantly in recent years. Advancements in networking and storage result in increased bandwidth requirements of such devices. With the increase in complexity and volume of data center workloads, there is never enough or fast enough storage available. Emerging workloads, such as big data and machine learning, using data sets in the order of exabytes (10^{18}), are good examples of this [31]. Therefore, new storage protocols are quickly adopted, with NVMe over PCI Express being the latest achieving bandwidths of several gigabytes per device. The previous generation reached only half a gigabyte at best. Network bandwidths are increasing similarly by quick adoption of both the 100 Gbit/s and 400 Gbit/s standards in only a few years time. With more and more data and services moving to the cloud, network bandwidths have to increase rapidly.

2.2.2 Bandwidth Trends at Device-Level

Fritz Kruger, a SanDisk Fellow, collected data regarding DRAM, network and storage bandwidth for a presentation in 2016 and predicted the future until 2020. Figure 2.1 shows his findings, with the addition of PCI Express bandwidth to act as a proxy for interconnect bandwidth. For each generation of the PCI Express standard, the bandwidth of sixteen lanes is plotted, since this is typically the maximum number of lanes per PCI Express device. DRAM is inherently uni-directional and several cycles are required to turn the channel around. A memory controller takes care of this by combining reads and writes to limit the overhead cycles spent in configuring the channel. Therefore, DRAM bandwidth should be interpreted as either a read or write channel with the plotted bandwidth, while attached devices such as network and storage typically have a bi-directional link.
The slope of each of the fitted lines is important here. Clearly, both network and storage bandwidths are increasing at a much faster rate (steeper slope) than DRAM and PCI Express. The network and storage slopes are similar and double every 18 months. PCI Express doubles every 48 months while it takes DRAM 84 months to double in bandwidth. A shift in system balance is expected for future systems where DRAM, interconnect, network and storage bandwidth are about the same.
The fitted straight lines for each of the four data sets shown in Figure 2.0a indicate exponential behavior, which can be clearly seen in Figure 2.0b. This figure shows exactly the same data, but on a linear y-axis instead. While it might look like accelerators, such as GPUs and FPGAs, will have to compete for interconnect bandwidth with network and storage, vendors can always scale memory and interconnect bandwidth accordingly. While scaling is the trend, as becomes apparent from the next paragraph, and works in the short-term, it does not solve the fundamental problem of lacking DRAM bandwidth improvements.
The reason that DRAM bandwidth is not increasing at a similar pace is two fold. Typically, the number of channels or the channel frequency is increased. However, each solution has significant implications. Every additional channel requires a large number of pins (order of 100) on the processor package (assuming an integrated memory controller) that increases chip area

cost. Increasing channel frequency requires expensive logic to solve signal integrity problems at the cost of area, and more aggressive channel termination mechanisms at the cost of power consumption [32, 33].



(a) Data plotted on a semi-log scale.

(b) Data plotted on a linear-linear scale.

Figure 2.1: Bandwidth trends at device-level [34][1].

2.2.3 Bandwidth Trends at System-Level

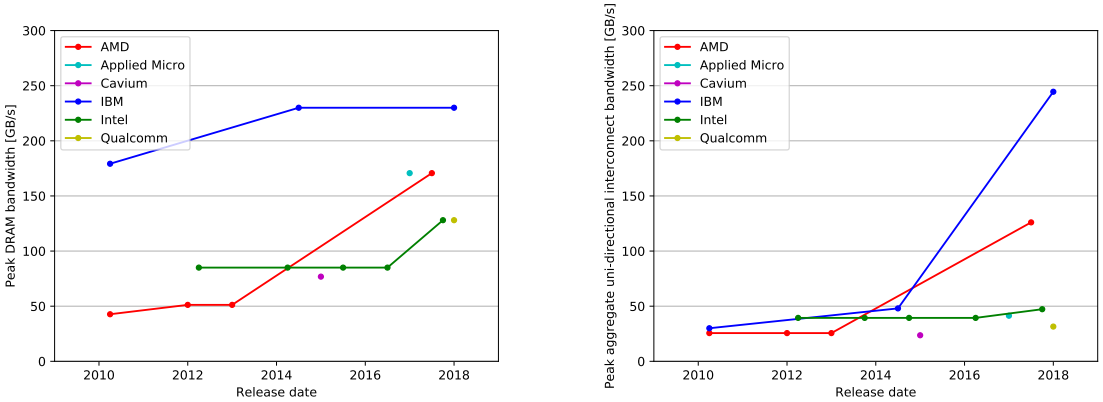
In order for vendors to stay ahead of the inevitable crossing of DRAM, interconnect, network and storage bandwidths, scaling is applied to both DRAM and interconnect bandwidth for the short-term. While the previous figures show predictions for future generations, more recent information regarding upcoming or recently released CPUs indicate that there is a large push to more DRAM and interconnect bandwidth. Figure 2.2 shows the peak[2] DRAM and interconnect bandwidth per CPU generation for single socket systems over a span of eight years. Per vendor and generation, the highest rated model is shown. Note that in the case of Intel, the E5 models have more interconnect bandwidth, while the E7 models have more DRAM bandwidth. Unreleased CPUs are plotted at the year 2018.

DRAM Bandwidth Trends

The DRAM bandwidth data shown in Figure 2.1 was mostly taken from the Intel Xeon product family according to the author [34]. Figure 2.1a shows that AMD and IBM, compared to Intel, scaled their DRAM bandwidth aggressively by increasing the number of memory channels or by using additional buffer chips between the last-level cache and the DRAM. Only after several years did Intel improve the memory bandwidth in their latest generation by increasing the operating frequency and number of channels. While the Intel Xeon DRAM bandwidth follows the predictions quite well, both AMD's EPYC processors and IBM's POWER8 and POWER9 processor break the trend. The POWER processors offer roughly twice as much memory bandwidth compared to the latest Intel Xeon family.
An implication for the slow DRAM bandwidth increase is that flash storage can be attached as accelerator and act as DRAM or local storage for a data intensive accelerator, for example by exploiting data locality.

[1] Data points were approximated from the referenced figures in order to add the PCI Express standard bandwidth and represent all bandwidths in GB/s.

[2] The only exception are the IBM POWER8 and POWER9, which state the sustained DRAM bandwidth.



(a) Peak DRAM bandwidth.

(b) Peak aggregate uni-directional interconnect bandwidth.

Figure 2.2: Study of bandwidth trends at system-level for single socket CPUs per generation.

Interconnect Bandwidth Trends

Figure 2.1b shows the peak aggregate uni-directional interconnect bandwidth per CPU socket. This is calculated as the product of the uni-directional bandwidth per lane and the number of lanes. In the past, nearly every vendor used their own interconnect standard, possibly with a bridge to PCI Express. The general consensus is to use PCI Express and it has become the industry standard. While the introduction of PCI Express Gen 4 took longer than previous generations (see Figure 2.0a), initiatives took off to extend the PCI Express standard with coherency protocols for seamless integration of attached devices. IBM started in 2014 with CAPI and more recently the CCIX consortium tries to implement a protocol for current and future PCI Express standards.
The aggregate interconnect bandwidth at system-level is increasing by an order of magnitude and has passed DRAM bandwidths. In the latest generation, both AMD and IBM are investing aggressively. AMD scaled the number of PCI Express Gen 3 lanes per socket to 128 lanes. For comparison, the latest Intel processor offers a maximum of 48 lanes for the same PCI Express generation. However, for multi-socket systems the AMD CPU will sacrifice half of its lanes for SMP, while Intel has a dedicated link for that. IBM's upcoming POWER9 processor has almost twice the interconnect bandwidth compared to AMD. This is achieved by having 48 lanes of PCI Express Gen 4 and 48 lanes of a new 25 Gbit/s link called BlueLink. Note that both types of interconnect on the POWER9 can be used for SMP as well. This fundamental improvement enables a massive increase in bandwidth compared to other state-of-the-art systems. Although a significant improvement, it is only a small bump on the log-scale in Figure 2.0a.
Very recently, new players such as Applied Micro, Cavium and Qualcomm are entering the server processor market with ARM-based processors. Such servers are targeted for cloud, content delivery, storage and web workloads and differ greatly from the traditional high performance and power consuming POWER and x86 architectures. These types of workloads do not require high-bandwidth accelerators, but instead prefer fixed-function on-chip accelerators and integrated network adapters. This class of servers does allow for memory bandwidths similar to those of Intel.

This study shows that the latest generation of processors tries to keep up with the exponentially increasing bandwidth of network and storage by scaling the interconnect in a similar fashion,

and no longer follows the tradition shown in Figure 2.1. It is important that DRAM bandwidth scales as well, in order to not become a bottleneck. Increasing the interconnect bandwidth by an order of magnitude forces us to reconsider design choices for accelerators.

2.3 Current Interconnect Bottlenecks

It has been shown that an increase in interconnect bandwidth is required for emerging workloads in the data center. However, this problem is not solely solved by blindly scaling the current interconnects because the traditional Input/Output (IO) model will become a bottleneck. This section introduces the traditional IO model and its bottlenecks.

2.3.1 Traditional IO Model

Workloads contain serial and parallel components, where parallel components typically benefits from highly parallel architectures as mentioned in Section 2.1.3. Due to the need for heterogeneous systems, different compute elements must communicate efficiently without decreasing potential execution speedup.
In a traditional IO model, the host processor has a shared memory space across its cores with coherent caches. Attached devices such as FPGAs, GPUs, network and storage controllers are memory-mapped and use a DMA to transfer data between local and system memory across an interconnect such as PCI Express. Attached devices can not see the entire system memory, but only a part of it. Communication between the host processor and attached devices requires an inefficient software stack in comparison to the communication scheme between CPU cores using shared memory.
Figure 2.3 shows the data flow in the traditional IO model. The yellow boxes within the purple physical memories indicate seperate address spaces. In order to offload data from host memory to an attached device, data is copied from the application (app) region by the CPU into a pinned (non-pageable) section of the host memory called a buffer. Only the buffer is visible to the attached device (due to different address spaces, therefore different instructions for memory and Memory Mapped IO (MMIO) accesses), because it does not share the same translation tables with the CPU. By reading the buffer, with the help of device drivers, the CPU is able to write the data, across an interconnect, into the local memory of the attached device. The attached device fulfills a certain Function, which reads the data from the local memory and processes it. The result will then be written into the local memory after which the Function informs the host it has finished. A DMA for example will move the result into a second buffer within the host memory. From here, the CPU copies the data to the application address space after which the application can continue.

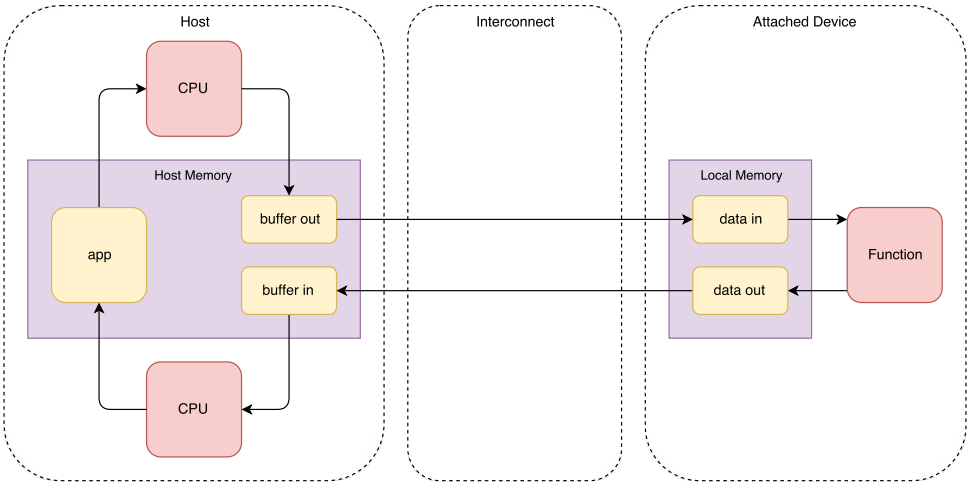


Figure 2.3: Multiple copy operations in the traditional IO model.

2.3.2 Communication and Synchronization Overhead

Communication and synchronization between the host and attached device using interrupts and MMIO operations typically involves the device driver and introduces overhead [35]. These communication channels are used to start or stop the attached device.
While the attached device is operational, typically the app running on the host is idle and waits for the Function to finish, after which they synchronize and continue.
Functions with a relatively low number of data transformations suffer from a high interconnect overhead per data element with the traditional model. The communication overhead decreases the potential execution speedup, limiting acceleration to data transformation intensive workloads.

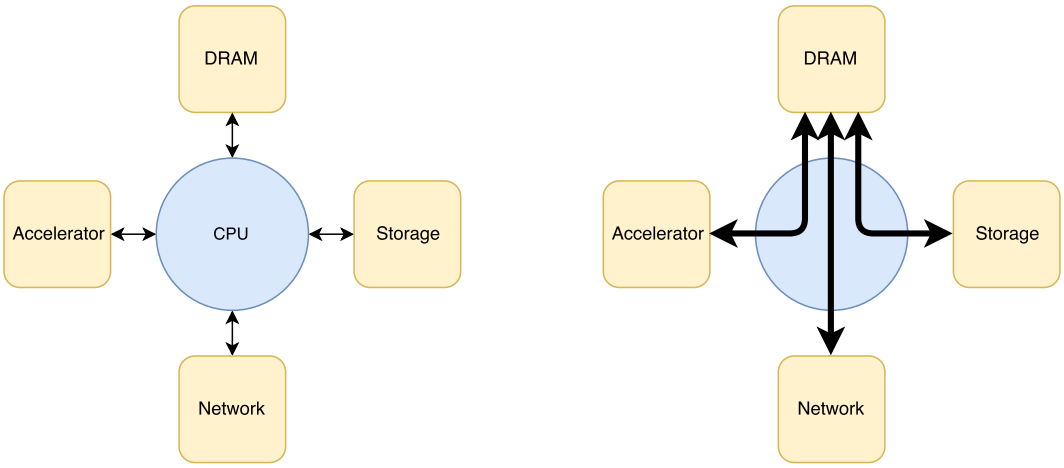
2.3.3 Host Memory Access Congestion

Another bottleneck, illustrated by Figure 2.3, is the host memory bandwidth for systems with increasing interconnect bandwidth such as the ones shown in Section 2.2.3. The traditional model requires four memory copy operations, of which the CPU is involved in a total of three reads and two writes across the host memory channel. Since all attached devices can potentially use all of their assigned bandwidth, the host memory channel will suffer from severe congestion for servicing all copy operations. The fact that host memory is inherently simplex and cycles are required to turn the channel around is not beneficial either.

Case Study: IBM POWER9

To illustrate the problem of host memory congestion, the IBM POWER9 processor is chosen as an extreme example. It supports two new high-bandwidth interconnect standards: PCI Express Gen 4 and OpenCAPI. Figure 2.3a shows a hypothetical system architecture where host memory, an accelerator (ASIC, FPGA or GPU), and a network and storage controller are attached to the CPU.
A possible system configuration could have host memory attached with $200\,\mathrm{GB/s}$ of bandwidth and each attached device with roughly $150\,\mathrm{GB/s}$ of bandwidth. Using the traditional IO model, the IO devices will be fighting for access to the system memory channel. $450\,\mathrm{GB/s}$ of data is competing for the same $150\,\mathrm{GB/s}$ memory channel, since data exchange happens by copying through memory. Imagine that in this example, memory access requested by the CPU is not

even taken into account. The host memory would seriously bottleneck the whole system in this case, as shown in Figure 2.3b where the thick arrows indicate data transfers.
A system configuration like this would not be possible with a traditional IO model. The 300 GB/s of bandwidth that OpenCAPI provides mitigates the presented bottlenecks by introducing a coherent and shared memory access model to host memory.



(a) Hypothetical system architecture. (b) Host memory access congestion.

Figure 2.4: Hypothetical system architecture suffering from the traditional IO model.

Case Study: AMD EPYC

Systems that heavily rely on interconnect bandwidth to improve performance by attaching FPGAs, GPUs, network and storage have to take care that DRAM bandwidth will not become a bottleneck. As shown in Section 2.2.3, vendors make sure that DRAM bandwidth will not become a bottleneck in the overal system architecture.
In contrast, AMD's EPYC processor has a DRAM and duplex interconnect bandwidth of 170 GB/s and 256 GB/s, respectively. Since PCI Express Gen 3 is used, it could very well be that a traditional IO model is used since PCI Express Gen 3 does not support a shared address space between the processor cores and attached devices. If this is the case, system performance could be seriously limited. However, half of the PCI Express Gen 3 lanes can be used for SMP, and this way would remove the bottleneck.

2.4 Interconnect Coherency and Shared Memory: A Necessity

Recent initiatives target the limitations described in Section 2.3. Current widely adopted interconnect standards such as PCI Express and AXI are based on the traditional IO model. However, extensions try to improve the usage model. Chapter 3 will discuss the state-of-the-art interconnects in more detail. This section explores necessary changes to the traditional IO model to accommodate for the bandwidth scaling employed by current and future processors.

2.4.1 Coherent IO Model

In order to continue scaling of IO bandwidth to fulfill the requirements of emerging accelerators and attached devices, the traditional IO model has to change and the bottlenecks presented in

Section 2.3 have to be addressed.
Shared memory and coherency should be extended to attached devices since accelerators will become an integral part of the ecosystem and should act as a peer to processor cores. System memory should be relieved from performing data copies by employing a shared memory address space between the processor cores and attached devices. Coherency will simplify the programming model for attached devices. Communication overhead can be improved by allowing for shared variables between the host and the attached device in shared memory.
The coherent IO model is shown in Figure 2.5. In comparison to the traditional IO model, shown in Figure 2.3, there is no CPU involvement and no longer any driver overhead, since copying to buffers is not required. The application has direct access to the address space of the attached device and data exchange is coherent.

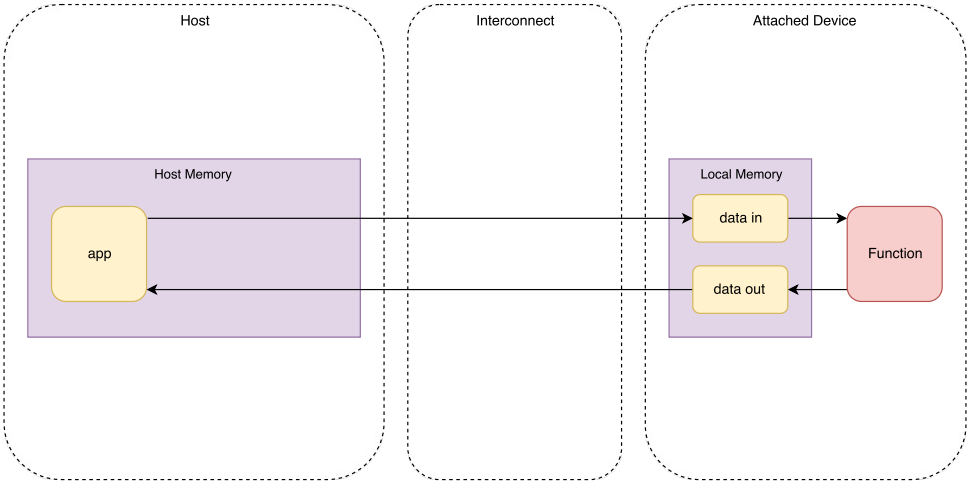


Figure 2.5: Coherent IO model.

With the coherent IO model, attached devices are no longer notified using MMIO communication but instead using shared memory. Similarly, completion by the attached device is signaled using shared memory instead of an interrupt for example. By using shared memory, device driver and operating system overhead is decreased significantly.
In comparison to Figure 2.4, the coherent IO model not only removes the memory copy overhead and therefore avoids memory access congestion, it also allows attached devices to move data between each other without touching host memory. An example could be that data is coming in through a network controller and is immediately moved to an accelerator, as shown in Figure 2.6.
The following sections discuss the required changes for the coherent IO model in more detail.

2.4.2 System-Wide Shared Memory Address Space

Shared memory is an abstraction of a system with multiple physical memory resources and presents these memory resources as a continuous address space. It is desired by programmers to hide details of memory accesses from different physical locations. Without shared memory, data structures look different between the host and accelerator because the accelerator is not able to dereference a pointer. All pointers have to be resolved before sending the code to the accelerator for execution, and this may not even be possible.
A shared memory address space will decrease congestion of accessing system memory to result in lower access latency and device driver overhead [36]. It also improves programmability since it enables shared buffers (zero-copy) and pointer dereferencing, thus simpler data movement be-

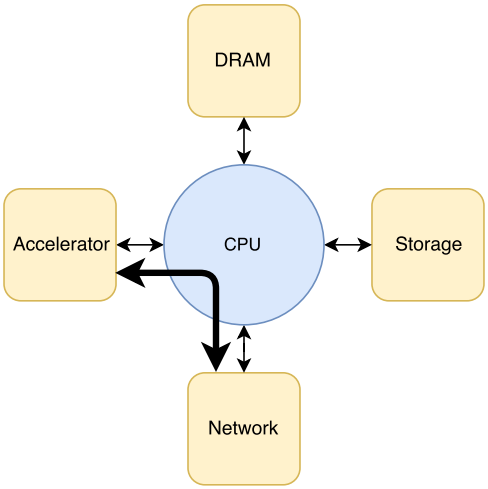


Figure 2.6: Hypothetical system architecture with a coherent IO model.

tween physical memories (host and device memory). The accelerator acts similar to a processor thread and this allows to easily accelerate a single function in a big application, without having to massively restructure the code.
It is obvious that interaction with attached devices using the traditional IO model is not perfect. It requires complex drivers on the host, is error prone due to user data movement, and has a high latency rendering it unpractical for latency sensitive workloads. Shared memory allows the system memory address space to be shared with the IO. This enables for example direct data movement from a network card to a GPU, instead of first copying to main memory or interaction by the CPU. This is achieved by providing the IO with the same address translation tables as the CPU.

2.4.3 System-Wide Coherence

Caches are widely used in microprocessors and cache hierarchies are becoming larger and more complex, by using various cache levels. By scaling the number of processor cores on a single die, multiple copies of the same data can exist. Cache coherency is required to keep data coherent without any software-intervention. Relying on software to provide snooping, write-back, and invalidation is relatively slow. Hardware-based coherence is preferred since it simplifies the programming model and is faster.
Currently, without coherence, the user has to take care of moving data between resources manually, which is error prone. Typically, data is transferred to the accelerator and the application waits for an interrupt signal from the accelerator to signal completion. The application fetches the result from the accelerator across the interconnect. Caches have to be flushed before other resources can access the data. The FPGA feels more like an off-load engine instead of an extension to a thread running on the CPU.
With coherency, a consistent view of memory contents by all participants (CPU cores and IO devices) is guaranteed. Attached devices operate natively within the application's user space and coherently with the host CPU. This allows attached devices to fully participate in an application without kernel involvement or overhead [5].
With software coherence, a large burden is placed on the application, drivers and OS to manage timed cache cleaning, maintenance and invalidations. Such operations take time and effort (cache contents have to be written out to system memory). Since caches are invisible to software, managing all of these copies in software is difficult. Keeping caches coherent in software

means that all caches have to be flushed [37].
Hardware coherency removes the software challenges and makes sharing transparent to the application, at the cost of additional memory traffic between caches about the state of their contents in order to keep all of them synchronized.
Coherency also enables new types of workloads such as pointer chasing, but is not truly required. It does significantly simplify the programming model by providing synchronization between the host and attached device mechanisms in hardware or software, therefore making slow interrupt-based solutions obsolete.

2.4.4 Thread Synchronization

Before the usage of accelerators and other attached devices was as common as today, multiple processor cores were used to exploit workload parallelism. Such workloads typically make extensive use of synchronization operations such as barriers, mutexes and semaphores. Therefore, when attached devices act as a peer to processor cores, it makes sense to employ similar synchronization operations. By doing so, notifications using interrupts can be avoided and makes porting existing multi-threaded workloads easier. Instead, the host and attached device communicate through shared variables in shared memory [38].
A lock operation could be implemented from which more complex synchronization operations can be built. Recent interconnect standards incorporate more complex atomic operations to replace the lock operations.
Another benefit of moving away from interrupt-based synchronization is that the number of hardware interrupt signals no longer limits the number of interrupts, since interrupts are handled using shared memory. This solution scales much better.

2.5 Preliminary Concluding Remarks

Emerging workloads require a change in system architecture. A diversification of compute resources enables speedups not possible with a single type. The adoption of FPGAs is slow due to interconnect limitations and a complex programming model. To address these issues, attached devices are required to be tightly coupled with the host processor at memory-like bandwidths. Currently, this is not the case and interaction with attached devices involves unnecessary overhead. By extending the shared memory space and coherence domain across the interconnect, attached devices act as a peer to the processor cores. This simplifies FPGA acceleration and enables new usage models. Chapter 3 takes a closer look at state-of-the-art interconnect standards and evaluates the current state of the interaction between the host and attached devices.

Chapter 3

State-of-the-Art Interconnects

Nowadays the Peripheral Component Interconnect Express (PCI Express) and AXI are the interconnect industry standard for PC and server systems, and embedded platforms, respectively. Recently, three open initiatives were announced: CCIX, Gen-Z and OpenCAPI. These open standards are all driven by ISA-agnostic tighter coupling of processors and accelerators, by enabling direct memory access between compute resources and reducing data movement. Also new and emerging memory and storage technologies are exploited [39].
This chapter focuses on interconnects targeted for accelerators, network, and memory and storage solutions. However, specific memory and storage features of such interconnects will not be discussed. Also interconnects tailored for specific domains such as Ethernet and InfiniBand for networking, NVLink for GPUs and Gen-Z for storage are not discussed, nor are SMP protocols.

3.1 PCI Express

PCI Express has been around since 2003 and gone through several generations. The PCI-SIG is a group of over 900 companies that maintain the standard. Currently the most widely adopted generation is number three and generation four compliant devices are slowly being released. The remainder of this section briefly explains the architecture of PCI Express and summarises key features of current and future generations.

3.1.1 Architecture

This section explains the architecture of PCI Express Gen 3 and later, since various changes have been made at the packet level compared to previous generations of PCI Express which will not be discussed.
PCI Express is a packet-based, split transaction protocol with a point-to-point or switched topology. Split transaction means that a request and response are separated by time. Each device is connected to the root complex. The root complex is the root of the IO hierarchy and is connected to the processor and host memory. A PCI Express bus link supports full-duplex communication between two endpoints, with no inherent limitation on concurrent access across multiple endpoints. It uses credit based flow control and typically each link consists of one, four, eight or sixteen lanes. Legacy PCI features are backwards compatible with PCI Express.

Protocol Description

The PCI Express architecture consists of three logical layers called the Transaction Layer, the Data Link Layer, and the Physical Layer [1]. Figure 3.1 shows a layering diagram and the receive

(RX) and transmit (TX) channels of the architecture. Each layer will be briefly discussed.
PCI Express uses packets to communicate between participants of the link. Packets are formed in the Transaction Layer and are extended with additional fields when passing through other layers. These additional fields contain information required by other layers to handle the packet appropriately. The receiver of a packet removes these fields in reverse order and uses the information.

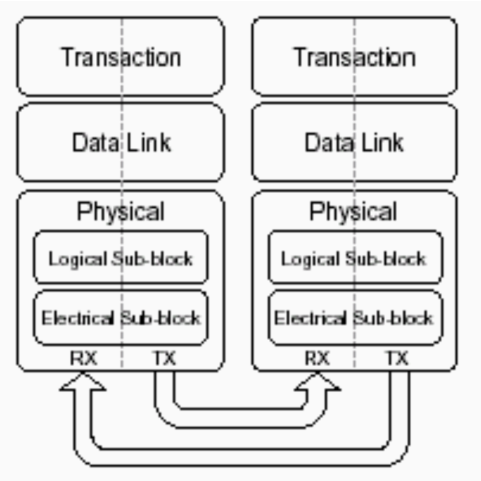


Figure 3.1: Layering diagram of the PCI Express standard [1].

Figure 3.1 shows the layering diagram of the PCI Express standard. The Transaction Layer is the top layer and assembles and disassembles Transaction Layer Packets (TLPs). TLPs are the packets used to communicate information and data between two endpoints and consist of a header and data part. Also flow control is managed by using a credit-based scheme and ensures that TLPs are only transmitted when a buffer is available on the other endpoint. This eliminates wasting bandwidth on packet retries due to resource constraints.
The Data Link Layer is the middle layer and tags TLPs and handles error detection and correction. The transmission side of this layer pre-pends a sequence number (tag) to the TLP and appends a CRC field to it. The receiving side validates the sequence number, by checking if it is continuous with the sequence number from the previous TLP. It also validates the data by checking the CRC. If the TLP is valid, an acknowledgement (ACK message) is sent to the transmitter. If one or both are invalid, a NAK message is sent and re-transmission of all TLPs starting from the invalid one is requested. The ACK and NAK messages are communicated between layers as Data Link Layer Packets (DLLPs).
The Physical Layer is the lowest layer and consists of both electrical circuitry, such as a serialiser/deserialiser (SerDes), and logical components to initialize the interfaces. A lane between two endpoints consists of two unidirectional differential signalling pairs and multiple lanes can be bundled together to form a link.

Cache Coherency Snooping

Originally proposed as an extension to PCI Express Gen 2, TLP Processing Hints (TPH) [40] provide hints for the host to improve memory access performance by taking the cache hierarchy into consideration. This is done by providing several bits in the TLP. The host snoops memory access requests from PCI Express attached devices to enforce cache coherency by hardware [41]. However, these snoop hints are not required for every memory access request. An example

is when a speculative read-ahead buffer is used by the operating system to access storage. Snooping this data could pollute host caches and, therefore, the TPHs can be configured on a per packet basis [42].

Address Translation Services

Another extension proposed for PCI Express Gen 2 are the Address Translation Services (ATS) [43]. This extension was included in the PCI Express Gen 3 specification and translates untranslated addresses to physical addresses. ATS enables attached devices to request address translation from the host in advance to alleviate potential congestion during times with intensive communication across the interconnect [42].
To relieve the host translation agent, the extension proposes attached devices to implement an address translation cache (ATC) on the device itself. This allows device designers to size the ATC depending on the usage model of the device.

Atomic Operations

Atomic Operations [44] also have been added as an extension to PCI Express Gen 2 and were incorporated in the PCI Express Gen 3 specification. Atomic operations are used as a locking mechanism for shared memory and as a means of communication between host and attached device to reduce overhead compared to traditional solutions [42]. Three different atomic operations are supported: FetchAdd, Swap, and Compare-And-Swap.

3.1.2 PCI Express Gen 3

The third generation of PCI Express was introduced in 2010 [41] and introduced various changes at packet level compared to the previous generation. The theoretical bandwidth of a single lane is 1.0 GB/s but in practice is lower due to encoding, packet and traffic overhead [45]. Latency characteristics are difficult to come by. A study conducted in the field of real-time Ethernet found highly variable results [46]. A network interface card is attached to an Intel i5 3550 processor, either in the graphics or IO PCI Express Gen 3 slot. The graphics slot is connected directly to the root complex of the processor while the IO slot is connected through the chipset. The impact of the location of the slot is clearly visible in the obtained latencies of 1.38 µs for the graphics slot and 3.11 µs for the IO slot. These results have been obtained by reading the clock register located in the network card. The latency is defined as the time passed between two consecutive read requests of the clock register.

3.1.3 PCI Express Gen 4

In October 2017, the final specification for PCI Express Gen 4 was released, limited to members of the PCI-SIG [47]. The most significant improvement is the doubling of bandwidth to 2.0 GB/s per lane while retaining compatibility with previous PCI Express generations.

3.1.4 PCI Express Gen 5

In June 2017, PCI Express Gen 5 was announced [48]. Not much information has been shared publicly besides the doubling of bandwidth to 4.0 GB/s per lane compared to the previous generation. The information presented in Table 3.1 on Page 45 assumes that PCI Express Gen 5 supports all the features from previous generations.

3.2 CAPI

To address emerging workloads and inefficiencies present in the traditional IO model (see Chapter 2), IBM's POWER8 processor introduced the Coherent Accelerator Processor Interface (CAPI) in 2014. CAPI enables accelerators to be plugged into PCI Express slots and act as a coherent peer of other caches within the system memory hierarchy [2]. CAPI also allows data to be referenced by an effective address in the same way as an application running on the host processor without the need for a device driver. A generic kernel extension enables CAPI in the host operating system. This removes software overhead, traditionally present for a software thread running on the host processor to share data with an attached device.

3.2.1 Architecture

PCI Express does not natively allow attached devices to operate as a coherent peer, since it has no notion of the coherency protocol used within the host processor. To bridge both protocols, a hardware proxy unit, called the Coherent Accelerator Processor Proxy (CAPP), resides within the host processor and is connected to the coherent fabric as shown in Figure 3.2. Besides the CAPP, also the PHB is present within the host processor and provides the necessary hardware for the underlying PCI Express protocol used. The attached device, either an ASIC or FPGA, contains the POWER Service Layer (PSL) and one or multiple Accelerator Function Units (AFUs).

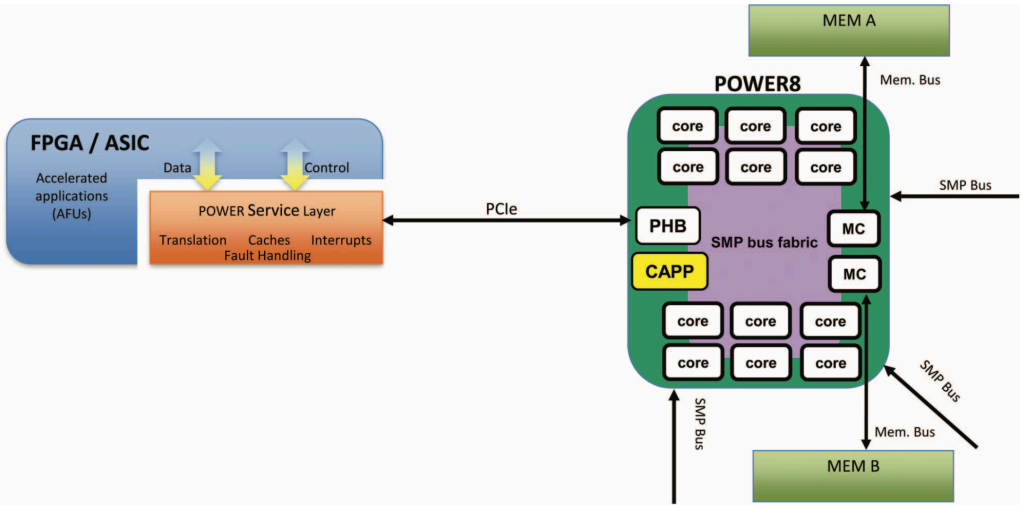


Figure 3.2: System architecture of a CAPI attached device [2].

When the CAPP, PHB, PCI Express and PSL are combined, the AFU is able to operate coherently on data in host memory. To reference the requested memory, the AFU uses effective addresses that are translated by a memory management unit (MMU) within the PSL. The PSL may also send interrupts to the host processor on AFU completion or to indicate a translation fault.
The interface provided by the PSL to the AFU hides cache coherence complexities and address translation. AFUs request host memory using a load-store model to user space effective addresses. Requests can be either cacheable or write-through (not cached). Cacheable requests are typically used to communicate control information between multiple processes and can be stored within a 256 kB cache within the PSL [49]. Write-through requests are typically used for

data manipulation outside of the coherence domain and therefore require less messages to be transmitted over the PCI Express link and reduce overhead.
The programming model requires the application to setup data for the attached device in host memory, and to notify the AFU when the data is ready to be consumed. It is not possible to transfer data from the host processor to the AFU directly because the AFU has to master read and write commands for data located in host memory [50]. The AFU can be notified in two different ways, either the AFU polls a location in host memory or the application running on the host writes into a specific MMIO register on the attached device.

Coherence

Coherent host memory access is enabled by a combination of the CAPP and PSL and both contain cache lines used by the AFU. The CAPP acts as a proxy for the PSL and snoops coherence messages on the fabric. Snoops that hit cache lines present in the CAPP could generate messages, transmitted across the PCI Express link, for the PSL. Coherence enables an AFU to cache data from host memory and to request locks to implement atomic operations for example [51].

Address Translation

In order for the AFU to operate on effective addresses, the PSL consists of an MMU that uses the host processor's page tables [35]. This enables AFUs to de-reference pointers, similar to a thread running on the host processor. System software manages page faults [49]. The MMU performs address translations and caches recent translations, in order to avoid page table walks. The CAPP also snoops translation invalidation messages from the fabric since the PSL consists of a translation cache.

3.2.2 CAPI 1.0

The first generation of CAPI was introduced with the IBM POWER8 processor in 2014 [49]. Since there is only one CAPP unit per POWER8 processor, the number of attached CAPI 1.0 devices is limited by the number of processors in the system. To improve adoption, Xilinx released an AXI4 to CAPI 1.0 adapter [52].
The total bandwidth available to a CAPI 1.0 attached device is determined by the underlying PCI Express Gen 3 interconnect and the number of lanes. CAPI 1.0 supports eight or sixteen lanes per attached device [5]. The online CAPI Developers Community used a Nallatech P385-A7 FPGA attached using an eight lane PCI Express Gen 3 interface in conjunction with a POWER8 S824 system to assess the bandwidth and latency of CAPI 1.0 [50]. The measurements were obtained using a modified memcpy demo supplied with the developer kit. When data resides in host memory, a read bandwidth of $3.42\,\mathrm{GB/s}$ was achieved with an average latency from PSL request to response of $864\,\mathrm{ns}$. Similarly, write bandwidth of $3.88\,\mathrm{GB/s}$ with an average latency of $838\,\mathrm{ns}$ was achieved. Reads and writes that hit in the PSL cache achieved a latency of $120\,\mathrm{ns}$.
Due to the protocol overhead of CAPI 1.0 on top of PCI Express Gen 3, the obtained bandwidth is significantly less compared to the theoretical capabilities of PCI Express Gen 3. On the flipside, CAPI 1.0 provides several features and usage models that are not possible with a traditional PCI Express Gen 3 interconnect.

Streaming Framework

The Streaming Framework is an extension to CAPI 1.0, designed and implemented by Matthijs Brobbel [11]. Instead of presenting the PSL interface to an AFU designer, a simple read and write streaming interface is presented with cache line granularity (128 bytes), similar to a DMA. It supports a single stream (multiple streams in simulation only) and returns read data in order, whereas the underlying CAPI interface does not guarantee such ordering. The philosophy behind the Streaming Framework is to hide the hassle of directly talking to CAPI by simplifying the interface. Preliminary results of a memcpy AFU show a bandwidth of nearly 3.3 GB/s using a Nallatech P385-A7 FPGA card with eight PCI Express Gen 3 lanes [12]. Similar to the explanation in Section 3.2.2, the CAPI 1.0 protocol overhead on top of PCI Express Gen 3 limits bandwidth.

Storage, Networking, and Analytics Programming

The Storage, Networking, and Analytics Programming (SNAP) framework enables designers to easily integrate FPGA-based accelerators to work with data located in host memory, flash or attached storage, or from other connected devices such as Ethernet [14]. One could argue whether it is a continuation of the Streaming Framework in the sense that a simplified interface eases integration. SNAP consists of an AXI-to-CAPI bridge, MMIO registers, a host DMA, and a job management unit [13].
An AFU can be controlled using an AXI-lite interface and the AFU has access to host memory through a coherent 512 bits wide AXI interface operating at 250 MHz [53]. Additionally, AXI bridges to DRAM and NVMe are available. All of these hardware units are accompanied by a software library. No bandwidth results are public yet, but SNAP ought to be able to achieve the same bandwidth as CAPI 1.0.

3.2.3 CAPI 2.0

The successor to CAPI 1.0 can be found in the IBM POWER9 processor [19]. Features from CAPI 1.0 are retained and the main improvement is the use of PCI Express Gen 4 that doubles the available bandwidth per lane. The effective bandwidth, compared to CAPI 1.0, will be more than double because protocol overhead is reduced by including packets with a larger payload. Another improvement is the addition of a host thread wake-up construct in hardware [5].

3.3 OpenCAPI

OpenCAPI is a continuation of CAPI, but an open standard, that allows a microprocessor, agnostic to processor architecture, to attach to coherent user-level devices and advanced memories [19]. It provides a high-bandwidth, low latency interface optimized for ease of programmability and integration. By implementing complexities of coherence and virtual addressing on the host microprocessor, attached devices can be simplified and are interoperable across multiple processor architectures. Attached devices operate natively within an application's user space and coherently with processors, since it appears as a peer to the host processor cores by sharing the same virtual memory space. This allows an attached device to fully participate in an application running on a host processor without kernel and device driver overhead of data copies or pinned pages and simplifies the programming model. Besides accelerators, OpenCAPI also supports classic and emerging memory and storage technologies. Chapter 4 discusses OpenCAPI 3.0 in much more detail.

3.3.1 Architecture

In essence, OpenCAPI uses the philosophy behind CAPI and replaces the underlying PCI Express based protocol with a streamlined point-to-point standard designed from scratch [5, 39, 7]. With CAPI, address translation was done on the attached device in the PSL. In OpenCAPI, the virtual-to-physical address translation occurs in the host processor and enables a shared virtual address space. By pushing the translation hardware into the host silicon, the protocol layers are asymmetric on the host and attached device, and the data link and transaction layers are very thin on the attached device. This reduces design complexity and is especially beneficial for FPGAs since less resources are spent on the interconnect and more can be used for the actual accelerator. Initially, OpenCAPI will not support a coherent cache on the attached device, contrary to CAPI. Nonetheless, coherent memory accesses between the host and attached device are supported.
Attached devices never have access to physical addresses due to the shared virtual memory space. This improves security by eliminating the possibility of a defective or malicious device accessing memory locations belonging to the kernel or other applications that it is not authorized to access. Also pointers can be dereferenced on the attached device. This enables memory access patterns such as pointer chasing and linked lists without driver involvement. Multiple contexts are supported within the protocol, allowing multiple threads to utilize the attached device simultaneously. To synchronize threads and facilitate multi-thread programming, atomic memory operations are supported. In addition, special opcodes are available for warming up the address translation cache in the host. Finally, in order to reduce translation latency when using a new page, a new mechanism for waking up a host thread with low latency instead of interrupts or polling of host memory.

3.3.2 OpenCAPI 3.0

The first processor to use OpenCAPI is the IBM POWER9. Since it is a continuation of CAPI, the first generation of the OpenCAPI standard is called OpenCAPI 3.0. It will share the BlueLink 25G I/O facility with NVLink 2.0, peaking at a half-duplex bandwidth of $25\,\mathrm{Gbit/s}$ at eight lanes per attached device. More information regarding the POWER9 processor and OpenCAPI 3.0 can be found in Chapter 4.
The most recent source as of writing this thesis is a slide deck to inform on the progress of OpenCAPI 3.0 presented at the end of 2017. A POWER9 processor was paired with an Alpha Data 9V3 FPGA card with a Xilinx VU3P FPGA and achieved a bandwidth of roughly $22\,\mathrm{GB/s}$ for streaming read and write operations [5].
Throughout this thesis, OpenCAPI will be used as a proxy for OpenCAPI 3.0, unless specifically stated otherwise.

3.3.3 OpenCAPI 4.0

OpenCAPI 4.0 will continue where 3.0 left off and is still in definition. One of the main features is the re-introduction of cache coherency on the attached device. This provides a latency advantage for frequently used data. This cache will use effective addresses while CAPI uses real addresses [5]. Address translation will occur on the host similarly to OpenCAPI 3.0.
New features consist of additional link widths of four, sixteen and 32 lanes compared to a single link configuration of eight lanes for OpenCAPI 3.0 [54]. The low latency communication mechanism (wake host thread) between the attached device and host application as is present in OpenCAPI 3.0 will be enhanced by rollover to interrupt. This avoids the use of current inefficient mechanisms such as interrupts or polling of the host memory

3.4 CCIX

Similar to OpenCAPI, CCIX is an initiative promoted by several companies for an open interconnect standard that is host architecture agnostic [55]. CCIX extends the processor's coherency domain to heterogeneous devices such as accelerators, network adapters, and memory and storage solutions. This is done using a driver- and interrupt-free framework for data sharing across the link in hardware and allows low-latency main memory expansion as well.

3.4.1 Architecture

CCIX is an extension to PCI Express. Therefore, little to no modification to PCI Express controllers is required. It uses the PCI Express extension for address translation services via ATS/PRI [39] and requires additional logic, as shown in Figure 3.3, to implement the CCIX transaction layer. This layer carries the coherence messages, while the CCIX protocol layer and link layer implement the coherence protocol and act upon it. These blocks require tight integration with internal system-on-chip (SoC) logic for caching, and depend on the SoC's ISA. SoC designers implementing CCIX typically partition the CCIX protocol and link layers from the CCIX transaction layer, so they can achieve tight integration with the internal SoC logic [3].

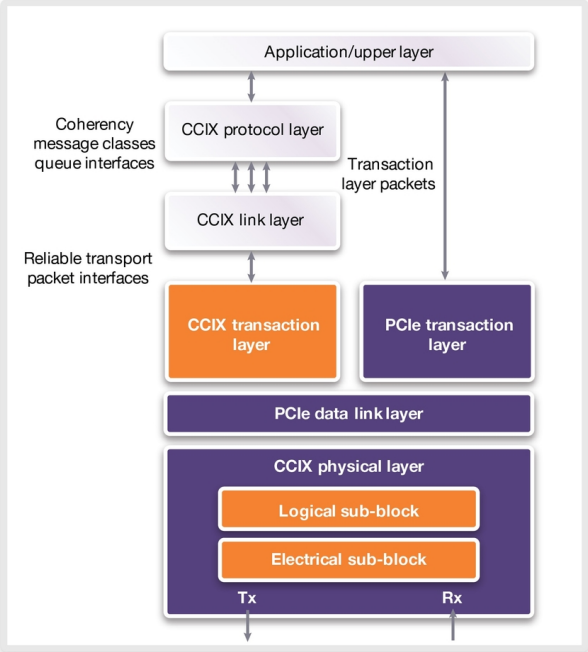


Figure 3.3: Layering diagram of the CCIX standard [3].

The standard supports peer-to-peer and switched topologies with unidirectional bandwidths between 1 Gbit/s and 3.125 Gbit/s per lane. Links consist of eight or sixteen lanes. It provides full cache coherence between the processor and accelerators. Communication with CCIX-attached devices is managed by vendor-specific drivers and libraries.
During the typical PCI Express initialization process, the highest mutually supported transfer speed is determined between the two CCIX components. Software running on the host checks configuration registers on the attached device to decide on the transfer speed.

3.5 AMBA AXI

The Advanced Microcontroller Bus Architecture (AMBA) is an open standard by ARM for interconnect protocols. Today, protocols from AMBA are the de facto standard in the world of ASICs, FPGAs, SoCs and embedded devices for communication between cores and between cores and attached devices. Currently, FPGAs mostly use the Advanced eXtensible Interface (AXI) protocol by AMBA that will be discussed in this section.

3.5.1 Architecture

While AMBA owns multiple interconnect standards, AXI is the most widely used standard in the world of FPGAs. An example is the abundant use of it in the Xilinx FPGA tool chain and all hardware modules available in the SNAP framework. The latest generation of AXI was released in 2010 in the fourth iteration of the AMBA specification [56].
AXI targets high performance and high clock frequency systems by providing a memory-mapped interface that consists of five independent channels: read address, read data, write address, write data, and write response. AXI does not define a specific clock frequency or data width in order to serve the needs for multiple requirements. As an example, the Xilinx AXI generated modules can be configured with a data width of up to 1024 bits [57]. The frequency depends on the FPGA used in this case and could be in the order of 250 MHz [53].

3.5.2 Handshake Protocol

The simple uni-directional protocol employs a ready-valid handshake to support flow control in both directions [52]. In essence, the master provides data with an associated valid signal and the slave indicates when it is ready by asserting a ready signal. When at a rising clock edge both valid and ready signals are asserted, data is transferred and both signals are deasserted again.

3.5.3 AXI Protocol Derivatives

AXI allows for bursts of data of up to 256 transfer cycles with a single address phase. A derivative is the AXI-Lite standard that is mostly used for low-throughput memory-mapped data transfers. In comparison to AXI, it does not support burst transfers but instead only the transfer of an address-data pair. Another derivative is AXI-Stream, used specifically for streaming applications. The complex address channels are removed and only the ready-valid handshake signals are left. This results in the simplest interface within the AXI family.

3.5.4 AXI Coherence Extension

The AXI Coherence Extension (ACE) [56] is an extension to the fourth iteration AMBA specification introduced in 2011. ACE introduces additional signals to enable system-wide coherence [37].
The introduction of ACE enabled heterogeneous SoCs such as the ARM big.LITTLE architecture. A derivative of ACE called ACE-Lite enables IO coherency in the sense that an attached device can read from the cache present in the ACE-capable host processor.

3.6 Interconnect Comparison

The preceding sections provided information about the state-of-the-art interconnects. This section summarizes their main characteristics and compares them to the desired requirements for future interconnects discussed in Section 2.4.

3.6.1 Bandwidth and Latency

In order to compare various interconnect standards in more detail, the protocols involved and the system that is used should be investigated in more detail. Protocol overhead in transmitted data is for example not taken into account in the comparison presented in Table 3.1, nor are other factors such as packet overhead, and how acknowledge packets incluence bandwidth. System parameters can also affect performance such as payload size and the topology of the interconnect [45]. With a switched topology, packet congestion can occur if multiple endpoints make multiple requests simultaneously. Determining the latency has similar difficulties in that it depends on so many factors.
In general, there is a clear trend to increase the per lane transmission rate in order to provide for example enough bandwidth to accelerators for emerging workloads such as big data and machine learning.

3.6.2 Address Space

There is a trend towards a shared memory model and coherent interconnect with more bandwidth and lower latency. A shared address space across the host and attached devices simplifies the programming model and allows for new use cases.
Since PCI Express Gen 3, ATS is included in the specification and enables an ATC on attached devices. While this improves address translations, it does not share a translation table with the host processor as CAPI and OpenCAPI do. Therefore, emerging use cases are not yet possible on PCI Express nor on CCIX.

3.6.3 Coherence

Hardware-based coherence across host cores and attached devices should greatly simplify the programming model.
PCI Express uses a snoop filter to keep host caches coherent when host memory is accessed by the attached device. While this improves performance by decreasing data access latency, no coherent cache is present on the attached device itself. CAPI does support a coherent cache on the attached device at the cost of an increased protocol overhead. OpenCAPI will support this in the next generation as well, while currently only coherent memory access is provided. The additional protocol layers of CCIX enable cache coherency and the ACE extension works similarly for AXI interconnects.

3.6.4 Synchronization

With access to shared memory by the attached device, synchronization using shared variables between a thread running on the host and the attached device removes inefficiencies from using interrupts or communication using MMIO registers.
PCI Express supports three atomic operations to implement synchronization mechanisms, while CAPI only implements locks. OpenCAPI on the other hand supports multiple atomic operations with a ton of configuration possibilities per operation. This enables a wide variety of synchronization mechanisms to be implemented.

Table 3.1: Comparison of state-of-the-art interconnect standards.

Standard	Bandwidth	Address Space	Coherence	Synchronization
PCI Express Gen 3	16.0 GB/s	ATS	Snoop	Atomics
PCI Express Gen 4	32.0 GB/s	ATS	Snoop	Atomics
PCI Express Gen 5	64.0 GB/s	ATS	Snoop	Atomics
CAPI 1.0	16.0 GB/s	Shared	Cache	Locks
CAPI 2.0	32.0 GB/s	Shared	Cache	Unknown
OpenCAPI 3	25.0 GB/s	Shared	Memory	Atomics
OpenCAPI 4	100.0 GB/s	Shared	Cache	Atomics
CCIX	32.0 GB/s	ATS	Cache	Unknown
AXI[1]	16.0 GB/s	Memory Mapped	Cache	Unknown

3.7 Preliminary Concluding Remarks

It is obvious that many advancements in interconnects have been made recently to bridge the gap between host processor cores and attached devices by tighter coupling and extending common concepts for homogeneous multi-core processors to attached devices.
Due to the desire of backwards compatibility, PCI Express is limited in terms of innovation and protocol changes. This resulted in a slow release of the Gen 4 specification. At the same time, multiple initiatives started to develop new interconnect standards such as CAPI, OpenCAPI, and CCIX.
Due to the support of higher signaling rates than PCI Express, shared memory with address translation and coherent host memory access, OpenCAPI 3.0 is of special interest during the remainder of this thesis. All of these features allow for new usage models and new workloads to be accelerated. In general, upcoming interconnect standards receive a significant increase in bandwidth and this impacts accelerator architectures and design choices.
The objective of the remainder of this thesis is to explore how multiple classes of accelerated workloads can be fed with the same or similar reconfigurable buffer architecture to improve designer adoption of OpenCAPI, or other high-bandwidth interconnects. In order to do so, Chapter 4 provides a deeper understanding of OpenCAPI.

[1] The bandwidth is calculated as the product of a 512 bit wide data bus operating at 250 MHz as used by the SNAP framework [53]. However, higher bandwidths could be obtained by improving either or both parameters.

Chapter 4

OpenCAPI Characterization

To get a better understanding of the first OpenCAPI capable system, the system architecture of IBM's POWER9 processor is presented. While OpenCAPI is host architecture agnostic, the standard will be looked at in more detail with respect to the POWER architecture. Several supported accelerator paradigms are shown and finally two initial compatible Xilinx FPGAs are characterized.
While our research was being conducted, much of the information on OpenCAPI could only be obtained by talking to the right people, and we hope that this overview may prove useful to others who want to gain a better understanding of this interface.

4.1 POWER9 System Overview

OpenCAPI [7] is a successor to CAPI 1.0 [2] and enables direct attachment of any micro-architecture CPU to coherent user-level accelerators like ASICs and FPGAs and I/O devices such as network and storage controllers. The goal is to have a high-bandwidth and low latency interconnect optimised to enable streamlined implementation of attached devices. The first OpenCAPI enabled system will be in the upcoming POWER9 processor by IBM [4].
Figure 4.1 shows the system architecture of a superset of the various configurations. Depending on the model (SU or SO), there are up to 12 SMT8 or 24 SMT4 cores available accompanied by a collection of various on-chip accelerators such as gzip, 842 compression and AES/SHA cryptography engines. The memory controller supports either eight DDR4 channels or eight "Centaur" memory buffer chips that also act as an off-chip L4 cache. This yields sustained bandwidths of at least 120 GB/s and 230 GB/s, respectively.
POWER9 supports a wide collection of interconnect standards. In total, 48 PCIe Gen 4 lanes are available, which can also be used for CAPI 2.0, for a total half-duplex bandwidth of 96 GB/s. Additionally, there are 48 BlueLink lanes operating at 25 Gbit/s servicing either Nvidia NVLink 2.0 attached GPUs or OpenCAPI 3.0 attached devices. This enables an additional 150 GB/s of half-duplex bandwidth. Other POWER9 sockets can be attached through an SMP interconnect, by using 16 Gbit/s or 25 Gbit/s lanes. Finally, the cache hierarchy and on-chip interconnect, called the fabric, ties all units together at a maximum bandwidth of 7 TB/s [19]. Due to the coherent nature of the fabric, attached devices can seamlessly communicate with each other and system memory. What sets the POWER9 apart from other vendors is the extended coherency domain across processor cores and attached devices.

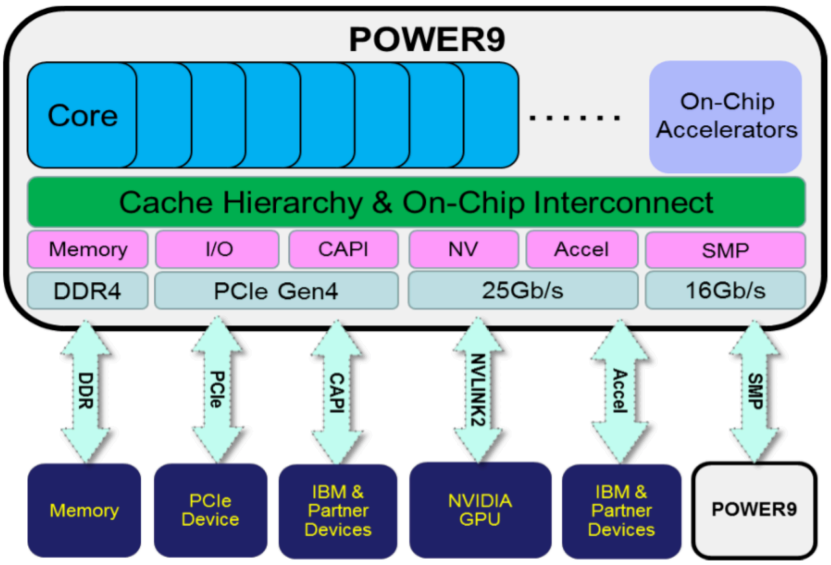


Figure 4.1: POWER9 system overview [4].

4.2 OpenCAPI Architecture

The OpenCAPI architecture consists of several protocol layers divided between the host and attached device [58] [6]. Logic is required on both sides to enable the protocol stack. All logic required for enablement on the host is called the Coherent Accelerator Processor Proxy (CAPP) and is architecture dependent. The CAPP for the POWER architecture is briefly mentioned, but our main focus is on the attached device side. While OpenCAPI also targets emerging storage class memory features, those will not be discussed further and are outside our scope.

4.2.1 Protocol Stack

Figure 4.2 shows the OpenCAPI stack that is partly located on the host CPU and partly on the attached device. OpenCAPI is a credit-based point-to-point protocol. The link has a number of credits that are consumed when data is transferred in order to throttle traffic. While in essence there are no differences from the point of view of the stack between attaching an ASIC or FPGA, only FPGAs will be considered due to the aim of the thesis.
In comparison to CAPI 1.0, the PSL is now located in the host processor, which removes the logic overhead within the FPGA. Complexities of coherence and virtual addressing are implemented on the host CPU to simplify the design of attached devices and facilitate interoperability across different CPU architectures. Due to the coherent nature, attached devices can operate natively within an application's user space. This allows attached devices to fully participate in an application without involvement or overhead of the kernel [19].
The stack consists of several layers that are briefly introduced. Messages (packets) can be initiated by either the CAPP (host) or AP (AFU) and are called CAPP or AP command packets, respectively (see Section 4.2.3). When the Transaction Layer on either side receives a command packet, a response packet has to be sent back, called a CAPP or AP response packet, respectively. Note that while the stack looks symmetric, the transaction layers on both sides are very different, since their implementation depends on the host architecture and AFU protocol, respectively.

- Host Protocol Layer is the coherent connection to the rest of the host. The implementation is host architecture dependent and all required logic to implement OpenCAPI is called the CAPP (see Section 4.2.4).
- Transaction Layer (TL) converts host protocol-specific requests into CAPP command packets and generates and handles CAPP response packets. Internally it consists of a framer and parser.
  - Framer packetizes CAPP commands and responses along with credit packets into control flits according to various packing templates. Different templates exist since packets have different sizes. Control and data flits are sent to the DL while ensuring frame order.
  - Parser receives DL frames consisting of control and data flits (see Section 4.2.2). It parses the control flit into AP command and response packets and returns credits to the Framer. The AP packets are passed to the host.
- Data Link Layer (DL) converts DL frames, consisting of a single control flit and between zero and eight data flits, to PHY transmittable data.
- Physical Layer (PHY) represents the actual connector and link on the host CPU. Each lane operates at 25.781 25 GHz. For the POWER9, each PHY brick consists of eight duplex lanes.

In opposite order, the PHYX, DLX and TLX layers present on the FPGA act similarly to their host counterparts, where X stands for eXternal. However, the TLX does not handle responses. This has to be taken care of by the APL or AFU, depending on implementation.
An additional layer present on the FPGA is the AFU Protocol Layer (APL). This is an optional layer and an AFU can also directly interface with the TLX if desired. An example of this layer could be a bridge to AXI, the de facto standard for FPGAs, similarly to the AXI4 to CAPI 1.0 adapter by Xilinx [52]. On the other side the APL interfaces with the Attached Functional Unit (AFU), or accelerator.

4.2.2 Data Link Layer Frame Format

Typically, packets are broken up in smaller pieces called flits, which stands for FLow control unIT. The first flit is the packet header and contains control information. Other flits, corresponding to the same packet, are data flits. In the context of OpenCAPI, the network packet is called a (DL) frame and the term packets is reserved for CAPP and AP commands and responses. The term TL packets acts as an umbrella for all different packets.
Both the TL and TLX framer and parser work with DL frames [58]. A DL frame consists of one control flit and between zero and eight data flits, where at most four data flits belong to a single TL packet. Every flit is 64 bytes in size. Flits are transmitted starting at the lowest order control flit bytes and continuing in increasing address order. After that, the data flits are transmitted similarly. A control flit consists of an 8 byte DL content field and 56 bytes of TL packets, packed according to a predefined packing template. The OpenCAPI 3.1 TL specification [6] adds a datum field to the control flit which embeds data smaller than 64 bytes within the control flit for improved frame utilization.

The DL content field contains both DL and TL generated subfields. Important are the DL injected CRC and TL injected TL template subfields. The CRC covers both the current control flit and the data flits from the previous control flit. Upon detection of an error, all data flits

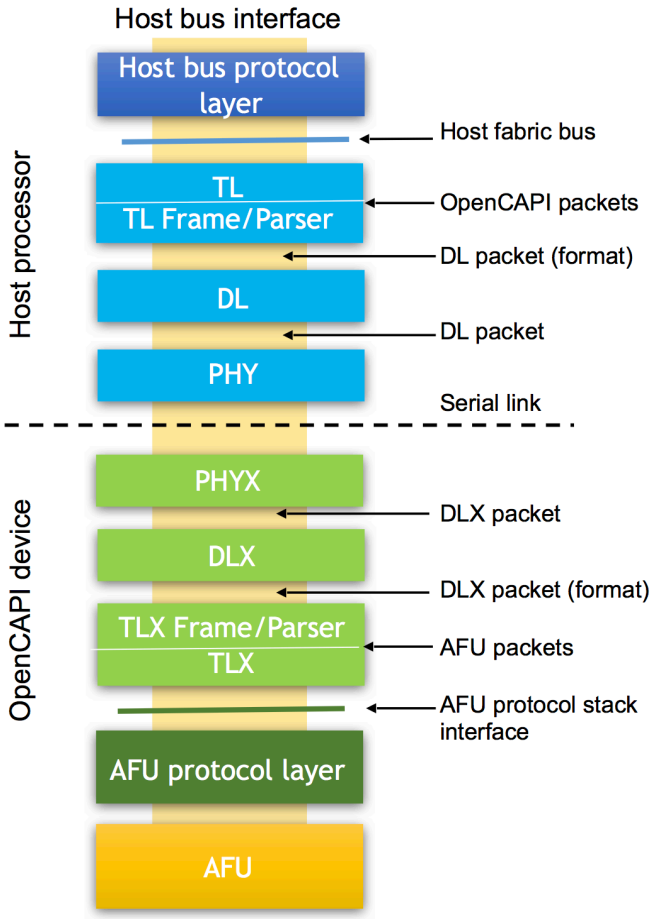


Figure 4.2: The OpenCAPI stack from host to attached device [5].

from the previous control flit are invalid and the transmitter is requested to replay the data flits. The final control flit never has any data flits, since it has to validate the last data flits using its CRC.
Figure 4.3 shows several DL frames and their respective control and data flits. The same colored flits indicate how a CRC in the DL content field corresponds to the data flit(s) from the previous control flit. The TL template subfield specifies the location of TL packets in the remainder of the control flit. The 56 bytes, or 448 bits, of TL packets in the control flit consist of sixteen, 28 bits, slots. TL packets differ in length and can consume between one and six slots per packet. Different package templates exist that indicate how many TL packets are present and how many slots they consume.

4.2.3 Transaction Layer Packets

TL packets are control instructions that can be sent within a control flit [6]. Depending on which side of the OpenCAPI link initiated the packet, the prefix CAPP (for the host) or AP (for the AFU) is used. There are several different types of packets, depending on the source. The host can issue from the following categories of commands. Bear in mind that each command requires a response (not listed below).

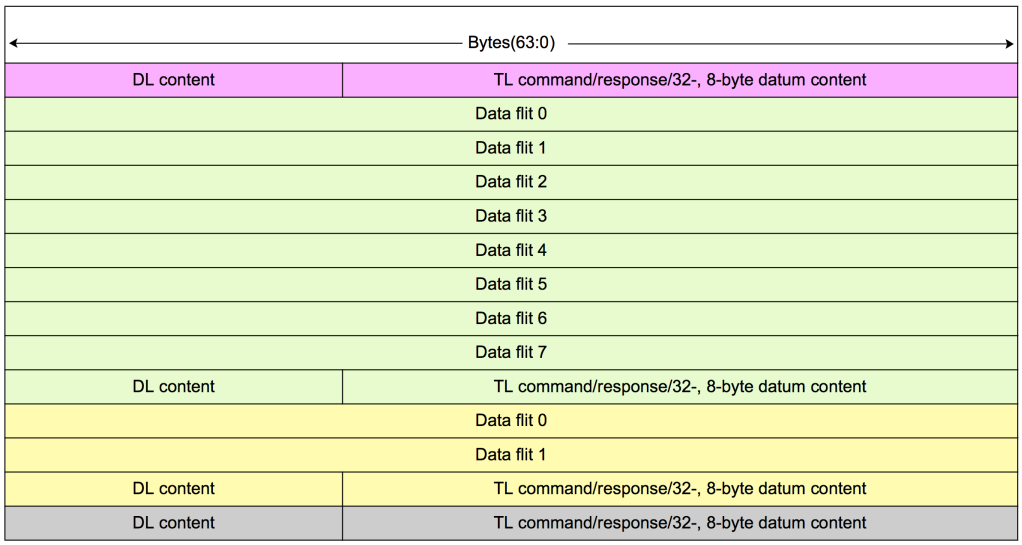


Figure 4.3: Example of several DL frames showing the CRC control and data flit coverage [6].

- Address Translation The host notifies the AFU that a previous address translation request has been completed.
- Configuration Space Reading and writing to the configuration space of the AFU is supported with specific commands.
- Interrupt The host updates the AFU regarding a previous interrupt request.
- Memory Access The host can read and write AFU memory at 64, 128 or 256 bytes data sizes at a time. It supports partial reads and partial and byte-enabled writes.
- Metadata An optional and implementation specific field to hold metadata for a data block held in memory.

The AFU supports a different set of command categories.

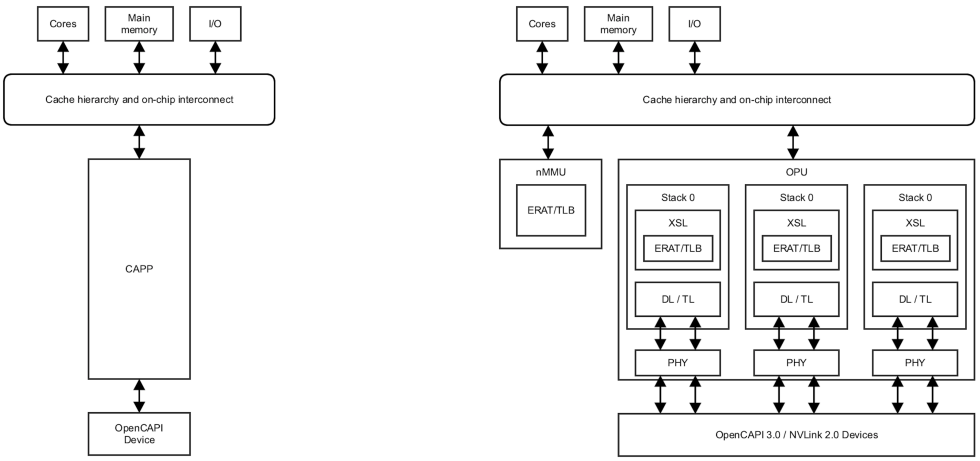
- Address Translation The AFU can request address translation prefetching for an EA to warm up the address translation cache. This allows an accelerator to reduce translation latency when using a new page.
- Assign acTag The Address Context Tag (acTag) is used to index a host table containing the associated BDF and PASID. The BDF and PASID are used for address translation authorization and operation validation.
- Atomic Operations are supported to host memory (read, write, read-write). An accelerator can perform atomic operations in the same coherent domain just like any other host processor thread. Multiple variations are supported by hardware.
- Interrupt The AFU can request interrupt service on the host.
- Memory Access Currently, the AFU has no coherent cache. Therefore, read commands have a suffix to indicate no intent to cache. A coherent cache will be supported in OpenCAPI 4.0. A partial read is supported, as well as byte-enabled and partial writing.

- Wake Host Thread is an efficient low-latency mechanism used for communication between the host and attached device instead of either interrupts or host polling mechanism of host memory.

Both the CAPP and AP side support a response packet for returning credits. As mentioned earlier, these are used for flow control.

4.2.4 Coherent Accelerator Processor Proxy

The CAPP, in a host architecture agnostic context, contains all logic required on the host to enable OpenCAPI. Figure 4.3a shows a possible system architecture where the CAPP is coherently connected to the rest of the host. An OpenCAPI device is then connected to the other side of the CAPP. Note that module names used here might change in the official documentation. Figure 4.3b shows CAPP implementations for the POWER9 [59]. It includes the OpenCAPI Processing Unit (OPU) and Nest Memory Management Unit (nMMU). The OPU consists of three stacks and each stack services two physical bricks of eight lanes. This brings the total to 48 lanes. Each stack consists of an XSL, and the DL and TL layers. The XSL handles address translation and has a dedicated ERAT/TLB of 64 entries and each entry represents a 64 kB page. Two stacks support OpenCAPI and are statically configured to use either OpenCAPI or NVLink 2.0 DL and TL layers, that in turn are connected to the PHY. The nMMU handles translation requests from other units than the CPU cores. It has its own ERAT/TLB with 8192 page entries, significantly more than the XSL.



(a) Host architecture agnostic. (b) POWER9 architecture specific.

Figure 4.4: System level view of OpenCAPI enablement on the host using the CAPP.

4.2.5 OpenCAPI Attached Device

The CAPP is provided by the host architecture. On the FPGA side, the physical layer is supplied by the FPGA card vendor. Both the DLX and TLX are implemented using configurable resources on the FPGA and a reference design is provided by the OpenPOWER Foundation. The APL is an optional layer between the TLX and the AFU. Based on experience and customer feedback of CAPI 1.0 [59], the OpenCAPI consortium decided to supply no specific interface between the TLX and APL in the sense that there is no cache or buffer present that the AFU can talk to directly. Instead, it provides an interface where TL packet opcodes can be sent to

or received from the host.
Figure 4.5 shows the presented TLX interface from a high level [59]. The TLX consists of a framer and parser, just like the TL. The parser receives frames from the DLX, unpacks the TL packets and splits the command and response packets, each presented at a separate interface. Each of these two interfaces consists of parsed control information from each TL packet and corresponding data payload. The data payload interface is 64 bytes wide, the same size as a data flit. If the payload of a single TL packet is more than one data flit, it takes multiple cycles to receive the entire data payload. The framer has similar interfaces, but packs TL packets instead to form control flits. There is also a configuration module present on the AFU which holds registers for configuration of the TLX. For example, to enable certain packing templates. There are separate interfaces for this module, not shown in the figure. The TLX also manages credits and each interface has a credit interface in opposite direction. The TLX parser also gives credits back to the TLX framer.
The latest generation of Xilinx FPGAs are used that allow an increased operating frequency of up to 450 MHz. To minimize latency, the target frequency of the DLX and TLX is 400 MHz. Each of the four data interfaces can supply up to 64 bytes per cycle at the target frequency. Typically, highly pipelined FPGA designs can reach up to 250 MHz. If an AFU operates at 200 MHz, it will seem like OpenCAPI provides 128 bytes per cycle. This is also the size of a cache line in the POWER architecture.

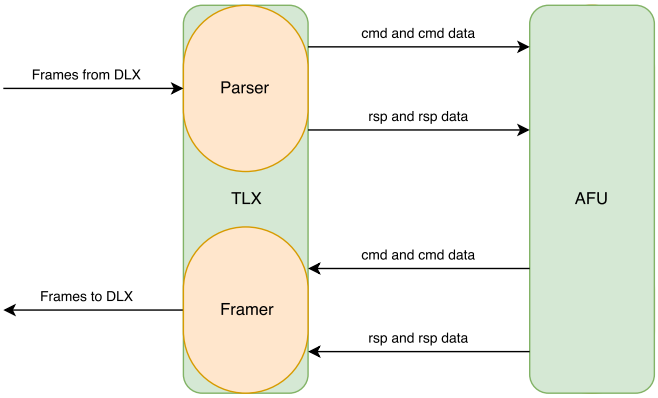


Figure 4.5: Interface between the TLX and APL or AFU, depending on the AFU design.

An OpenCAPI device may have three physical address spaces. The configuration space is in a separate space from the MMIO and system memory spaces. These spaces share a PA space and the host can differentiate between the two since the system memory space always starts at offset zero of the PA, while the MMIO space starts at a fixed configured offset from zero. The MMIO offset is configured using a BAR (Base Address Register) and multiple BARs are present to service multiple attached devices.

- Configuration space may be used for configuring the TLX or AFU. It is accessed by using the dedicated read and write commands. A reference configuration space module will be provided by the OpenCAPI consortium.

- MMIO space may be used for configuration of the AFU and is mapped in the system memory address space.

- System memory space is a memory space owned by the OpenCAPI device and is mapped to the system's RA space. This memory is coherently accessed using the load/store model used by the host.

A typical usage model is to write a work pointer in an AFU MMIO register or by communication through shared memory. The MMIO module is flexible in the sense that the MMIO base address register and sizes can be configured by the AFU designer. Also configuration registers are present on a per-process basis. The MMIO module is provided by the OpenCAPI consortium and can be integrated directly within a design.

4.2.6 Address Spaces and Translation

While address translation is present on the host and managed by hardware, it is of interest to the AFU designer since it can greatly influence performance regarding translation misses. Page table walks are very expensive and take many cycles to complete [59]. Therefore tuning the AFU to optimally use the TLB and warming up the TLB is a good practice.

Address Spaces in the POWER Architecture

Three different address spaces exist in the OpenCAPI and POWER architecture.

- Effective Address (EA) is the address seen by a program, also known as a virtual address.
- Real Address (RA) is the address used to access the entire system address map. The entire map consists of physical DIMM memory, PCIe memory, GPU memory, etc. Each of the regions has a base and size region of the RA that maps to it. These addresses are for example used within the fabric on the POWER9 to communicate between caches and DIMM memory.
- Physical Address (PA) is the address used by a physical memory source, such as a DIMM or local memory of an attached device. You can think of the physical address as the RA minus the base. It maps directly to a location within the memory device.

The system memory address space includes all addresses within the system and uses real addresses. Main memory is the portion that is normally backed by physical DIMMs and marked coherent. This can be cached by processor caches and coherency maintained. MMIO (memory mapped I/O) is mapped into the system memory map, but it is marked in the page tables as non-cacheable. This includes the PCI Express devices MMIO regions, AFU MMIO regions, as well as the processor MMIO addresses. MMIO regions consist usually of registers and are used for configuration of the system and communication with the device driver for I/O devices. It is a region in the system memory map because it is accessed across the fabric from a program running inside a core to communicate with the attached device via EA-RA translation.

Address Translation

Taking a look at Figure 4.1 again, the fabric uses real addresses. If a memory location within a DIMM is accessed, the memory controller resolves a physical address from the real address. Cores have their own MMU in order to translate an effective address to a real address via page/segment table walking. An effective address will be translated to a real address if a memory location has to be accessed outside of the current core.
In OpenCAPI, all translations happen on the host and occur either in the OPU or nMMU.

In CAPI 1.0, part of the translation was located on the FPGA (in the PSL). Moving the address translation to the host reduces design complexity of attached devices. Since an attached device never has access to a physical address, malicious attached devices are not able to access unauthorized memory locations, such as addresses belonging to kernel or other applications.
The AFU only uses effective addresses for mastering commands to host memory, that are translated on the host to real addresses. When the AFU acts as a slave and receives commands from the host, physical addresses are used to access the three different physical address spaces mentioned in Section 4.2.6. These physical addresses have been translated from program EA, to system RA, to PA. The host converts the RA to a PA using configuration settings in the host that are determined during initialization of the attached OpenCAPI device.
Real addresses are mapped into the physical address space specified for an OpenCAPI device. This eliminates any requirement placed on the OpenCAPI device to have knowledge of the host's real address space or how the OpenCAPI device's PA space is mapped into it. The PA for CAPP commands is all translated in the host (MMIO, Config, host memory). The host has programmable base address (BAR) and offset registers for everything. The PA space in the FPGA is either config space (indicated by command type), MMIO (indicated by matching the AFU's base address), or host memory (if it doesn't match the AFU's BAR).

Address Translation Example

As an example, consider a POWER9 with an OpenCAPI-attached device. A program on the processor only sees an Effective Address for addressing the system address map. If the program requests the AFU to fetch data from host memory it uses an EA. The AFU sends the EA across the OpenCAPI link and the XSL translates the EA to a Real Address (RA). To do this, first it looks in its ERAT. If it misses the XSL ERAT it forwards the request to the nMMU. The nMMU will look in its ERAT before walking the page table in host memory to resolve the translation into an RA. If the page walk fails it will return a bad status to the OPU, and it will generate a fault interrupt to resolve the fault.
In the other direction, a program uses an EA and the OS sets up mappings between a page to a physical resource. To communicate across the fabric, the program EA is translated to an RA. The RA is then matched to a range within the OPU that is configured as AFU memory and translated from RA to PA.

4.3 Coherent Programming Model

OpenCAPI enables new, easier and more natural programming models, as found on multi-core systems, for IO that was not possible with the traditional approach. Attached devices are more easily accessible due to the shared virtual memory space and appear as peers to the processor cores. Also the setup and completion phases, by interacting with device drivers, have been simplified and made faster. Combining this with attaching devices with a low-latency interconnect, the attached devices become tightly coupled that allows for thread-level parallelism between the application running on the host and the attached device.

4.3.1 Coherent Shared Virtual Memory

The approach of OpenCAPI (and CAPI for that matter) offers a virtual address space shared between the processor caches and the attached device [7]. The host and accelerator can coherently access each others physical memories. This removes the problem of having multiple copies of the same data in a traditional IO model (see Section 2.3.1) and reduces setup and completion time significantly for an application wanting to use the attached device. Typically, this might

take 12.8 µs as shown in Figure 4.6. Now it might take as little as 0.36 µs.
Not only can the host use atomic operations, but also the attached device has a vast set of atomic operations to implement synchronization operations. A special wake host thread opcode can be used for communication as well.
The shared virtual memory space also allows programmers to dereference pointers everywhere, while previously host pointers could only be dereferenced on the host and vice versa [39]. This removes the manual movement of data between the host and attache device. Overall it simplifies the programming model since the attached device operates as a peer to the other processor cores. With the traditional IO model, sharing an attached device between applications was difficult because pinned memory belongs to only one application and cannot be shared. If the attached device supports multiple hardware buffers, multiple applications could use the device. The number of applications is dependent on the hardware. OpenCAPI allows for sharing the accelerator between applications due to a special feature in the standard (context bits).
With CAPI, a coherent cache was present on the accelerated device. This allows for even more programmer flexibility and lower latency for highly referenced data. However, this feature is absent in OpenCAPI 3.0 but will return in OpenCAPI 4.0.

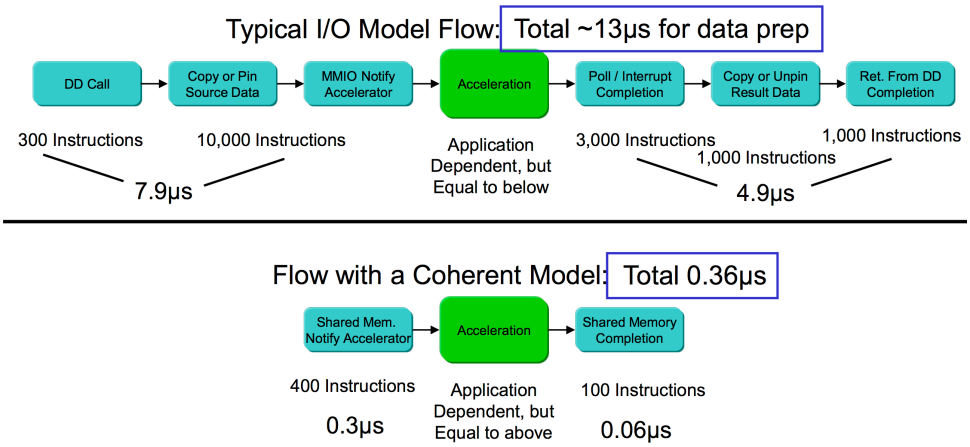


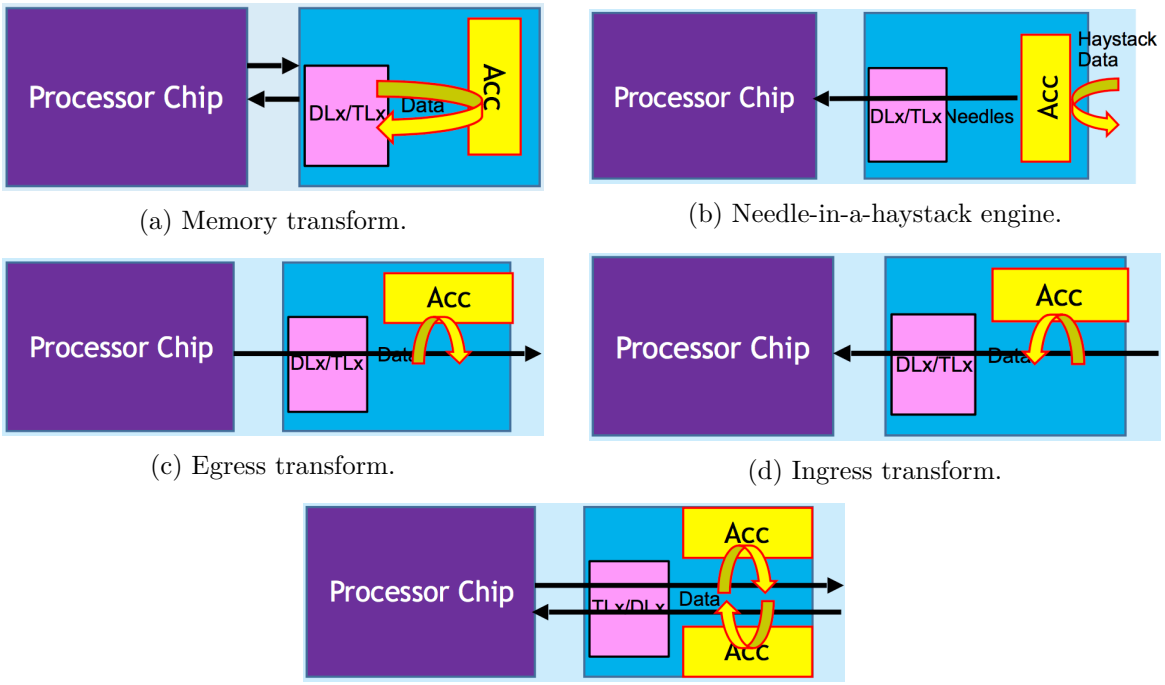
Figure 4.6: Traditional IO device setup and completion flow versus the OpenCAPI flow [5].

4.3.2 Accelerator Paradigms

Traditionally, an offload paradigm was used for accelerators, shown in Figure 4.7a. Figure 4.7 shows current and new paradigms enabled by OpenCAPI. A perfect example is an application that uses pointer chasing or linked lists [54]. This was not possible because pointers were not able to be dereferenced since the processor and IO device did not share the same address space. Other applications could include using both the shared host memory and local memory of the accelerator, since accessing host memory has a very low latency. An example of such a bi-directional accelerator is shown in Figure 4.7e.

- Memory transform is the traditional offload paradigm. GPUs fall into this category.
- Needle-in-a-haystack engine processes a large data set from disk for example and filters specific data of interest.
- Egress transform processes outgoing data, as it leaves the system. Examples are compression and encryption on its way to centralized storage.

- Ingress transform processes incoming data, for example from a NIC, and places the output in host memory. An example could be decompression and decryption.
- Bi-directional transform can be used for a hash-join database operator using a database located on disk. The hash table can be build in host processor memory. Stream through data from disk and do computations that while fetching hash table from host memory.



(a) Memory transform.

(b) Needle-in-a-haystack engine.

(c) Egress transform.

(d) Ingress transform.

(e) Bi-directional transform.

Figure 4.7: Accelerator paradigms enabled by OpenCAPI [7].

4.4 FPGA Characterization

FPGAs are integrated circuits that can be reconfigured after fabrication (field programmable). A lot can be said about FPGAs and their operation, but the focus of this section is to provide background information for those aspects of FPGAs that most directly relate to the implementation of the proposed architecture. The background information provided in this section will be used throughout the rest of the thesis.

4.4.1 FPGA Architecture

Typically, FPGAs consist of arrays of programmable logic blocks that can be wired together. Logic blocks can be used to implement combinatorial logic, by configuring look-up tables (LUTs), or to implement sequential logic such as flip-flops and small memories. Besides configurable blocks, also hardwired logic is present such as multiplexers, special DSP slices, networking stacks, and PCI Express controllers.
Figure 4.8 shows the physical architecture of an FPGA. The most common resources such as IO Blocks, CLBs, memories, and DSP Blocks are shown. What is important to notice is that each type of resource is located in a separate column. Memories for example could be located relatively far away from where their data is processed. Depending on the target frequency of

the design, wire delays could start to play a large role. When routing within a single clock cycle fails, additional registers are required between a memory and the data consumer. For this reason, memory primitives typically contain one or multiple additional pipeline stages within the primitive to help with routing, at the cost of increased latency.

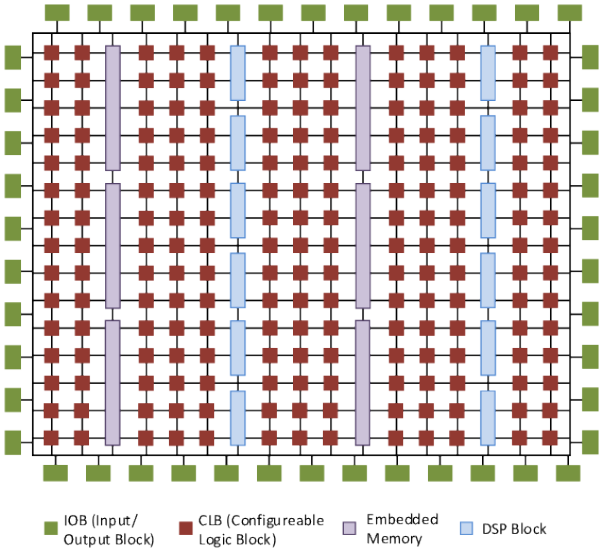


Figure 4.8: Diagram showing the physical architecture of an FPGA [8].

4.4.2 Typical Resources

We study two Xilinx FPGAs for their suitability to be used as OpenCAPI accelerators. Both FPGAs, the KU15P and VU37P, are from the latest UltraScale+ architecture and are the highest model in each device family, Kintex+ and Virtex+, respectively. Compared to previous architectures, UltraScale+ adds Ultra RAM (URAM). This memory resource falls in between the typical BRAM and DRAM capacities, and High Bandwidth Memory (HBM) Gen 2 for the top tier Virtex+ FPGAs. Table 4.1 shows a summary of specifications for both FPGAs. The GTY transceivers mentioned in the table have a maximum bandwidth of 32.75 Gbit/s, which is more than that of the OpenCAPI lanes at 25 Gbit/s. The table shows that both FPGAs can easily handle eight lanes to attach to OpenCAPI. The following itemization explains most of the specifications in more detail. It is important to note that the VU37P excels in every aspect compared to the KU15P.

- CLB flip-flops reports the number of flip-flops across all CLBs in thousands.
- CLB LUTs reports the number of LUTs across all CLBs in thousands.
- Distributed RAM reports the maximum memory capacity.
- BRAM reports the total memory capacity, both with and without ECC support.
- URAM reports the total memory capacity, both with and without ECC support.
- HBM reports the total available HBM capacity on the FPGA.
- DSP slices reports the number of available DSP slices on the FPGA.
- GTY transceivers reports the total number of 25 Gbit/s transceivers available.

Table 4.1: Specification summary of the Xilinx KU15P and VU37P FPGAs [10].

Resource	KU15P	VU37P
CLB flip-flops [k]	1045	2607
CLB LUTs [k]	522	1304
Distributed RAM capacity [Mb]	9.8	36.7
BRAM capacity with ECC [Mb]	34.6	70.9
BRAM capacity without ECC [Mb]	30.8	63.0
URAM capacity with ECC [Mb]	36.0	270.0
URAM capacity without ECC [Mb]	32.0	240.0
HBM capacity [GB]	0	8
DSP slices	1968	9024
GTY transceivers	32	96

4.4.3 Configurable Logic Blocks

The UltraScale+ architecture consists of an array of Configurable Logic Blocks (CLB), with two distinct flavors [9]. Each CLB consists of eight six-input LUTs, sixteen flip-flops, and seven hardwired multiplexers to select between LUT outputs if needed. A LUT is used to implement a logic function and each of them is accompanied by two flip-flops and can be configured as either a six-input one-output or a five-input two-output LUT. Besides the hardwired multiplexers, the LUTs can also be used to implement a 4:1 multiplexer by using four inputs for data and two inputs for the selection signal. Figure 4.9 shows how a 16:1 multiplexer can be implemented in half of a CLB. Similarly, a 32:1 multiplexer can be implemented in a single CLB by using all LUTs and hardwired multiplexers. Internally, a CLB can either be a SLICEL or a SLICEM. The previously mentioned logic is present in a SLICEL. In addition to this, a SLICEM can also be configured with up to 512 bit of distributed RAM or with a 256 bit shift register. Distributed RAM can be configured as single, dual, quad, octal or simple dual-port with a minimum memory capacity of 32 bit per primitive up to a maximum of 512 bit per slice with either 1 or 2 bit wide data.

4.4.4 Memory Resources

Several different memory resources exist, either inside or outside of the FPGA. The UltraScale+ architecture brings two additional memory resources to the table: URAM and HBM. The following list is sorted based on increasing memory capacity and latency [60].

- Distributed RAM is RAM built from LUTs within a slice and supports several configurations. Multiple read ports are supported, but multiple write ports are not.
- Block RAM are dedicated RAM primitives with ECC support of 36 kbit in size. The primitive consists of two independent 18 kbit RAMs that can be configured as a Simple Dual Port (SDP) or True Dual Port (TDP) memory. Each BRAM has two independent read and write ports. A 36 kbit BRAM can be configured with independent ports as 32K x 1, 16K x 2, 8K x 4, 4K x 9, 2K x 18 or 1K x 36 for a TDP configuration or additionally as 512 x 72 for an SDP configuration. An 18 kbit BRAM can be configured with the same data widths, but with half of the entries. By default the read latency is one cycle, but an optional read register can be configured.
- Ultra RAM are also dedicated RAM primitives with ECC support but have a larger capacity compared to BRAM primitives. Two ports are available and one configuration

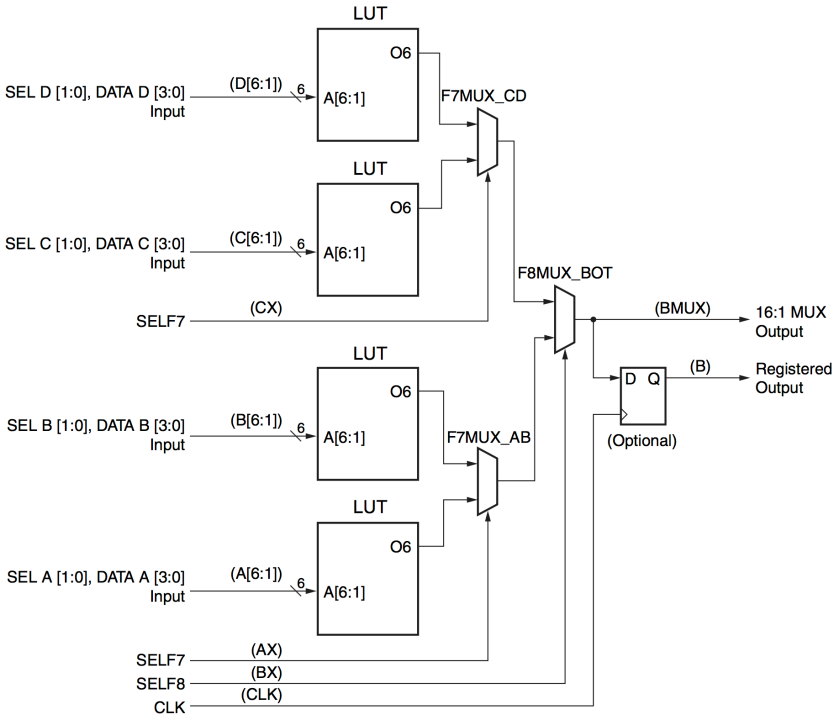


Figure 4.9: 16:1 multiplexer using half a CLB in the UltraScale+ architecture [9].

of 4K x 72. A port either operates as a read or write port and port A always completes before port B does. Similarly to a BRAM primitive, optional registers can be configured between one and four cycles of latency.

- HBM is high-bandwidth memory reaching bandwidths up to 460 GB/s. This memory is only supported on top tier Virtex+ FPGAs up to 8 GB [61].

- DRAM is located outside of the FPGA and has a capacity of up to several gigabytes.

BRAM Address Collision

An address collision occurs when both ports of a BRAM primitive access the same address in a single clock cycle. The resulting behavior depends on the port configuration. If both ports read, both accesses complete successfully. If both ports write, the memory location contains non-deterministic data. If one port reads and the other writes, the write data operation completes successfully. The read access is only successful for common clock designs and the write port is configured as read-first [60].

URAM Address Collision

Similarly, an address collision can also occur for a URAM primitive. When both ports write in the same clock cycle, the port B write takes effect since port A always completes before port B. If port A reads and port B writes, port A receives the old data and the new data is stored at the memory location. If port A writes and port B reads, the new data is written to the memory location and port B reads the new data immediately [62].

4.4.5 DLX and TLX Reference Design

As mentioned in Section 4.2.5, a DLX and TLX reference design is provided that can be integrated as a module by the AFU designer. Because the reference design is implemented in FPGA logic, care must be taken that both the reference design and the AFU fit inside the specific FPGA resource budgets. Table 4.2 shows the resource utilization obtained using Vivado 2017.1 when targeting a Xilinx VU3P FPGA on the Alpha Data 9V3 OpenCAPI-capable card. The number of resources used and the percentage consumed of the total number of resources available are shown. These results are from October 10, 2017 and are subject to change [5].

Table 4.2: Resource utilization for the DLX and TLX layers [5].

Resource	DLX	TLX	Total
CLB Registers	9392 (1.19%)	13806 (1.75%)	23198 (2.94%)
CLB LUTs	19026 (4.83%)	8463 (2.15%)	27489 (6.98%)
LUT as Memory	0 (0%)	2156 (1.09%)	2156 (1.09%)
BRAMs	7.5 (1.0%)	0 (0%)	7.5 (1.0%)

4.5 Preliminary Concluding Remarks

The bottlenecks present in the traditional IO model are addressed by OpenCAPI and this chapter provided a deeper understanding of this protocol. Additionally, various emerging use cases for attached accelerators are presented. A brief summary of the latest generation of Xilinx FPGAs, compatible with OpenCAPI, is required for the remainder of this thesis. Due to the order of magnitude increase in interconnect bandwidth, new challenge arise for AFU designers. Questions regarding the partitioning of the algorithm, concurrent usage of multiple software threads, and the design of the accelerator come to mind.
Another question we can ask ourselves is how the accelerator is fed with data, since no buffer or cache is present on the attached OpenCAPI device by definition. Chapter 5 generalizes across multiple common accelerator memory access patterns and proposes an architecture to keep accelerators fed with data at OpenCAPI-like bandwidths.

Chapter 5

Requirements and Naive Designs

Section 3.7 concluded that recent advancements in interconnect standards try to tightly couple attached devices to host processor cores at an order-of-magnitude larger bandwidths. This has serious implications for accelerator design and, more importantly, for feeding them.
This chapter compares common accelerator memory access patterns and tries to generalize across several streaming-like access patterns. This benefits the data feeding architecture, since it will be applicable to a wider range of accelerators. Thereafter the merge-sort database operator is used as a case study to show that naive traditional design methodologies at these bandwidths will not suffice.

5.1 Accelerator Classification

As mentioned in Section 2.1.5, FPGA accelerators are most commonly used for a specific class of workloads. Therefore the memory access patterns found in these workloads are limited. The following list shows the most commonly found memory access patterns of accelerators including an example application [63].

- Complex accesses are considered to be more difficult than for example strided access, but still regular and known a priori. An example is a Hessian computation found in augmented reality. Image processing also uses similar access patterns.
- Gather accesses multiple pieces of data from non-contiguous locations in host memory. Each request consists of an absolute address and an amount of data to retrieve. This type of access often occurs for vector arithmetic.
- Indirect array accesses an array using a second array: $A[B[i]]$. The data for $B[i]$ is retrieved and the returned value is used to access array A. An example can be found in calculating a histogram in image processing.
- Linked-list reads an address that points to another address and so on. An example could be a network controller that handles header and payload data structures. The header contains information regarding the payload and is typically stored as a linked-list.
- Streaming accesses continuous chunks of data from host memory and stores it into a local buffer for processing. Examples include encryption and video processing.
- Strided accesses chunks of data with a fixed distance from the current address, called strides. Matrix multiplication is an example of a strided access since columns of the matrix have to be read where the stride is the row size.

Typically a DMA engine is used to direct memory access between the host memory, over the interconnect, and the local memory on the accelerator. A DMA enables compute to operate in parallel with memory transfers and provides data in large spatial-continuous blocks of memory to the accelerator. From this list, it becomes apparent that only streaming access directly benefits from a typical DMA transfer. The other access patterns do not, since they require either spatial-continuous data from multiple starting addresses or a single element from multiple starting addresses. For the first case, multiple DMA transfers have to be issued sequentially or a scatter-gather engine could be used, where a list of transfers is provided. The latter case requires buffering, since transferring data element-wise is inefficient over the interconnect due to under-utilization of the available data bus. The complex access for example, would even benefit from less than a cache line[1] granularity of data. For the complex access, element granularity is preferred, where an element is the data size of a single piece of meaningful data. In the image processing example, the data of a single pixel. Element granularity access is not supported by current frameworks such as the Streaming Framework or SNAP, discussed in Section 3.2.2.
However, a generalization across several streaming-like access patterns can be made. This renders the buffer architecture applicable to multiple access patterns and takes it out of the design process of an AFU designer. While the three database operators mention in Section 1.1 are streaming based, a smaller granularity allows for more flexibility, especially with other access patterns in mind. Streaming access is the simplest pattern, because spatial-continuous data is read from a single starting address. A similar pattern is the strided access because in essence it consists of multiple stream accesses at the same time, where the starting addresses are at equidistant. A generalization of the strided access is the gather access, where also multiple non-spatial-continuous locations in memory are read. However, the starting addresses do not necessarily have to be equidistant. Finally, the complex but regular access is a generalization across all of the previous patterns. It could require element-sized data from multiple memory locations at the same time. A buffer architecture with element-wise access enables, for example, the Hessian matrix complex access case. Each pixel required for the computation is located in a separate stream buffer, after which the accelerator is able to access each stream at any point in time.

As mentioned in Chapter 1, other students are studying three different accelerators for database operators: decompress-filter, hash-join, and merge-sort. In essence, each accelerator requires streaming access, but in different ways. The decompress-filter operator decompresses incoming compressed Parquet files and applies a certain filter to it. This accelerator exhibits perfect streaming access behavior. The merge-sort operator merges multiple pre-sorted key-value pair streams from main memory and merges them into a single sorted output stream. In the case of the hash join, while hash table accesses are irregular, an efficient implementation of hash-join will hold the hash tables in local memory such that the accesses to host memory are predominantly streaming.

5.2 Merge-Sort Accelerator Case Study

The merge-sort operator is an interesting database operator since it inherently uses multiple streams of key-value pair data from host memory. The complexity of the operator comes from the merging of multiple streams, where the next chosen stream to read is unpredictable. For databases, key-value pairs are a common data type. A key is a unique identifier with a size of 8 bytes for example. Most important for the key size is that it is large enough such that no

[1] The data size transferred between the last level cache and host memory.

collisions will occur. The value could be actual data or a pointer to the actual data. Since the value could be a pointer, 8 bytes are sufficient to hold a 64 bit address. Combining both the key and value size results in a 16 byte key-value pair, referred to from now on as an element, and is the smallest amount of meaningful data.
As mentioned in Section 4.2.5, the cache line size for the POWER architecture is 128 bytes. This means that each cache line holds eight of such elements. A reasonable, but extreme, use case for a merge-sort operator would be to assume a system with 1TB of host memory capacity. The latest generation of Xilinx FPGAs support up to 8 GB of HBM (or RAM, but our goal in this thesis is to keep up with the OpenCAPI bandwidth rather than HBM bandwidth). In order to merge-sort all pre-sorted streams in one pass, a total of 128 streams are needed. Sustaining throughput is difficult when using a single stream. The inherent requirement of multiple streams makes it easier to fully utilize the interconnect bandwidth since requests can be made concurrently. For example, in the scenario where eight elements from different streams are requested, a typical cache line granularity interface has to fetch eight cache line granularity data blocks, select the requested element from each cache line, and buffer or discard the rest. By using read ports with element granularity, no data is wasted.

5.2.1 Naive Buffer Design

Usually a buffer is placed between the incoming data interface and the data consumer (accelerator) since the availability and consumption of data may happen at different rates. A buffer is also used to hide the latency of the interconnect by placing the data close to the consumer. Since the addresses of all cache lines are known a priori for the streaming and the streaming-like patterns discussed before, the cache lines can be easily pre-fetched to keep the buffers filled without having to flush the buffers since no data is fetched speculatively.
In order for the accelerator to keep up with the OpenCAPI bandwidth, it has to consume 128 bytes (cache line size) per cycle. The interface provided by the TLX to the AFU designer consists of 64 byte data buses, operating at 400 MHz, over which 64, 128 or 256 bytes of data are transferred (taking one, two or four cycles, respectively). Since a cache line is 128 bytes in the POWER architecture, transferring less than that seems wasteful. When 64 bytes of data are requested for example, the CAPP fetches the requested 128 byte cache line and invalidates one half and transmits the other.
OpenCAPI supplies 64 bytes per cycle at 400 MHz while a realistic target frequency for an accelerator is 200 MHz. To cover an interconnect latency of 1 µs per stream (see Section 7.2.1 for a more detailed explanation), 200 cache lines, or 256 when rounded up to the nearest power of two, have to be buffered on the FPGA per stream. Due to the unpredictable access pattern of the merge-sort, each stream has to be able to be read, thus have its own buffer, in any given cycle. The total buffer size B for such an architecture can be calculated as shown in Equation 5.1.

$$B = N \times C \times L \implies 128 \times 128 \times 256 = 4\,\text{MB}, \text{where} \tag{5.1}$$

- N is the number of streams,
- C is the cache line size in bytes, and
- L is the latency to be covered for OpenCAPI (rounded up to the nearest power of two).

As shown in Section 4.4, the Xilinx FPGA from the Kintex+ product line has a BRAM capacity of 4 MB and a URAM capacity of 4.2 MB. This means that all BRAM or URAM resources are consumed by this naive buffer architecture, leaving no resources for additional control, and more

importantly, the actual AFU. Both N and L in this equation are subject to change and pose a trade-off. Either have fewer streams or buffer fewer cache lines per stream. Fewer streams may not be problematic, depending on the application. Buffering less per stream means that for a worst-case latency scenario, the AFU has to stall since there is no meaningful data present in the buffer. This might not be a problem if the access pattern of the respective AFU is evenly distributed across all available streams (as is the case for the decompress-filter). However, in the case of the merge-sort, the access pattern is unpredictable across streams. In order to maintain the OpenCAPI throughput and support the worst-case, the number of streams is reduced to 64. This configuration consumes roughly half of all the available BRAM resources instead of all of them.

5.2.2 Crossing the Cache Line Boundary

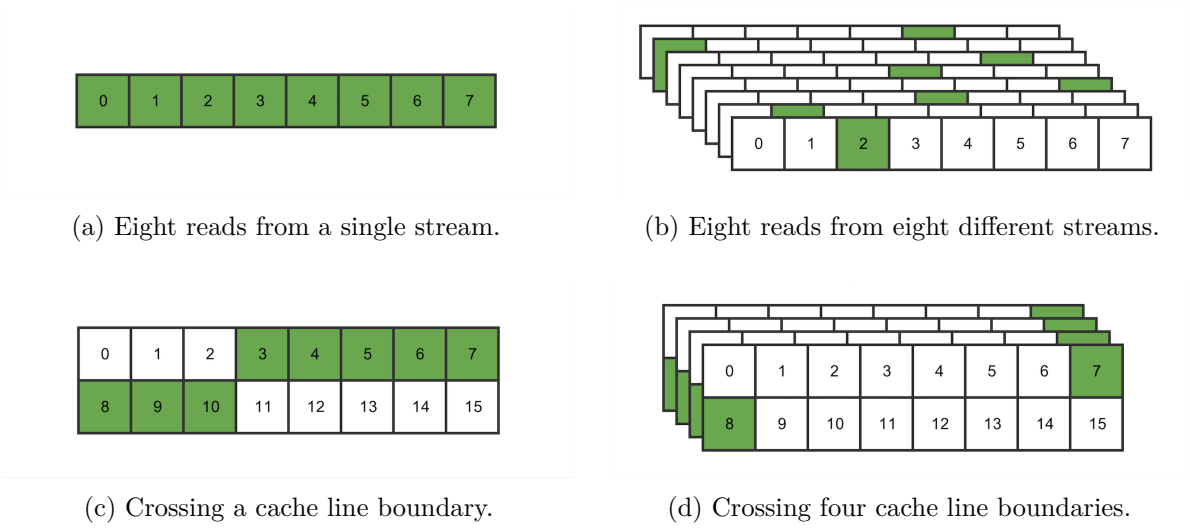
As previously mentioned, each cache line consists of eight key-value pairs or elements. Due to the absence of a cache or buffer in OpenCAPI 3.0, multiple smaller than cache line data granularity read ports are desired and would make the buffer architecture also more general. A cache would be nice but is difficult to keep coherent in the current state of OpenCAPI 3.0. Relying on software is expensive and results in overhead and increased latency. Because streaming-like patterns are targeted, a buffer is sufficient. In the future, the buffer could be extended to a cache.
In order to accommodate element-wise read granularity, the buffer has to have eight read ports to consume a cache line size of data every cycle. Each read port has to be able to supply an element from any of the available streams. Due to the nature of streaming access, the next read from a stream is always the next element. In essence there are four particular access patterns, shown in Figure 5.1, that are interesting to explore, since the buffer architecture has to be able to handle those access patterns. In this figure, each numbered block represents an element in a cache line. The number represents the offset within the cache line and if the element is green, the element is being read in order to show the access pattern.

- Figure 5.1a shows the entire current cache line being read for a single stream. In this case, a single stream and cache line are being read.

- Figure 5.1b shows the current cache line being read for eight different streams, each reading the next element at the respective offset per stream. In this case, eight different cache lines and therefore eight different cache lines are being read.

- Figure 5.1c shows the current and next cache line being read, crossing the cache line boundary. Crossing a boundary implies that at any given moment from any stream, two consecutive cache lines have to be read in the same cycle. In this case, a single stream from which two cache lines are being read.

- Figure 5.1d shows the worst case scenario. If four boundaries are crossed in a single cycle, four different streams and eight different cache lines must be read concurrently.



(a) Eight reads from a single stream.

(b) Eight reads from eight different streams.

(c) Crossing a cache line boundary.

(d) Crossing four cache line boundaries.

Figure 5.1: Four AFU access patterns using an eight read port buffer.

Internally increasing the cache line size will not solve the problem of crossing cache line boundaries because the new boundary can also be crossed at some point. In essence, besides requiring a large memory to buffer the proper number of cache lines per stream, an eight read port memory is required with the granularity of a single element. This problem becomes less complex if the eight read ports would be constrained in a certain way. For example, by dividing the number of streams by the number of read ports and have each read port be dedicated to a subset of the streams. However, this contradicts the desire to keep up with the high-bandwidth and low latency interface provided by OpenCAPI. For unpredictable access patterns across streams, certain read ports may have nothing to do because of such a constraint. It also limits the access pattern generalization to not exclusively support purely streaming accesses.

5.3 Design Requirements

It has become clear that a buffer architecture is needed for accelerators with streaming access patterns for which we wish to fully utilize the high-bandwidth and low latency interface of OpenCAPI. In order to do so, an access pattern generalization has been made that requires multiple read ports to concurrently and independently read from multiple independent streams of data. The merge-sort case study showed that a first-order naive design approach is not able to solve this problem and that care must be taken in order to maintain the design philosophy of high-bandwidth and low latency memory access for FPGA accelerators. Based on the previous chapters, the following is a set of requirements to which the final design has to comply in order to achieve these goals.

- Buffer 64 different streams in order to sustain the high bandwidth of OpenCAPI.
- Cover the OpenCAPI latency of 1 µs per stream.
- Go from 128 byte read port granularity to 16 byte data element granularity.
- Provide eight individual read ports such that if one read port cannot continue, the rest can still make progress.

- Handle reading across multiple cache line boundaries in a single cycle.
- Provide read ports with a low-latency between read request and data response.
- Provide a simple AFU interface by requesting data elements using stream identifiers. Streams are initially functionally reset such that an AFU has no knowledge of the addresses used.
- Provide a simple but generic interconnect interface that can be bridged to any current or future interconnect standard.
- Target the KU15P FPGA since the design can be scaled to fit the VU37P. Especially due to the vast increase in number of GTY transceivers, multiple OpenCAPI bricks can be attached.
- Confirm that the reference DLX and TLX design, provided by the OpenPOWER Foundation, also fits on the FPGA.

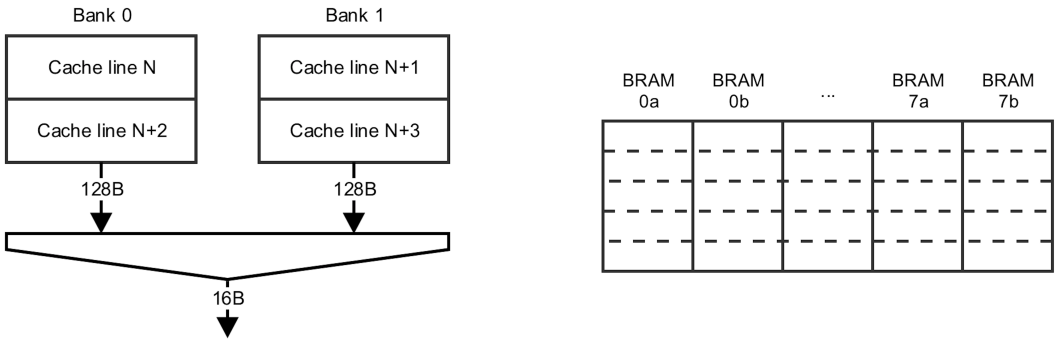
5.4 Naive Design Exploration

Section 5.2.1 showed that a traditional approach of placing a buffer between the OpenCAPI interface and the accelerator for each stream will not fit on the target FPGA. The additional problem is the possibility of reading across a cache line boundary as shown in Section 5.2.2. While the number of streams, the number of cache lines buffered per stream, or both can be reduced, the difficulty at hand is the required eight individual read ports with access to two consecutive cache lines in every stream buffer. This problem can be generalized as an eight read-port memory. Since FPGAs do not have the same level of flexibility as an ASIC has in terms of custom multi-ported memory cells, a solution has to be found regarding the available memory primitives. There are several traditional memory organizations that enable multi-port read access, built from smaller memory primitives.

- Banked Memory divides the total memory capacity into smaller memories called banks. Read requests are distributed across the banks. If multiple requests require access to the same bank, arbitration is required.
- Duplication replicates the memory contents in multiple memories and divides the read ports among them. Care has to be taken in order to keep the memories synchronized. Either by writing to all memories simultaneously or by keeping a live value table that keeps track of where valid data is located.
- Multi-pumping enables sharing of a common resource by running part of the logic at a multiple of the global frequency. For example, a memory primitive can be run at twice the frequency, effectively doubling the read ports or the data width.
- Multi-port Primitives provide multiple ports to the same memory. Since the target device is an FPGA, custom cells are not an option, but BRAMs can be configured as a true dual port primitive for example.

5.4.1 Cache Line Interleaving

For each stream buffer, both cache line N and N+1 could be read in the same cycle, as illustrated in Figure 5.1c. This can be achieved by using a banked memory organization and interleaving even and odd cache lines, as shown in Figure 5.1a. Each bank consists of half of the total number of cache lines per stream and is built from multiple BRAM primitives as shown in Figure 5.1b. BRAMs are chosen because there is not enough distributed RAM available. URAMs have a fixed configuration of 4096 entries. This is too large for a single bank and the access latency of DRAM is too high. Each BRAM is configured as a 512 entry, 8 B wide memory, of which sixteen are needed per bank. Basically each BRAM holds half of a 16 B element, where the element offset is denoted by the numbers zero through seven and each half is denoted by suffix a or b. The green box shows how a single element is divided between two BRAM primitives and each half is located at address zero. The entire cache line is spread horizontally across BRAM primitives and can be accessed by reading at the same address in all sixteen BRAMs simultaneously. Doing this in both banks yields two successive 128 B cache lines from which a single 16 B element has to be selected by means of a 256 B:16 B multiplexer for each read port. This architecture works under the assumption of a streaming access pattern because only successive elements are read. In a single cycle, only elements in two consecutive cache lines can be read. Therefore, each bank does not have to have eight physical read ports, but the requested element can be read by selection. While this architecture works in theory, there are two distinct drawbacks: multiplexer logic and BRAM primitive usage.



(a) Two cache line banks per stream buffer. (b) Memory organization of a single bank.

Figure 5.2: Interleaving even and odd cache lines.

Multiplexer Logic

In order to extract the correct 16 B element from these two cache lines, the correct cache line (2:1 MUX) and the correct element (8:1 MUX) have to be selected. Since any read port can access any stream, the correct stream has to be selected, adding an additional 64:1 multiplexer. In total, each read port requires a 1024:1 multiplexer per bit of an element, resulting in 128 (16 bytes is 128 bits) of these multiplexers per read port.
Table 5.1 summarizes several configurations of multiplexers after synthesis in the Xilinx Vivado 2017.1 tools targeting the KU15P. The column denoted by Ways is the number of selectable inputs and Width is the width in bits of each of those inputs. The estimation of the number of LUTs is based on the multiplex structures discussed in Section 4.4.3. Multiplexing of 32 or fewer inputs can be done within a single CLB. Figure 4.9 can be used to deduce the circuit for 16 or fewer inputs. For multiplexing 32 or more inputs, under the assumption that the number

of ways is a power of two, the optimal structure of eight LUTs is multiplied by the number of times it is needed plus additional LUTs for selecting between the multiple optimal structures. An additional LUT is required for every multiple of four optimal structures, since a single LUT can act as a 4:1 multiplexer.

Table 5.1: Summary of several multiplex configurations by estimation and after synthesis with the Xilinx Vivado tools targeting the KU15P.

Ways	Width	LUTs	
		Estimation	Vivado
8	1	2	3
8	64	128	192
8	128	256	384
16	1	4	5
32	1	8	9
64	1	17	18
128	1	33	34
256	1	66	68
512	1	132	137
1024	1	264	273

Table 5.1 shows that the synthesis tool always consumes more LUTs than expected. LUTs can also be used for routing of wires or for multiplexing instead of the hardwired multiplexers available. It is clear that if the number of ways increases, the estimated and observed number of LUTs start to differ more, or in other words, as the number of wires increases and therefore the routing complexity, more LUTs are used for wiring. The table also shows that when the width is increased, the amount of LUTs increases linearly with the initial observation for eight ways. From these results an improved estimation regarding the total number of LUTs required can be made.
Using the synthesis result for 1024 ways times the element width times eight read ports results in 279552 LUTs. This is roughly 53% of the total LUT resources available on the KU15P. Besides, such deep multiplex structures are required to be pipelined in order to comply with the target frequency of 200 MHz. Depending on the number of pipeline stages, an increasing number of flip-flop resources are required that also decrease the available resources for control logic and the accelerator itself. Besides that, there is also demultiplexing logic required for distributing the read requests among the stream buffers. The bottom line is that building these multiplex structures is very inefficient in terms of FPGA resource utilization.

Multiplexers can also be implemented using DSP slices. A single DSP slice can be configured as a 48 bit wide 2:1 multiplexer [64]. Slices can also be cascaded due to the internal multiplexer that selects between the two slice inputs and one cascaded input from another slice. Therefore, to implement a 1024:1 multiplexer, 512 slices are needed. Since each slice has a width of 48 bit, three slices will cover the width of a 16 B element. A single read port requires 1536 slices. Thus a total of 12288 slices are required, which is more than six times as many slices as available on the KU15P.

BRAM Primitive Usage

Besides the difficulty of selecting the correct element, due to using two banks per stream, each bank consists of half the number of cache lines. As discussed in Section 4.4, there is only a

limited number of different BRAM configurations. The 512 entry by 8 B wide configuration has the smallest number of entries. A single cache line can be distributed over sixteen BRAM primitives, but each primitive will effectively only utilize 128 out of the 512 entries or 25%, as shown in Figure 5.1b. This solution would use 2048 BRAM primitives which is roughly 208% of the available resources.
One optimization is to double pump the BRAMs such that each BRAM houses a single element, denoted by the number zero through seven in Figure 5.3. The green box shows that a single element is located within the same BRAM primitive: half at address zero and the other half at address one. A cache line is horizontally spread over the eight BRAM primitives and vertically over two indices within a primitive. Double pumping results in utilizing 50% of each BRAM and 104% of the total BRAMs available.

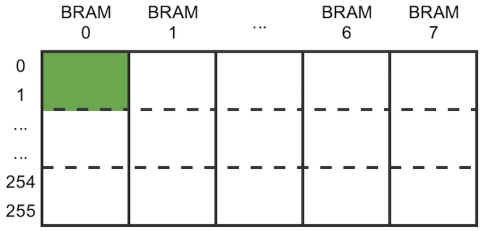


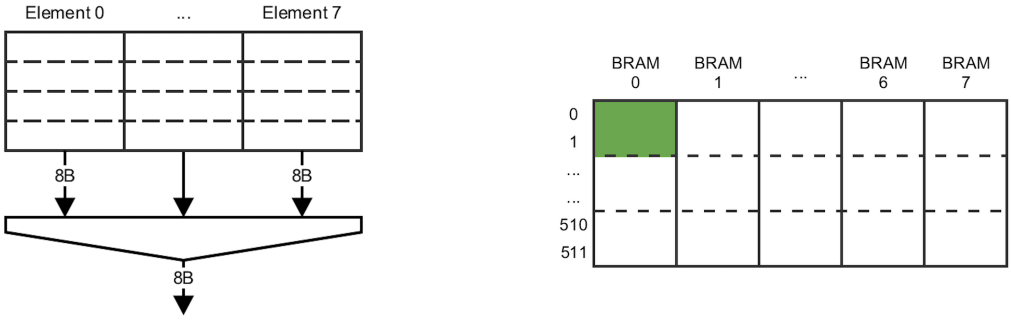
Figure 5.3: Memory organization of a single double-pumped bank.

5.4.2 Element-wise Double-pumping

By observing the streaming access pattern more closely with respect to the organization of BRAM primitives, it is clear that when a cache line boundary is crossed, each offset is read once per cycle at most, but never twice at the same offset. By exploiting this observation, a variation of the previous solution is possible which removes the need for two banks per stream buffer. Figure 5.3a shows a single stream buffer where eight horizontal BRAMs contain a cache line. Each BRAM is double pumped as shown in Figure 5.3b. The green box shows that a single element is located within the same BRAM primitive, similarly as in the previous solution.
Since two banks are no longer required, all cache lines fit within the same primitive resulting in a utilization of 100% per BRAM primitive. For eight read ports roughly 52% of the BRAM primitives are used. Also, there is no longer need for the first level of multiplexing, that reduces the per read port multiplexing to 512:1. Despite these improvements, this solution still requires 70144 LUTs utilize roughly 13% of the resources.
While the design would fit theoretically, each BRAM primitive will have a fan-out equal to the number of read ports. Since getting data out of a BRAM and into processing logic is complex at the target frequency of 200 MHz, the additional fan-out problem does not make this easier. Besides that, using more than half of the available BRAM primitives means that multiple large columns of BRAM primitives are used, scattered all over the FPGA as shown in Section 4.4.1. This complicates a unified access latency for each data entry (also keep the fan-out in mind) due to wiring delays across the entire FPGA. Pipelining helps with timing closure, but also increases the read latency of the buffer, a critical metric in the design.

5.4.3 Cache Line Duplication

Duplication guarantees that each read port has access to every element in every cache line at any time. Figure 5.5 shows a possible architecture for cache line duplication. Each read port has a copy of every cache line in each stream buffer, removing the need for large selection logic.



(a) Element selection from eight BRAM primitives.

(b) Memory organization of a single stream buffer.

Figure 5.4: Element-wise double-pumping.

The drawback is that the required memory primitives scale linearly with the number of read ports. As shown in Section 5.2.1, the number of supported streams has been decreased to 64 in order to save memory primitives and have resources to spare for control logic and the AFU. A single copy of all buffered cache lines for all streams requires roughly 2 MB or 52% of BRAM resources. This becomes 16 MB or 416% for eight read ports which is four times as much as available on the target FPGA. Each read port also has to select the right element. A 512:1 multiplexer is required and if the BRAMs are double-pumped, the total LUT consumption is roughly 13%. It is obvious that this solution will not fit the target FPGA either. However, the benefit is that the wiring per read port stays within this slice that should make wire routing easier. Also the fan-out per BRAM output has decreased from eight to one, in comparison with the element-wise double-pumping proposal in Section 5.4.2.

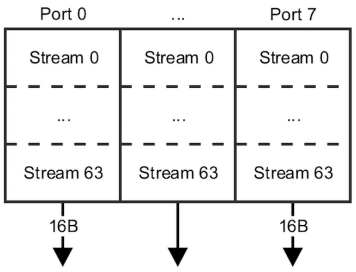


Figure 5.5: Stream buffer duplication for each read port.

5.4.4 True Dual Port BRAM

Finally, memory primitives can be used with multiple ports. A BRAM can be configured as either a simple dual port or a true dual port, as mentioned in Section 4.4. In the case of a simple dual port, one port is a designated read port and the other a designated write port. The ports of a true dual port can be configured on-the-fly to act as either a read or write port. Due to the read-write imbalance of writing a cache line in one cycle and completely reading it for eight cycles, the flexibility of a TDP memory seems appealing. However, the effective capacity of a BRAM in TDP mode is reduced to 18 kbit. Therefore, not only are twice the number of primitives required to obtain the same effective capacity, but also complex control logic to configure the ports on-the-fly is required. Even worse is that the previously used configuration

of 512 entries is not available in TDP mode. Instead, the smallest number of entries is 1024 with a data width of 36 bit. This means that any of the previously discussed proposals will require twice as much BRAM primitives,not to speak about the additional control logic.

5.4.5 Summary of Naive and Traditional Designs

Traditional solutions for multi-ported memories have been shown above to not satisfy the desired requirements. Interleaving is an effective way to increase the number of read ports without increasing the required memory primitives. It does, however, pose the problem of under-utilized BRAM primitives and requires large multiplex structures that do not fit in the FPGA resource budget, not to speak of the wiring nightmare that will occur.
Element-wise double-pumping is a promising architecture. However, the fan-out of eight for every BRAM and the large columns of BRAM primitives scattered around the FPGA will require additional pipeline registers. This increases a critical metric: the read latency of the AFU.
While duplication solves the problem of the large multiplexing structures and the fan-out, there is no resource budget for it. Double-pumping BRAM primitives improves utilization and decreases the required LUTs, but does not solve the underlying problem.
The bottom line is that duplication is needed to mitigate the fan-out and, therefore, the wiring problem, but not for all cache lines.

5.5 Proposed Architecture

In essence there is a conflict between storing enough cache lines to hide the latency of OpenCAPI, and providing eight individual read ports to the AFU. Simultaneously, on the one hand wiring delay must be taken into account, and on the other hand not all of the available memory resources can be used. Therefore, the proposed multi-stream buffer architecture is split into two levels by exploiting different memory primitives. Figure 5.6 shows both levels, L1 and L2, analogous to cache naming conventions.

- L1 is a 'small' buffer that is fed by L2. It stores a subset of consecutive cache lines per stream which are duplicated for each read port. This way, each read port has access to exactly the same data and there are no large multiplexer structures required.

- L2 is a 'big' buffer that gets up to one new cache line every cycle through OpenCAPI. It is targeted to cover the latency of OpenCAPI by using large memory primitives and therefore does not consume all memory resources as was previously the case.

L1 is optimized for low-latency and multiple read port access while L2 is optimized for memory capacity to cover the latency of OpenCAPI. By choosing the appropriate memory primitive for each level, an architecture can be devised that complies with the requirements and does not consume all available resources while doing so. To keep all duplicate cache lines synchronized in L1, L2 writes new data to all BRAMs simultaneously.

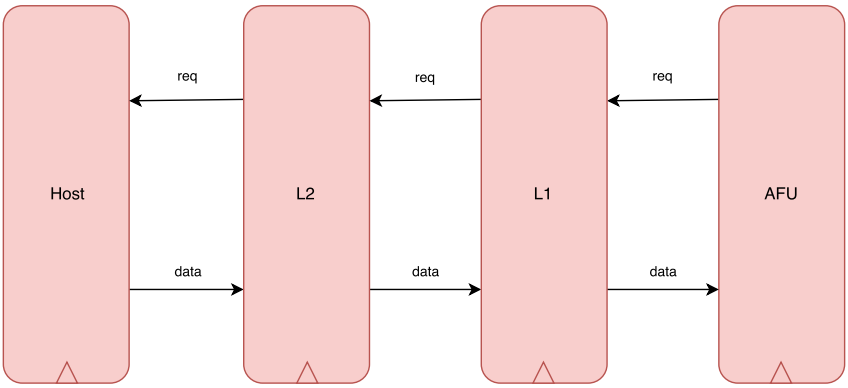


Figure 5.6: Proposed stream buffer architecture with two levels of buffering.

The proposed architecture mitigates the fan-out problem from the critical AFU read path to a signal path outside of this domain, namely, between the L2 and L1 buffers. This gives us more time to move data out of the URAMs and into all BRAMs simultaneously. However, since cache line granularity data is moved from L2 to L1, the number of fan-out signals increases. Pipeline stages can be added to overcome this if necessary, due to the fact that memories are located in specific columns as shown in Section 4.4.1. The smaller BRAM arrays in this proposed architecture, as compared to the element-wise double-pumped architecture, make it easier to obtain unified access latency for each data entry. This is also achieved by supplying a BRAM array per read port such that cache lines used in the near future can be located closest to their consumer.
Since L1 is basically a smaller version of the proposed data duplication architecture, the same amount of LUT resources of roughly 13% is required. Depending on the chosen configuration as to the number of streams and the number of cache lines, resource requirements differ. Section 7.2 describes the design choices made in more detail.

Chapter 6

Design Methodology

This chapter briefly introduces a cell-based design methodology with an inherent ready-valid protocol. The methodology consists of a workflow and cell-based design library. To motivate the ease-of-use and applicability of the methodology, several complex examples are shown.

6.1 Design Philosophy

The methodology described is designed over a course of several years by Andrew K. Martin from the IBM Austin Research Lab. The initial ideas concerning this methodology originate from a paper published in 2001 called The Theory of Latency-Insensitive Design [65]. It describes the fundamentals of a correct-by-construction design methodology. This paper forms the basis of the ready-valid design methodology by Andrew K. Martin [66].

6.1.1 The Theory of Latency-Insensitive Design

The paper [65] presents a foundation of a correct-by-construction methodology which separates a system based on communication and computation. A system is defined as a set of synchronous computational processes that exchange data with one another over communication channels. An abstract protocol is used for communication whose main characteristic is to be insensitive to the latencies of the channels. The protocol works under the assumption that the computational processes are stallable and guarantees that computational processes behave correctly independently of the channel latencies. This enables changing the latency of a communication channel without affecting the functionality of the system, which is a useful property for hardware design. Suppose a system is implemented, but due to wire delay timing constraints are not met. This theory guarantees that a relay station, or register, can be inserted in the communication channel without affecting the functionality. Therefore no costly redesigns are needed.
The idea of relay stations is borrowed from the pipelining concept and a relay station in function is similar to a register. This approach relaxes timing constraints during the early design phases when accurate measures of delay paths between computational processes are not yet available. If after physical implementation a mismatch exists between the timing constraint and the communication channel delays, they can easily be corrected by inserting the correct amount of relay stations. Since every computational process operates according to the latency-insensitive protocol, no changes are required in order to reflect the necessary changes in the communication channel latencies.

6.1.2 Ready-Valid Design Methodology

Typically, there are three types of flow control concepts a hardware designer can choose from.

- Credit-Based protocol manages credits between a start and end module. If no more credits are available, the logic in between is stalled.
- Cycle-Based protocol indicates after how many cycles a response will be received for sent data for example. A downside is that it is difficult to decide on what to do when no response is received, or more generally, what to do when the protocol breaks.
- Ready-Valid-Based protocol has a valid signal to indicate the validity of an output and a ready signal to indicate it is ready to receive a new input. Usually it is only used between large modules in a system.

The ready-valid design methodology discussed here differs from the typical ready-valid protocol since it uses the paradigm all the way down to the lowest level of a system. This combined with the latency-insensitive design theory results in a methodology that consists of a library of cells that are analogous to the computational processes. These cells are interconnected using a ready-valid protocol, the communication channel, which makes it possible to stall at a cell granularity. This means that progress is made whenever possible.
The cell-based nature allows to easily understand a design by following the ready-valid protocol and associated data, if present. Due to a rich library of cells, writing a design mostly consists of connecting the cells and compiling the design. Since cells are correct-by-construction, errors only consist of typos, signal width mismatches, and other easily fixable errors. After that the ready-valid protocol is functionally correct. Only the associated data signals could not be functionally correct. This means that transformations on the data signal have been done incorrectly.

6.1.3 Ready-Valid Communication Protocol

In general, a cell within the ready-valid design methodology has a set of configuration parameters and a set of input and output signals, as shown in Figure 6.1. Examples of configuration parameters are the number of inputs or the signal width in bits of each data input. The input and output signals are named according to a predetermined scheme. The signals on the left of the generic cell are all called input signals, even though physically the ready signal is an output. Together they form the input interface and follow the naming convention of `i_v` and `i_r` for the valid and ready signal, respectively. The data signal, typically denoted by `i_d`, is not obligatory and some cells only have ready and valid signals. According to the same naming convention, the output signals are called `o_v`, `o_d` and `o_r` and together form the output interface.

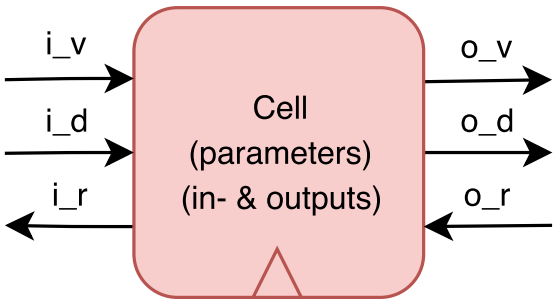


Figure 6.1: Input and output signals of a generic cell.

The ready-valid protocol used by the cells on the communication channels between computational processes is a handshaking protocol. Progress is only made if both the input valid and ready signal are asserted for a cell in a given clock cycle. The communication channel is explicit and building a system mostly consists of connecting cells by a communication channel consisting of a ready and valid signal. In a system consisting of multiple cells, the naming convention dictates that a preceding cell is called upstream and the successive cell is called downstream.
In the ready-valid design methodology, a cell or computational processes can be combinatorial or sequential logic. Examples of each type will be shown in Section 6.3. Independently of the type, a cell has at least two ready-valid signal pairs. One as input and one as output interface. In terms of physical input and output signals, a cell has two inputs, `i_v` and `o_r`, and two outputs, `i_r` and `o_v`.
Typically, when a cell comes out of a reset state, its input ready signal is asserted independently of any downstream cell ready signals it may receive at its output interface. The reason is that progress can be made by supplying the cell with a valid input to be serviced. The output valid signal is asserted when a valid input occurs and the cell is ready to service this valid input. In other words, the cell is not in a stalled state. If the cell is in a stalled state, the input ready signal will be de-asserted to indicate the upstream cells that a stall occurred. The concept of indicating to upstream cells that a cell is stalled is called applying back-pressure in the ready-valid design methodology. An example of applying back-pressure could be that a cell requires data from multiple locations that do not arrive at the same time. Therefore, the cell has to be stalled until all data is available and back-pressure has to be applied to upstream cells. It is important to note that the input ready signal is only de-asserted if the cell is stalled, or if a downstream cell has stalled that induces a chain reaction of stalls. This is different from a typical ready-valid protocol where the ready signal is de-asserted after a transaction occurred (an example is AXI [56]).

6.1.4 Differences Compared to Asynchronous Design

At first sight, the ready-valid design methodology may be interpreted as an asynchronous design methodology. However, there are several differences between the two. First of all, asynchronous design does not use a clock, while this methodology does. That means that the presentation of a ready or valid signal would be by changing the value of a signal, rather than by asserting it.
The ready-valid design methodology allows a ready or valid signal to be asserted in one cycle, and then de-assert in a subsequent cycle, even if the complimentary signal (valid or ready, respectively) was never asserted and no transaction took place. Similarly, the data corresponding with a valid signal is allowed to change on subsequent cycles, even though no corresponding ready signal was received and hence no data transfer took place. This would not work in an asynchronous framework and may have implications on design.
Finally, the `base_aburp` and `base_alatch` cells (discussed in Section 6.4.3) loose their meaning in an asynchronous framework. Although something analogous may be needed to maintain reasonable timing constraints between valid and data signals.

6.2 Workflow

Besides a supplied cell library, the ready-valid design methodology also comes with a workflow of several steps and special cells to speed up the workflow.

1. Build the System The first step is build the system with the provided cells, possibly accompanied by other logic. Relay stations should be placed between combinatorial cells and

with experience proper locations are easily identified. These relay stations are analogous to configurable registers.

2. Functional Verification After compiling the design from Step 1 and fixing compiler errors and warnings, the ready and valid signals should be verified first. Make sure they are always defined and only then verify the associated data signals. If more in-depth debugging is required, trace the valid and ready signals throughout the system and make sure cells apply back-pressure when needed, in order to not lose valid information.

3. Synthesis and Timing Constraints After functional verification, the system should be synthesised to see if it meets the timing constraints. If the constraints are met, the next step can be started. If the constraints are not met, the register cells instantiated in Step 1 should simply be reconfigured accordingly or inserted multiple times to allow for multi-cycle communication channels; this step should be restarted. This process continues until the timing constraints are met everywhere in the design.

4. Physical Verification After meeting the timing constraints, the system has to be verified again. When this step finishes successfully, it is possible to go back to Step 3 and tweak the design in order to improve metrics such as area or operating frequency.

6.2.1 Tweaking Relay Stations

The positioning of registers or relay stations depends on whether the system will be implemented on an ASIC or FPGA. For an FPGA, distribution of the wires is more difficult than driving them. FPGAs have buffers and repeaters everywhere to simplify this for the designer.
An ASIC does not have this luxury, thus the instinct of the designer concerning relay station positioning should be different. For ASICs, driving strength is more difficult. Positioning relay stations should take the combinatorial path of the function that is being implemented into account.

6.2.2 Synthesis Helper Cells

Systems or modules within a system typically have a large number of input and output signals, especially when data widths are large. When implementing a design on an FPGA, the provided tools try to connect each input or output signal of the system or module to a physical pin on the FPGA package. Depending on the size of the system or module and target device, there might not be enough physical pins for the tool to finish implementation. However, obtaining an estimate on metrics such as area and operating frequency is often desired, also for modules within the system.
In order to speedup synthesis and overcome the described problem, two helper cells are present in the cell library called `base_input_lat` and `base_output_lat`. These modules are parametrised shift registers. The `base_input_lat` cell is to be attached to one physical pin on the FPGA package and to all input signals of the system or module to be synthesised. Similarly but in opposite direction, the `base_output_lat` cell is used to connect multiple output signals to a single physical pin.

6.3 Delay-Insensitive Cell Library

The cell library that is part of the ready-valid design methodology has been published on GitHub [67] and is a work in progress. At the moment of writing, the version published on November 16, 2017 is used throughout this document.

6.3.1 Diagram Legend and Naming Conventions

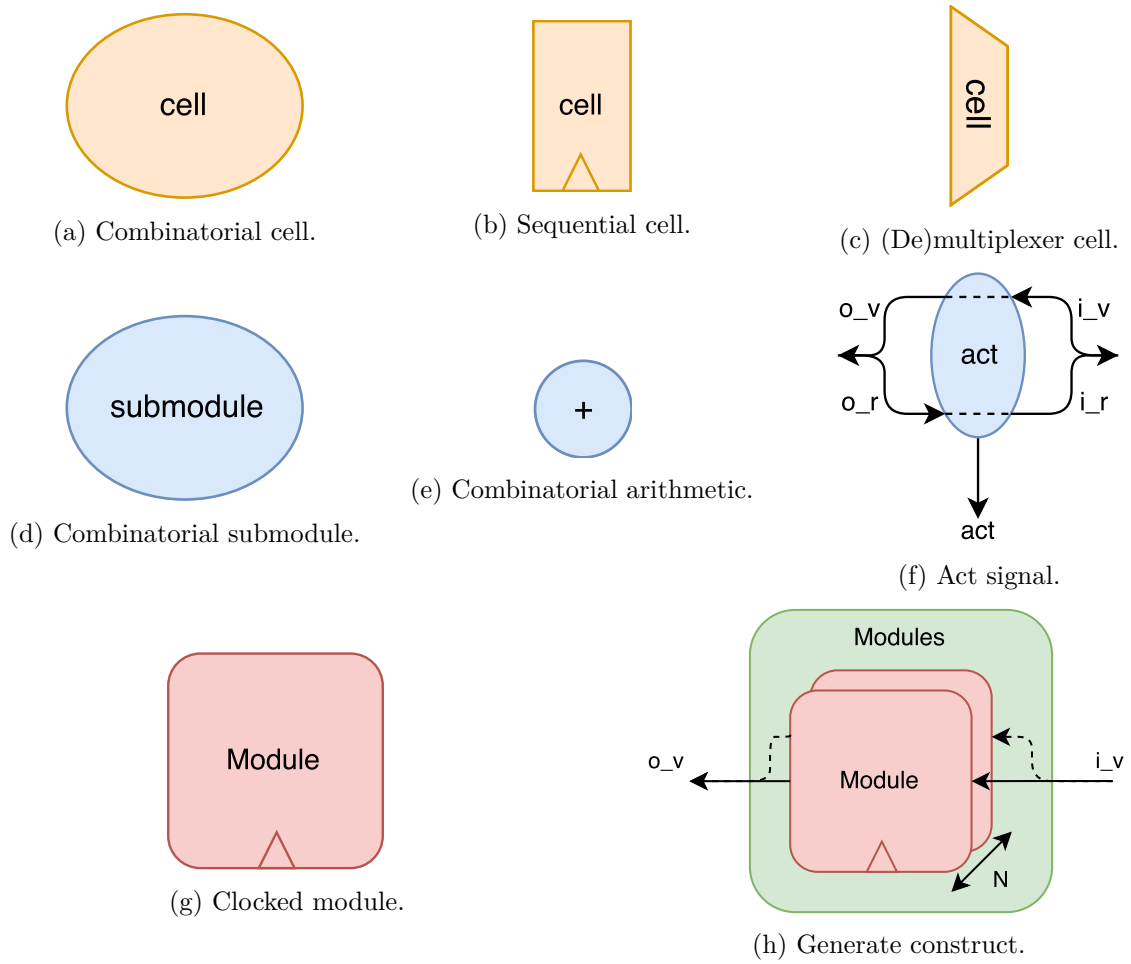
Throughout this thesis, implementation diagrams and signal names follow a predefined scheme consisting of object shapes and colors and naming conventions to indicate various characteristics.

Diagram Legend

A cell is a submodule that is part of the design library. Figure 6.2a shows a combinatorial cell and Figure 6.2b a sequential cell. Figure 6.2c shows a (de)multiplexer cell, depending on the direction of the input and output signals.
Besides cells from the library, also logic that is not a cell can be included in a design called a submodule, shown in Figure 6.2d. Figure 6.2e shows a combinatorial arithmetic operation. A typical example is an addition to increment a counter. That could be implemented using a sequential cell. A common construct in this methodology is shown in Figure 6.2f and is called an act signal, often denoted by `*_act`. It is a logical AND operation on a valid and ready signal, for example used as an enable signal.
To allow for hierarchy in diagrams, a collection of multiple cells and submodules can be replaced by a module. The module name always starts with a capital and is shown in Figure 6.2g. To show multiple identical instantiations of a cell or module, a generate construct is indicated by a green box. Multiple objects stacked behind each other and the number of instantiations is shown in the lower right corner, as shown in Figure 6.2h. The input and output signals are solid lines for the first instantiated object and dashed for any other objects stacked behind it.
Typically, only the downstream signals are shown in the diagrams and only rarely are upstream signals shown, such as ready signals. An act construct typically shows the ready signal until it is terminated.

Naming Conventions

Typically the input and output signals of any cell or module start with `i_*` and `o_*`, respectively. Internal signals of a cell or module also follow a naming convention, depending on the location of the signal with respect to the sequential elements and whether the signal has a special purpose, such as valid, ready, data, or act. Often the cell or module the valid signal originates from is included in the signal name. An example could be the output valid signal of the initial register cell: `s1_reg_v`. Another example could be a data signal after a counter cell operating after four pipeline stages from the input signals: `s4_cntr_d`. Listings shown later on provide more naming convention examples.
Cell or module instantiations follow a similar naming convention, indicating the pipeline stage the signal operates in, the cell or module name and indication if it is a combinatorial or sequential element. Typically combinatorial elements have no suffix while sequential elements have a `*_reg` or `*_lat` suffix, standing for register and latch, respectively. In the design methodology, a register is a sequential cell with a ready-valid pair as input and output. A latch is a sequential cell without ready-valid pairs.
Most cells have a prefix, either `a*` or `e*`. For historic reasons, the `a*` stands for asynchronous and is used to mean that the cell has a ready-valid interface. The `e*` prefix means encoded and typically refers to the select signal. Cells without this prefix typically have a decoded select signal.
Most of the cells are implemented as big-endian, but for certain cells the endianness matters. For example for multiplexers. For those cells a little-endian variant exists and is denoted by the suffix `*_le`.



(a) Combinatorial cell.

(b) Sequential cell.

(c) (De)multiplexer cell.

(d) Combinatorial submodule.

(e) Combinatorial arithmetic.

(f) Act signal.

(g) Clocked module.

(h) Generate construct.

Figure 6.2: Implementation diagram conventions.

6.3.2 Pass Gate Cell

A basic cell is the pass gate `agate` shown in Listing 6.1. Due to its straightforward behavior, it acts as an initial example of a cell from the library. The function of the cell is to grant access to a downstream ready-valid cell if a certain condition is met. The condition is supplied to the cell as the one bit input enable signal `en`. The cell has two additional input signals that are the input valid signal `i_v` and output ready signal `o_r`. As the listing shows, the two output signals `i_r` and `o_v` are nothing more than a logical AND operation on one of the input signals and the input enable signal. The `width` parameter is used to configure the number of ready-valid pairs this cell handles.
An application example of the pass gate cell is to grant read access to a memory, but only if the memory contains valid data. In this example, the input valid signal could indicate a valid read request, accompanied by a memory address as data signal. The enable signal could originate from a counter cell that keeps track of the number of valid data entries.

```
1 module base_agate #
2 (
3   parameter width = 1
4 )
5 (
6   input  [width-1:0] i_v,
```

```
7   output [width-1:0] i_r,
8   input  [width-1:0] en,
9   output [width-1:0] o_v,
10  input  [width-1:0] o_r
11 );
12
13  assign i_r = en & o_r;
14  assign o_v = en & i_v;
15 endmodule
```

Listing 6.1: Pass gate cell from the ready-valid cell library.

6.3.3 Decode Cell

The library also consists of non-ready-valid capable cells such as the decode cell `decode` shown in Listing 6.2. The function of this cell is to decode the input signal `din` if a certain condition is met. Similar to the `agate` cell, the input enable signal is called `en`. Due to the parametrized nature of the cell library, each cell has a variable width. In this case the input data width is configured using the `enc_width` parameter that implies the output width as well. In order to support the parametrization, a Verilog generate statement is used in combination with a for-loop. During compilation, the for-loop is unrolled followed by generation of the associated hardware.

```
1  module base_decode #
2  (
3    parameter enc_width = 1,
4    parameter dec_width = 2 ** enc_width
5  )
6  (
7    input                   en,
8    input  [enc_width-1:0] din,
9    output [dec_width-1:0] dout
10 );
11
12   genvar i;
13   generate
14     for(i=0; i<dec_width; i=i+1) begin : Gen
15       assign dout[i] = en & (din == i);
16     end
17   endgenerate
18 endmodule
```

Listing 6.2: Decode cell from the ready-valid cell library.

6.3.4 Multiplexer Cell

Another basic non-ready-valid capable cell is the multiplexer cell `emux` shown in Listing 6.3. In line with the cell library philosophy, a `width` parameter is available. Another recurring configuration parameter in the ready-valid cell library is the `ways` parameter. This parameter typically indicates the number of inputs of a cell, which in this case is the number of inputs to select from. This parameter implies the encoded signal width of the input select signal `sel` that uses the built-in `$clog2()` function of Verilog. This function returns the logarithm in base two of the input argument to the function.

```
module base_emux #
(
  parameter width = 1,
  parameter ways = 2,
  parameter sel_width = $clog2(ways)
)
(
  input  [(width*ways)-1:0] din,
  input  [sel_width-1:0]    sel,
  output [width-1:0]        dout
);

  wire [width-1:0] din_array [ways-1:0];

  genvar i;
  generate
    for(i=0; i<ways; i=i+1) begin : Gen
      assign din_array[i] = din[(i+1)*width-1:i*width];
    end
  endgenerate

  assign dout = din_array[sel];

endmodule
```

Listing 6.3: Multiplexer cell from the ready-valid cell library.

6.3.5 Ready-Valid Merge Cell

The reason for showing the previous two cells, besides introducing typical signal and parameter names, is to illustrate how a ready-valid cell can be constructed from non-ready-valid capable cells. A ready-valid merge cell, shown in Listing 6.4, merges `ways` inputs, each with `width` wide data. The input select signal `sel` is first decoded using the decode cell discussed in Section 6.3.3, with the enable signal always asserted. The output valid signal is determined by whether the particular input is valid and which input was selected. Since both signals are `ways` bits wide, a logical reduction OR operator is used. Similarly, but opposite, the input ready signal is determined by which input is selected, since that input is ready to receive a new valid input data, and the output ready signal to determine if the downstream module is ready. Since there are `ways` input signals and the output ready signal is only one bit, it is replicated as often as there are inputs. Finally the input data `i_d` is selected using the multiplexer shown in Section 6.3.4.

```
module base_aemux #
(
  parameter ways = 2,
  parameter width = 1,
  parameter sel_width = $clog2(ways)
)
(
  input  [ways-1:0]         i_v,
  output [ways-1:0]         i_r,
  input  [(width*ways)-1:0] i_d,
  input  [sel_width-1:0]    sel,
  output                    o_v,
```

```
13    input                       o_r,
14    output [width-1:0]          o_d
15  );
16
17    wire [ways-1:0] sel_dec;
18    base_decode_le#(.enc_width(sel_width),.dec_width(ways))
19      isel_dec(.din(sel),.dout(sel_dec),.en(1'b1));
20
21    assign o_v = |(sel_dec & i_v);
22    assign i_r = sel_dec & {ways{o_r}};
23
24    base_emux_le#(.ways(ways),.width(width))
25      imux(.sel(sel),.din(i_d),.dout(o_d));
26
27  endmodule
```

Listing 6.4: Ready-valid merge cell from the ready-valid cell library.

6.4 Advanced Examples

The previous section introduced the cell library that is part of the ready-valid methodology. This section uses the presented conventions to introduce several advanced examples of the methodology.

6.4.1 Interfacing with a Credit-Based Interface

A common interface type is based on credits, which is for example used by interconnects such as PCI Express and OpenCAPI to control the flow of packets. Typically it is used to share a resource and give each consumer a credit if a credit is available. When no more credits are available, consumers are no longer granted access until a credit becomes available again. The interaction with modules that are not based on the ready-valid protocol such as memory primitives may serve as an example. Memory primitives typically consist of a read and write interface of three signals: enable, address, and data. More details concerning memory primitive interaction will be discussed in Section 7.4.1. Another use case is when the ready-valid cells and modules consume too much area in the design. A local transition to a credit interface (and back) possibly overcomes this problem.
The cell library provides two cells to transition from a ready-valid protocol to a credit-based protocol (source) and vice versa (sink). The cells are named `credit_src` and `credit_snk`, respectively. Figure 6.3 shows a generalized setup and interaction with ready-valid-based cells.

6.4.2 Synchronization of Multiple Control Flows

When designs get more complex, often multiple control flows have to be synchronized with each other before progress can be made. The cell library provides a single cell to synchronize N-to-M inputs and outputs named `combine`. Two examples are shown below of uses of this cell during the implementation of the multi-stream buffer.

Many-to-One

The first encounter involves synchronization of multiple inputs with one output, as shown in Figure 6.4. In this example, the downstream module Request Consumer is shared by two ready-valid-based modules, shown on the left of the Combine cell. The Request Producer produces a

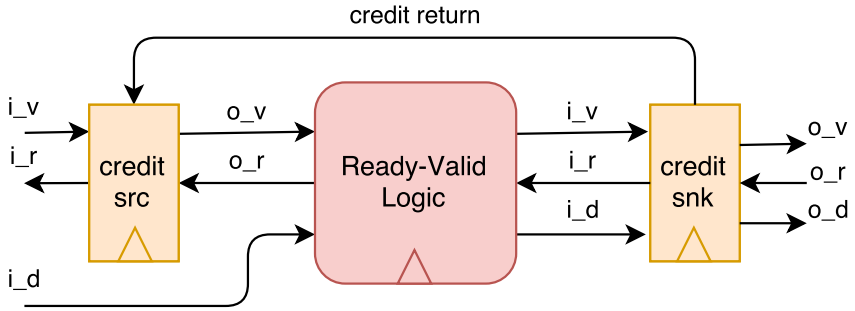


Figure 6.3: Diagram of a generalized transition between a credit- and ready-valid-based protocol.

request consisting of a ready-valid pair and a stream identifier as associated data signal. Since a resource is shared downstream, that is only able to accept a predefined number of outstanding requests, each request from the producer has to obtain a unique tag.
There are various scenarios possible in this case. If the downstream module is not ready, both the Request Producer and the Resource Manager have to stall in order to prevent discarding a valid request or waste a precious tag, or both. Similarly, if the Request Producer has no valid request, the Resource Manager should not waste a tag and, vice versa, the Request Producer should not discard a valid request when all tags are in use. The bottom line is that these two control flows have to synchronized.

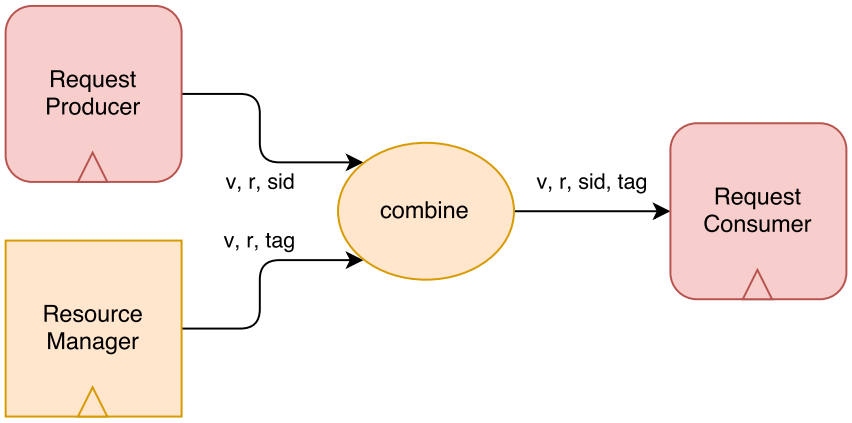


Figure 6.4: Diagram of a many-to-one synchronization example.

To do so, the cell library provides a cell that can be configured to accommodate any number of inputs and outputs, independently of each other. Listing 6.5 shows the ease of use of the `combine` cell (line 26), by simply connecting the various cells and modules together and configuring the number of inputs and outputs to synchronize. In this example, a hypothetical Request Producer module is used and a simplified Resource Manager cell from the library. The `res_mgr` cell provides a configurable number of unique tags that can be used to associate with a request. The tag return interface has been omitted for simplicity.

```
1   wire s1_req_v, s1_req_r;
2   wire [nstrms_width-1:0] s1_req_sid;
3   request_producer # (
4     .width  (nstrms_width)
```

```
5       ) is0_req_prod (
6       .clk    (clk),
7       .reset  (reset),
8       .o_v    (s1_req_v),
9       .o_r    (s1_req_r),
10      .o_d    (s1_req_sid)
11    );
12
13    wire s1_mgr_v, s1_mgr_r;
14    wire [tag_width-1:0] s1_mgr_tag;
15    base_res_mgr # (
16      .width  (tag_width)
17      ) is1_res_mgr (
18      .clk    (clk),
19      .reset  (reset),
20      .o_v    (s1_mgr_v),
21      .o_r    (s1_mgr_r),
22      .o_d    (s1_mgr_tag)
23    );
24
25    wire s1_comb_v, s1_comb_r;
26    base_acombine # (
27      .ni     (2),
28      .no     (1)
29      ) is1_cmb (
30      .i_v    ({s1_req_v, s1_mgr_v}),
31      .i_r    ({s1_req_r, s1_mgr_r}),
32      .o_v    (s1_comb_v),
33      .o_r    (s1_comb_r)
34    );
35
36    request_consumer # (
37      .width  (nstrms_width),
38      .tag    (tag_width)
39      ) is2_req_cons (
40      .clk    (clk),
41      .reset  (reset),
42      .i_v    (s1_comb_v),
43      .i_r    (s1_comb_r),
44      .i_sid  (s1_req_sid),
45      .i_tag  (s1_mgr_tag)
46    );
```

Listing 6.5: Many-to-One synchronization example.

One-to-Many

Another encounter involves synchronization of a single input with multiple outputs, as shown in Figure 6.5. In this example, an incoming read request from the AFU starts two separate processes. The first process calculates an address to index a memory based on a global pointer stored in one of the Stream Pointer modules and the number of other read ports accessing the same stream. The second process updates the global pointer based on all streams requested and will request new data upstream if needed. More details concerning the operation of the modules used can be found in Section 7.3.1 and 7.3.2.

Similarly to the many-to-one synchronization example shown above, the `combine` cell is configured with the appropriate number of inputs and outputs and connects the incoming read request to both process modules. In essence, it would be similar to Listing 6.5, traversed in reverse order. While the two presented examples have either one input or output, the `combine` cell is able to synchronize any number of inputs with any number of outputs.

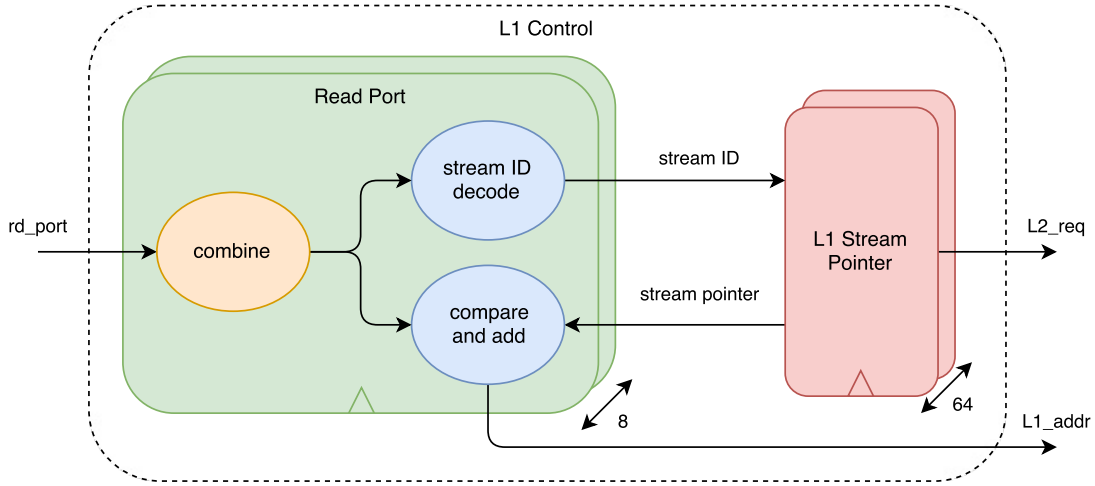


Figure 6.5: Diagram of a one-to-many synchronization example.

6.4.3 Timing Closure Using Relay Stations

The function of a relay station, as defined in the theory of latency-insensitive design [65], is fulfilled by the `reg` cell in the ready-valid design methodology. Typically the `reg` cell is used as a register for a ready-valid signal pair and the associated data signal. However, due to its parametrized nature, the cell can be easily reconfigured to meet timing by inserting an empty cycle, or to improve resource utilization by removing a cycle, either from the ready or valid path or both.
Listing 6.6 shows the instantiation of the cell. It contains the typical ready-valid input and output signal pairs and a parameter `width` to configure the width of the data signal. Unique to this cell is the `lbl` parameter that stands for latch-burp-latch. When each bit is asserted, the respective cell (`latch` or `burp`) is generated.

```
base_areg # (
  .lbl    (3'b110),
  .width  (width)
) is0_reg (
  .clk    (clk),
  .reset  (reset),
  .i_v    (i_v),
  .i_r    (i_r),
  .i_d    (i_d),
  .o_v    (o_v),
  .o_r    (o_r),
  .o_d    (o_d)
);
```

Listing 6.6: Ready-valid register cell from the ready-valid cell library.

Latch Cell

Listing 6.7 shows the `latch` cell from the ready-valid cell library. This cell is used within the `reg` cell and provides a sequential element in the valid signal path. The `o_v` signal is the output of a `vlat` cell of which the input signal is asserted only when the `latch` cell receives a valid input or when the downstream cell does not accept the current transfer. Similarly, when an associated data signal is present, an additional `vlat_en` cell is generated and is only enabled if the downstream cell accepted the current transfer. This enable signal is also used to drive the `i_r` signal.

```
module base_alatch #
(
  parameter width = 1
)
(
  input             clk,
  input             reset,
  input             i_v,
  input  [0:width-1] i_d,
  output            i_r,
  output            o_v,
  output [0:width-1] o_d,
  input             o_r
);

  wire o_v_in = i_v | (o_v & ~o_r);
  wire enable = o_r | ~o_v;
  assign i_r = o_r | ~o_v;
  base_vlat#(.width(1)) ivlat (.clk(clk), .reset(reset), .din(o_v_in), .q(o_v));

  wire [0:width-1] din = i_d[0:width-1];
  generate
    if (width > 0)
      base_vlat_en#(.width(width)) idlat (.clk(clk), .reset(1'b0),
        .enable(enable), .din(i_d), .q(o_d));
  endgenerate

endmodule
```

Listing 6.7: Latch cell from the ready-valid cell library.

Burp Cell

Listing 6.8 shows the `burp` cell from the ready-valid cell library. This cell is used within the `reg` cell and provides a sequential element in the ready signal path. The `o_v` signal propagates without any latency from the `i_v` signal, unless the previous transfer was not completed, indicated by the `burp_v` signal. Note that when `burp_v` is asserted, the input data signal is not captured in the `vlat_en` cell and the previously captured data signal is presented at the output. If not, the current input data is captured and presented and the output. The input ready signal is then equal to the inverse of the `burp_v` signal.

```
module base_aburp #
(
  parameter width = 1
)
```

```
(
  input             clk,
  input             reset,
  input             i_v,
  input  [0:width-1] i_d,
  output            i_r,
  output            o_v,
  output [0:width-1] o_d,
  input             o_r
);

  wire burp_v;
  wire burp_v_in = ~o_r & (burp_v | i_v);
  assign i_r = ~burp_v;
  assign o_v = i_v | burp_v;
  base_vlat#(.width(1)) ivlat (.clk(clk), .reset(reset), .din(burp_v_in), .q(burp_v));

  generate
    if (width > 0) begin
      wire [0:width-1] burp_d;
      assign o_d = burp_v ? burp_d : i_d;
      base_vlat_en#(.width(width)) idlat (.clk(clk), .reset(1'b0),
        .enable(~burp_v), .din(i_d), .q(burp_d));
    end
  endgenerate
endmodule
```

Listing 6.8: Burp cell from the ready-valid cell library.

Register Cell

Figure 6.6 shows a simplified view of the `reg` cell. Internally, the `burp` and `latch` cells, discussed in the previous paragraphs, are used. The orange ovals represent combinatorial logic, as shown in Listing 6.7 and Listing 6.8.

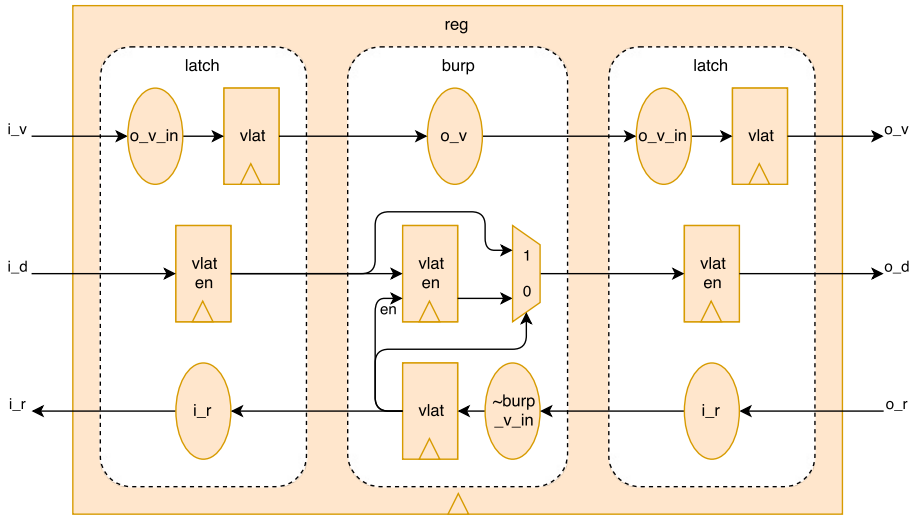


Figure 6.6: Diagram of the relay station cell with `lbl = 3'b111`.

A typical starting point is to configure the cell as `3'b110` that generates one latch and one burp. This configuration is shown in Figure 6.7 and results in a register in both the ready and valid paths. As mentioned in Section 6.2, it is good practice to insert a `reg` cell regularly during the implementation to minimize rewriting of the system description at a later stage. By simply reconfiguring the relay station after the design failed timing, for example, a new compilation can be run immediately without rewriting any code.

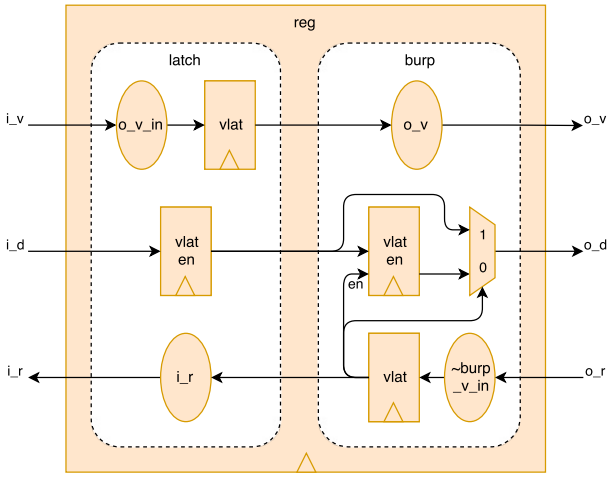


Figure 6.7: Diagram of the relay station cell with `lbl = 3'b110`.

Chapter 7

Implementation

Section 5.5 made clear that duplication is needed in order to unify access latency for each data entry. Due to exploiting different memory primitives, the proposed architecture consumes less valuable memory resources compared to traditional architectures. By providing an element-sized access granularity and select elements within or close to the memory primitive, the selecting structures are also significantly smaller compared to traditional architectures.
This chapter describes the operation of the proposed design, followed by a section which discusses the design choices made based on resource limitations imposed by the target FPGA. Afterwards the implementation of the proposed architecture is explained in more detail by going through the entire design module by module.

7.1 Functional Operation

Each of the two levels of buffering have different requirements regarding the respective memory primitive. L1 should be optimized for low latency and have enough capacity to cover the latency of L2, while L2 should be optimized for memory capacity to cover the latency of host memory accesses over OpenCAPI. Taking the memory resources presented in Section 4.4 into account, BRAMs are best suited for L1 since several mega bytes are available and due to their low read and write latency. URAMs are a good fit for L2 since also several mega bytes are available, but each primitive is larger, and requires a slightly higher access latency.

7.1.1 AFU Read Request Operation

Figure 7.1 shows a block diagram of the interaction between the AFU, the two levels of buffering and a host interface. In theory the host interface could be any current or future interconnect standard, but this particular architecture focusses on the bandwidth specification and cache line size of OpenCAPI 3.0 operating on a POWER architecture host system.
Because of the read-only nature of this buffer, there is a clear distinction between the control and data path, or request and response path in this case, which flow in opposite direction. The control flow starts at the AFU, which is able to request eight elements per cycle, each from any stream. An AFU request consists of a ready-valid signal pair and a stream identifier. Each Read Port module has its own logic to distribute the requests among the different L1 Control modules. Every stream has a separate controller in order to keep track of the current read and write pointers. This holds for both the L1 and L2 Control. If the last element of a cache line in L1 is read, a request is sent to the respective L2 controller which will read a new cache line from the URAM and write it in L1. This frees up an entry in the URAM and triggers L2 to generate a new cache line request for the host. The Request Generation module translates a

stream number into an address for the host and attaches a unique tag to each request, which also consists of a ready-valid signal pair. Meanwhile, both the L1 and L2 stream controllers calculate addresses to index the BRAM or URAM module respectively.

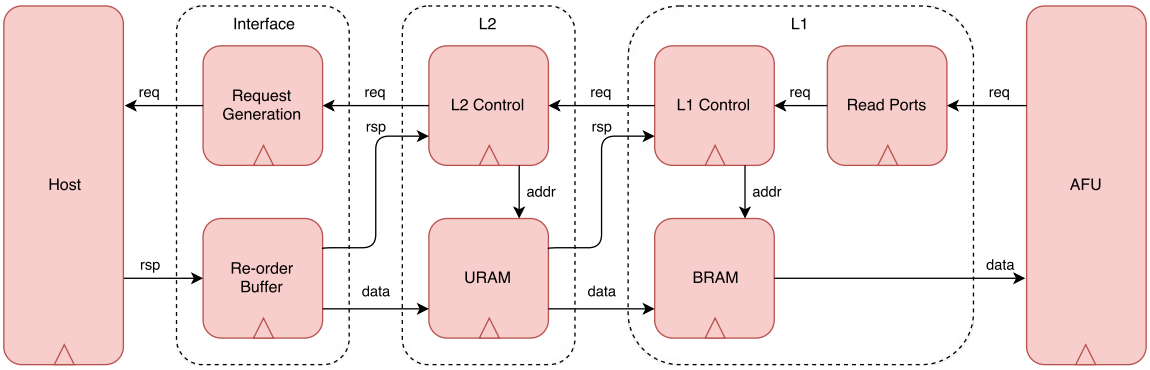


Figure 7.1: Diagram of the multi-stream buffer architecture.

7.1.2 Host Data Response Operation

The complexities of OpenCAPI, or any interconnect for that matter, could be abstracted away by using for example an OpenCAPI-to-AXI bridge. A similar bridge exists for CAPI 1.0 as mentioned in Section 3.2.2. Attaching an AXI interface to the multi-stream buffer would make it portable across interconnect standards, especially since AXI is the de facto standard in the world of FGPAs.
Independent of the chosen interconnect to the host, interconnect standards often allow response to be transferred out-of-order. Since the multi-stream buffer architecture expects response to come back in-order, a Re-order Buffer module is required. This module allows new cache lines to be written immediately to the URAM module, but only sends a response to the respective L2 Control module, which in turn updates the valid counter for that L2 stream pointer, if it is a consecutive response. If not, the response will be delayed until all previous response have been received.
When an L1 stream controller requests a new cache line from L2, the cache line is written simultaneously from the respective L2 buffer to all eight corresponding L1 stream buffers. Each individual connection between the URAM and BRAM module for such write operations is called a write channel. Ideally each stream has its own write channel, but that results in a complex wiring job since each write channel has a width of 128 bytes or 1024 bits (wires). Each of the modules shown in Figure 7.1 will be discussed in more detail in the remainder of this chapter.

7.1.3 Functional Stream Reset Operation

Before read requests from the AFU are accepted by the L1 Control modules, the desired stream has to be functionally reset. A functional reset request consists of a ready-valid signal pair, a stream identifier and two addresses to indicate the start and end location of the data in host memory. This allows streams to consist of a different number of cache lines per stream. Both addresses have to be cache line size aligned, which is 128 bytes in this case.
The functional reset interface is not shown in Figure 7.1 but is connected through a demultiplexer to each L2 Control module. From there, each L2 controller is connected to the respective L1 Control module which presents a one-hot signal to the AFU indicating whether a stream has finished or not. This signal is accompanied by a ready-valid signal pair such that the AFU can be notified when a functional reset has been accepted by both levels of stream controllers.

The functional reset interface input could for example be connected to a sideband signal of the interconnect or an MMIO region on the FPGA card.
If a functional reset request is accepted, the respective L2 Control module will start requesting cache lines from the host until its URAM is full. While requesting cache lines, the functional reset request is forwarded to the respective L1 Control module. If the request is accepted, the module will start requesting cache lines from L2 until its BRAM is full. A reason for not accepting a functional reset request is for example when the stream has not yet finished its current stream and thus still holds valid data which is not allowed to be overwritten.
When the AFU sends a read request for a stream, it is only accepted if the stream has been reset and if at least two valid cache lines are present in the BRAM, since in the desired configuration the AFU is capable of reading across a cache line boundary. Corner cases exist which will be discussed in more detail in Section 7.3.2.
While the interface modules have been briefly discussed from a functional perspective, the modules are not implemented in the final design. The Request Generation module has been built and tested, but was not integrated within the verification framework discussed in Section 8.1. The module can be found on the GitHub page of this project [68]. Integrating both modules is left as future work.

7.2 Multi-Stream Buffer Architecture Design Choices

This section motivates the design choices made before the implementation phase. The most important choice to be made is determining the size of each level of buffering, while making sure the memory primitives are fully utilized. Due to the wide data paths, routing delay has to be taken into account.

7.2.1 Buffer Depth Analysis

For any configured number of read ports P, at most P new cache lines are requested from L2 during a single cycle. Ideally these requests are processed and written to the corresponding L1 buffer in parallel. Since all stream controllers in both levels operate in parallel, transmitting new cache line requests from L1 to L2 occurs in parallel as well. The system architecture shown in Figure 7.1 suggests a single write channel between the L2 and L1 buffer. This could potentially become a point of congestion, depending on the AFU access pattern and the number of read ports, and result in an empty L1 buffer. An equal number of write channels as read ports is desired between L2 and L1 to minimize congestion. Each read port still has access to all streams, because the L1 buffers are duplicates of each other.
Figure 7.2 shows a generalized diagram of the memory organization for multiple write channels. Each write channel services a fixed subset of streams, even though not fixing this would be better. However, such a solution requires each write channel to be able to access every stream. This is very expensive in terms of wiring since each write channel is 1024 bits wide and drives multiple BRAM slices. A BRAM or URAM slice is an array of possibly multiple memory primitives that share the same write interface. In principle, a slice could contain any number of streams. However, both the BRAM and URAM slice serve the same number of streams, be it with a different number of cache lines. In the case of the BRAM slice, the write channel drives as many BRAM arrays as there are read ports configured. This means that all BRAM slices with the same slice number are driven by the same URAM slice. In the figure this is shown by placing the BRAM slices horizontally next to each other. When multiple L2 requests for the same slice occur in the same cycle, the different requests are serviced in parallel by the

corresponding stream controllers and merged using an arbiter for example before accessing the URAM slice, since only one read and one write interface is available per slice.

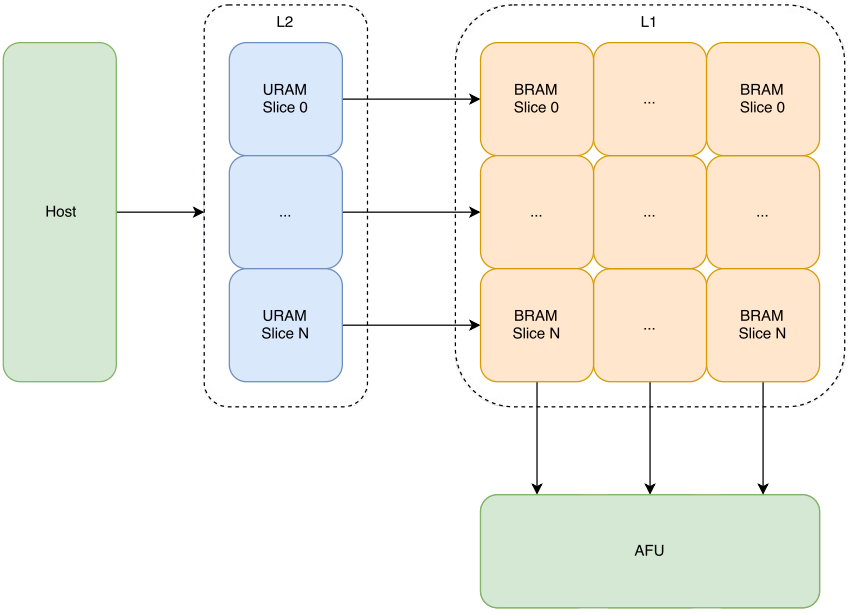


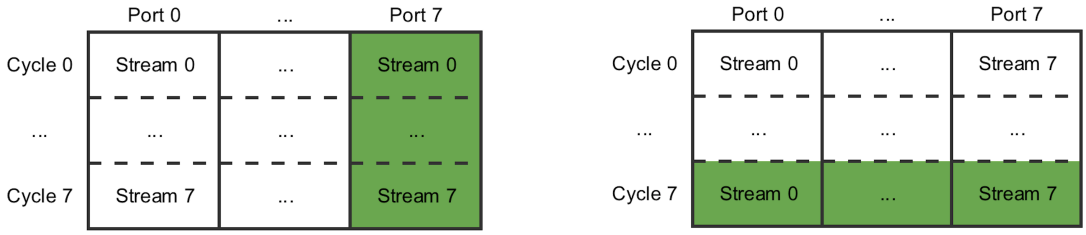
Figure 7.2: Diagram of the memory organization for multiple write channels.

AFU Access Patterns

For any number of read ports, there are two distinct access patterns that both generate the maximum number of new cache line requests from L1 to L2 in one cycle. Figure 7.3 shows the requested stream per read port per cycle. This is under the assumption that each cache line consists of as many data elements as there are read ports. Another assumption is that each stream is read by starting from the first data element at offset zero. All read ports have a predefined priority, with port zero having the highest. That means that if multiple read ports request the same stream, the highest priority read port will return the first unread data element. Figure 7.3a shows one case where all read ports read from the same stream in each cycle, for example stream zero. When the last element of a cache line is read, a request from L1 to L2 is made. In this case, this happens once every cycle as illustrated by the green box. After P cycles of this pattern, P cache lines have been requested, but the requests are evenly distributed as one per cycle. In this example P equals eight.
Figure 7.3b shows the other case where all read ports read from different streams in a single cycle. For example, stream zero through P-1. If such a pattern is sustained for P cycles, also P requests are made from L1 to L2 but all in the same cycle, which results in a burst of L2 requests. In this example P equals eight.

Stream Exhaustion

When the burst access pattern is succeeded by the distributed access pattern and both request streams from the same slice, L2 requests made by the distributed access pattern are queued up behind those of the burst access pattern. This is under the assumption that L2 requests are queued in ascending order, starting with stream zero. Depending on various factors such as the number of streams per slice, the latency of the control logic and memory organization, and the number of read ports, an L1 stream buffer could get exhausted. This is undesired behaviour



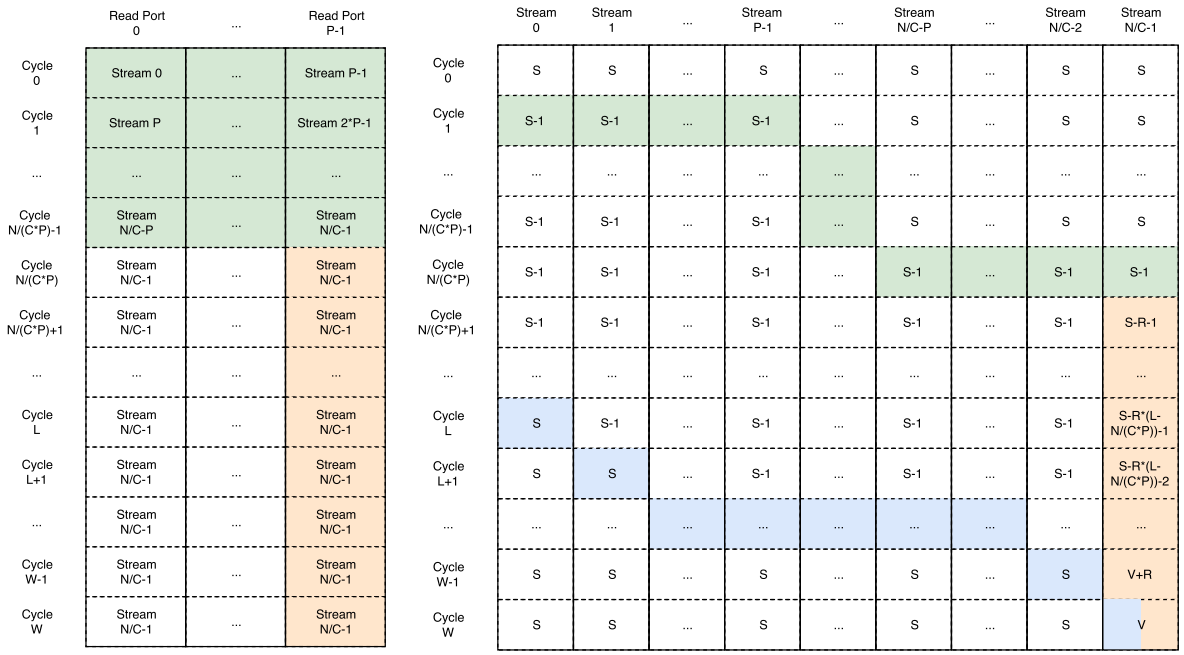
(a) Distributed L2 requests.

(b) Burst of L2 requests.

Figure 7.3: Eight read port stream access pattern.

and the L1 buffer should be adequately large.
Figure 7.4 illustrates a generalization of this worst case scenario by combining both access patterns. Figure 7.4a shows a configuration with P read ports. The access pattern shown is preceded by AFU read requests which resulted in a state where each stream has only one unread data element left in the current cache line. Therefore, when a stream is read, an L2 request is made.
For a configuration with a total number of streams N and C write channels, each slice services $\frac{N}{C}$ streams. To obtain a queue of L2 requests, the burst pattern is triggered using all read ports for $\frac{N}{C \times P}$ consecutive cycles to trigger all streams within this slice. The green boxes indicate a burst pattern read request resulting in an L2 request.
Next, stream $\frac{N}{C} - 1$ is read, the stream with the highest number, using the distributed access pattern. Under the assumption that streams are serviced in ascending order means that the highest numbered stream is in the last batch of read requests. If only this stream is read from here on, it will have to survive the longest number of cycles before a new cache line will be written into the corresponding L1 buffer, since the request is queued up behind requests from all other streams. The orange boxes indicate a distributed pattern read request, that could result in an L2 request. This depends on rate R and will be discussed later.

Figure 7.4b shows for each stream the number of valid cache lines present, as a result of the read access pattern shown in Figure 7.4a. In the initial state in cycle zero, each buffer is full with S valid cache lines. The valid counters per stream reflect changes due to the AFU access pattern in the consecutive cycle the AFU read request was made. As an example, in cycle one the green boxes indicate a change in the number of valid cache lines for the first P streams due to the burst pattern triggered in cycle zero in Figure 7.4a. The burst pattern continues until cycle $\frac{N}{C \times P}$.
Rate R is defined as $\frac{P}{E}$ and indicates how many cache lines can be requested per cycle for the distributed pattern. During the burst pattern it is known that a cache line from L2 will be requested. Therefore the valid counters are decreased by one. For subsequent cycles, this depends on the chosen configuration of the number of data elements per cache line E and the number of read ports P. This is under the assumption that P is always smaller or equal than E. Cycle L indicates when the first L2 request has been completed and a new cache line has been written in the corresponding BRAM array depends on the latency of the stream controllers and the memory primitives. Cycle L is defined as the sum of the L2 Control, URAM, and BRAM module latencies. Or in other words, the latency between issuing an L2 request and writing the corresponding cache line into the BRAM array. From cycle L onward, blue boxes indicate that a new cache line has been written and therefore the valid counter has increased by one. Since each BRAM slice has only one write interface, there is an imbalance between the generation and servicing of an L2 request by a factor of P to one. That means that in a single cycle, at



(a) Requested stream per read port.

(b) Number of valid cache lines present per stream.

Figure 7.4: Generalized worst case AFU access pattern.

most P L2 requests can be issued, while at best only one new cache line can be written. At cycle W, the initial L2 request made by stream $\frac{N}{C}-1$ has been written into the corresponding BRAM array. In essence, stream $\frac{N}{C}-1$ has to survive until cycle W and equals $\frac{N}{C}+L-1$. To calculate the buffer size S, an additional constraint V is added that indicates the minimum number of valid cache lines required to accept an AFU read request.

L1 Buffer Depth

To determine the L1 buffer size per stream S, Figure 7.4 and the associated access pattern is analyzed. Due to the initial burst pattern, at least one cache line has to be buffered. This is the cache line consumed within the first $\frac{N}{C\times P}$ cycles for every stream. In order to survive until cycle W, a minimum number of valid cache lines V must be present in the buffer. In between these two events, stream $\frac{N}{C}-1$ is read according to the distributed pattern and makes L2 requests at rate R. The number of cycles between the two earlier mentioned events is multiplied by rate R to obtain Equation 7.1.

$$S = R \times \left(W - \frac{N}{C\times P}\right) + V + 1 \implies \frac{P}{E} \times \left(\frac{N}{C} + L - 1 - \frac{N}{C\times P}\right) + V + 1, \text{where} \tag{7.1}$$

- S is the number of cache lines to buffer per stream,
- R is the rate at which L2 requests are issued during the distributed access pattern,
- W is the cycle in which the initially requested cache line by stream $\frac{N}{C}-1$ is written into the BRAM array,
- N is the total number of streams,

- C is the number of write channels,
- P is the number of read ports,
- V is the minimum number of valid cache lines present in the BRAM array in order to service an AFU read request,
- L is the sum of the L2 Control, URAM, and BRAM module latencies, and
- E is the number of data elements within a cache line.

Equation 7.2 shows the buffer size S when using the requirements for the multi-stream buffer as mentioned in Section 5.3. V equals two such that the AFU is always able to read across a cache line boundary. Only servicing a read request when there are two or more valid cache lines makes complex logic, for when only one valid cache line is present, unnecessary.

$$S = \frac{64}{C} + L - \frac{64}{C \times 8} + 2 \tag{7.2}$$

Since the number of write channels C is limited, a series of approximations of the buffer size can be made. Table 7.1 shows the number of write channels versus the buffer size per stream. The benefit of multiple write channels is obvious, but the cost of a write channel in terms of routing complexity has to be taken into account.

Table 7.1: Buffer sizes for a variety of write channels.

C	S
1	L + 58
2	L + 30
4	L + 16
8	L + 9

Based on this analysis, multiple configurations are possible, depending on the AFU access pattern, target FPGA, the latency of control logic and memory arrays, and many more. In the following sections the BRAM primitive configurations and routing complexity will be taken into account in order to make a final decision on the number of write channels.

L2 Buffer Depth

Determining the number of cache lines per stream to buffer for L2 is less complex compared to L1. The goal of the L2 buffer is to hide the latency of OpenCAPI. At best, one cache line is received per cycle from the host through OpenCAPI.
No real-world latency numbers are published for OpenCAPI at the moment of writing. Since one of the goals of OpenCAPI is to deliver a lower latency than current interconnect standards, a typical latency of PCI Express Gen 3 of 1 µs is used as a conservative upper bound as mentioned in 3.1.2. Assuming that the FPGA operates at 200 MHz, 200 cycles on the FPGA have to be covered and thus 200 cache lines have to be buffered. An access pattern where only one stream is constantly requested could occur. Rounding this up to the next power of two results in 256 cycles, or cache lines, to buffer per stream.

7.2.2 BRAM Sharing Among Streams

Section 7.2.1 made the assumption that the underlying BRAM primitives were utilized perfectly. While this is fine for a first-order analysis, the BRAM primitive configuration has to be taken into account during implementation. For example, implementing a single stream per BRAM primitive is very wasteful because only sixteen entries are used per BRAM which equals to a utilization of roughly 3%. Double pumping makes no significant difference. Therefore, BRAM primitives have to be shared among streams in order to achieve full utilization. Without double pumping, a BRAM primitive is fully utilized with 32 streams and with double pumping 16 streams are required. For example, in a configuration with eight write channels, or eight streams per slice, each BRAM primitive is only utilized for 50%. A deeper analysis has to be made in order to find the optimal number of write channels.

7.2.3 Multiple Write Channels Analysis

Section 7.2.1 showed that for a specific AFU access pattern outstanding L2 requests get queued up and a specific L1 stream buffer could become exhausted. Under the worst case memory access assumption and a configuration of 64 streams and one write channel, the required L1 buffer should contain at least 64 cache lines. It is obvious that with this architecture, the L1 buffer can get exhausted, resulting in stalls and degradation of throughput. The congestion of L2 requests can be improved by increasing the number of write channels between L2 and L1, such that potentially multiple new cache lines can be written in L1. Multiple write channels delay and might even prevent the exhausted state of the L1 buffer, depending on the number of write channels and the latency of surrounding logic and memory arrays. There is an incentive to minimize latency L because it enables to delay total exhaustion of the target L1 stream buffer. Since each L2 stream buffer ideally contains 256 cache lines, there should be more than enough cache lines ready to be written into the L1 buffer. The number of write channels is however bounded in two ways.

- BRAM Utilization impacts the number of streams per BRAM primitive and therefore the number of write channels which can be efficiently implemented. As shown in Section 7.2.2, each primitive is fully utilized with 16 or 32 streams. By dividing the streams across several BRAM primitives, the streams are interleaved. This efficient number of streams per BRAM means that the only possible number of write channels, which results in an evenly distributed number of streams per slice, is either one, two or four. It is desired to distribute the number of streams evenly across write channels in order to have a balanced system. If BRAM primitives are under-utilized, the required number of BRAM primitives will explode and the design will not fit within the KU15P's resource budget.
- Write Channel Wiring impacts the resource budget of the KU15P. Each channel requires 1024 bits. With multiple write channels, this results in a wiring nightmare and difficulty to route the design. However, the more write channels, the smaller the maximum congestion per slice.

Ideally, the system is balanced where each AFU read port is backed up by its own write channel as mentioned earlier. However, eight write channels result in an under-utilization of BRAM primitives, therefore an explosion of required primitives, and requires 8192 wires, making routing more complex. This leaves the two and four write channel designs. A big benefit of the four write channel design is that this results in using sixteen streams per BRAM primitive, double-pumped. Because of the double-pumping, the physical write channel can also be double-pumped

which means that 512 wires are used instead of 1024. Therefore, without any significant changes in logic, the same number of wires are required for a two or four write channel design, while the four write channel design avoids congestion by roughly a factor of two, under the assumption that latency L is equal for both, compared to the two write channel design.

7.2.4 Buffer Memory Organization

Figure 7.5 shows the final memory organization for the multi-stream buffer architecture, based on the previous sections in this chapter. The control flow starts at the AFU, which is able to request eight new elements per cycle. Each read port has its own logic to distribute the request among the different stream controllers (not shown). Every stream has a separate controller per level, while the cache lines per stream share memory primitives. This is because the minimum number of entries per memory primitive is much larger than what is required per stream. Sharing memory primitives between streams is not a problem, since cache lines are duplicated in L1 and each read port has access to a consecutive subset of cache lines per stream. Each AFU read port will at most request one element per cycle and therefore no conflict can occur per read port. Since all stream controllers operate independently and are connected directly to each other per stream, multiple new cache line requests in L2 can be serviced per cycle. By keeping the stream controllers independent and merging L2 requests only at the very last moment, when accessing the URAM and writing it to L1, throughput can be sustained for as long as possible. The only point of congestion is each write channel per slice. New data between the two levels is written in cache line granularity since complex logic would be required to replace single elements per stream for example.

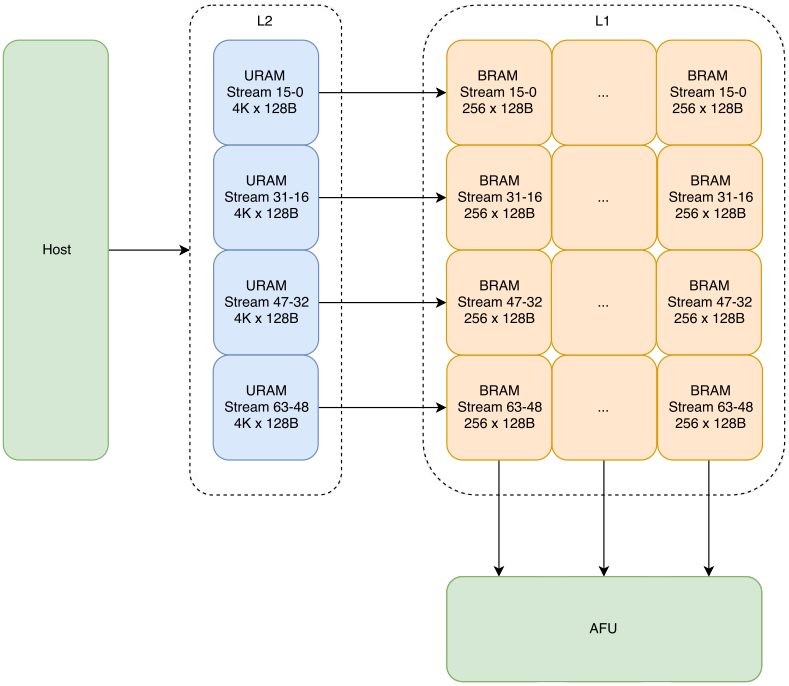


Figure 7.5: Memory organization of both buffer levels.

L2 Primitive Organization

OpenCAPI can provide at most one 128 B per 200 MHz cycle. This cache line is then written in the corresponding URAM of the requested stream. Since there are four write channels, L2 is

divided in four slices where each slice houses sixteen streams, each containing 256 cache lines. This results in 4096 entries. Since the memories are double pumped, two URAM primitives are required to obtain this number of entries. Cascading multiple 8 B wide primitives results in a cache line-width buffer.

L1 Primitive Organization

Similarly, L1 also consists of four slices where each slice houses sixteen streams, consisting of sixteen cache lines per stream. This results in 256 entries. However, all of these cache lines are replicated for each read port to solve the multi-read-port problem. When a new cache line is written from L2 to L1, that data is written simultaneously to all eight corresponding slices. By double pumping the BRAM primitive in a 512 entry 8 B wide configuration, a single BRAM primitive supplies all required entries at data element width. Then multiple primitives can be cascaded in order to obtain a cache line wide buffer.
For both levels the cache lines are direct mapped. During the functional reset of a stream, a start and end address are provided from which cache lines are automatically fetched. Based on the stream number and the physical address of a cache line, the address within a memory primitive is calculated. By direct mapping cache lines, the architecture might be extended in the future to act as a cache.

7.2.5 Expected Resource Utilization

The architecture described in this section has eight copies of sixteen cache lines per stream. In total, this will consume 256 BRAM primitives or 26% of what is available for the desired configuration. Each stream buffers 256 cache lines in L2. This will consume 64 URAM primitives or 50% of the total. Since sixteen streams share the same BRAM primitive, the multiplexing structure is smaller compared to the previous proposals. In this architecture, each read port selects the correct data element per slice, after which the correct slice is selected. This results in a 32:1 multiplexer at half the element width since the BRAMs are double-pumped. Roughly 2.6% of the available LUTs are required.
The large improvement in LUT utilization is due to the fact that BRAM primitives are shared by multiple streams. Therefore the primitive takes care of a large part of the selection structure, especially at these data widths. Double-pumping also has a large impact. A careful reader might ask why this architecture was chosen, since a significant portion of both BRAM and URAM resources will be consumed. The element-wise double-pumped architecture might work just as fine. While this might be true at first sight, the proposed architecture considers the anticipated routing problems and takes the FPGA topology into account. The BRAMs are situated in a vertical direction on the FPGA, which makes guaranteeing the same read latency for every location very difficult. Placing a subset of cache lines a special, smaller, L1 ensures that less cache lines have to have the same low access latency and makes closing timing easier. Besides that, this architecture consumes the least number of BRAMs of all proposals. Therefore most primitives are left for use by the AFU (and DLX and TLX layers).

7.2.6 Design Implementation Details

Now that the high-level design choices have been motivated, each level of the design will be discussed separately, based on the modules presented in Figure 7.1. The whole design is based around a request-response philosophy, where requests are made between modules and responses are given back. Implicitly this is done by using the design methodology described in Chapter 6. Explicitly this is done by, for example, viewing an AFU read as a request containing a stream

identifier, which is followed by a response containing the same stream identifier and associated requested data. The entire project can be found on GitHub [68].

7.3 L1 Control

The first level of the design, shown in Figure 7.6, consists of the read port logic, L1 stream pointer logic and a transpose module. In the desired configuration, each of the eight read ports has its own read port module and BRAM array, while each of the 64 streams has its own pointer module, indicated by N and M respectively. Note that the arrows drawn only represent the valid and, if present, the associated data signal. The associated ready signal is most often not explicitly drawn.
The read port module calculates the address to index the respective BRAM array and requests new cache lines from L2 if necessary. Since each read port outputs the requested stream identifier as a decoded signal (one-hot), the input of the transpose module is a matrix with the number of read ports as entries, where each entry is the number of streams wide. Since each L1 stream pointer module is only interested in requests made for that particular stream, this input matrix has to be transposed, which is nothing more than a rewiring to make interfacing easier.

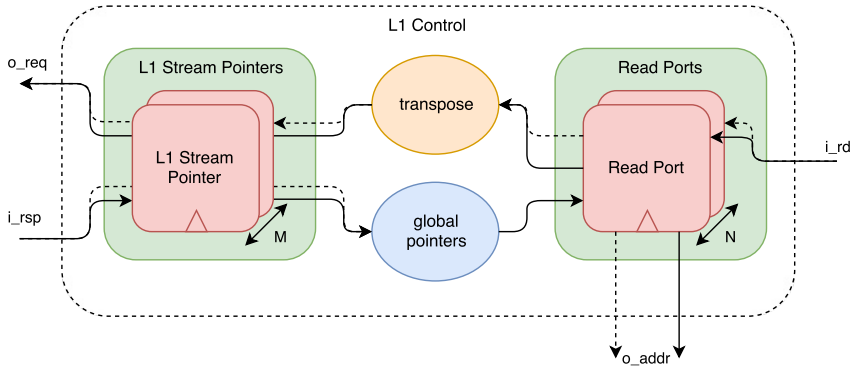


Figure 7.6: Diagram of the L1 control and data path showing the essential submodules.

7.3.1 L1 Stream Pointer

Figure 7.7 shows the L1 stream pointer, which is a controller to keeps track of the current read pointer within the BRAM and requests new cache lines from L2 when necessary. Each stream has its own dedicated controller and all controllers are identical. Since multiple streams share a single BRAM, care must be taken in properly calculating the addresses and updating the global pointer per stream, depending on the number of read ports accessing that particular stream per cycle. In the desired configuration, that could be any number of read requests between zero and eight, the number of read ports.

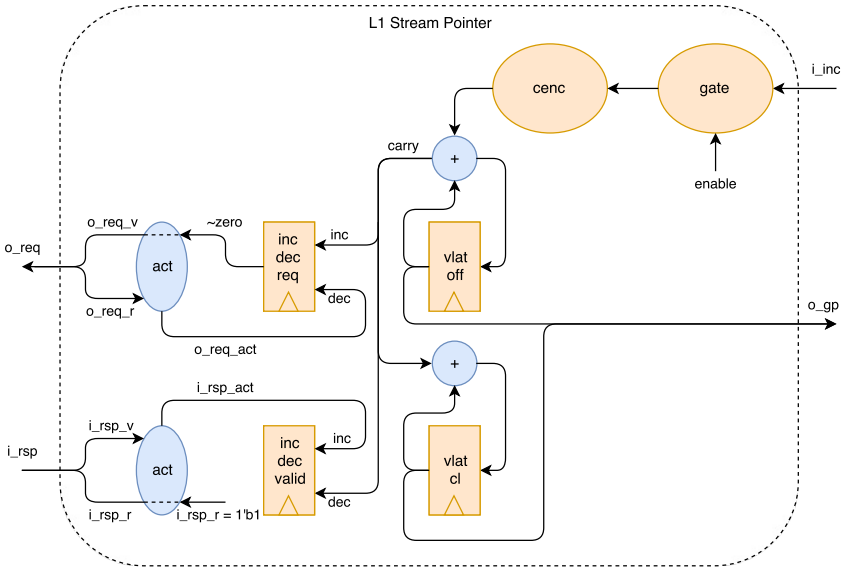


Figure 7.7: Diagram of the L1 stream pointer module showing the essential submodules.

Global Offset and Cache Line Pointer

At the heart of each L1 stream pointer is the global stream pointer. This pointer keeps track of which address has to be read next in the BRAM to obtain the next data element. The pointer consists of two parts, one to indicate the cache line, ranging between zero and fifteen, and one to indicate the offset within the cache line, ranging between zero and seven. This global pointer is used by the read port modules to calculate the address for each AFU read request. Each cycle, all AFU read request stream identifiers are presented to the L1 stream controllers as a matrix of the number of AFU read ports times the number of streams and each stream receives a vector with the number of read ports as width. This is a one-hot encoded vector, indicating which read port (if any), made a request for this particular stream. If so, the bit is high, otherwise it is low. This vector is then fed into a count and encode module which counts the number of asserted bits. This number in turn is added to the global pointer.

Valid and Request Counters

Besides the global pointer, two counters are present which can be incremented and decremented. One counter keeps track of the number of valid cache lines in the BRAM, where each cache line consists of eight elements. This counter puts back-pressure on incoming AFU read requests when there are no valid cache lines to read. In order to minimize the size of the counter and because the cache lines are assumed to be 128B aligned and therefore the offset within each cache line starts at zero, the counter granularity is in number of cache lines. In the desired configuration, each stream has at most sixteen valid cache lines in L1. Initially the number of valid cache lines is zero. The amount is increased when L2 write responses are received which indicate that a new cache line has been read from the URAMs and written in the BRAMs. The amount is decreased when the carry bit of the offset part of the global pointer goes high. This means that the seventh element within cache line N has been read and that the next read element will be the first element of cache line N+1.
The second counter keeps track of the number of outstanding requests to the associated L2 stream controller. This means that as long as the number of valid cache lines within L1 is less than sixteen, requests have to be made to the L2 control logic to write new cache lines

in the BRAM organisation. The purpose of this counter is keep the L1 buffer filled. Inversely to the valid counter decrement condition, when the carry bit is high, the request counter is incremented because a cache line has been completely read and therefore it has to be replaced with a new valid (unread) cache line. Finally, when an L2 stream request has been accepted, the request counter decrements by one.

Functional Reset Behaviour

When a functional reset occurs for this stream, it means that the corresponding L2 stream controller has been functionally reset successfully. A reason for this not to occur is for example when the respective stream still has valid data from the previous functional reset for this stream. When a functional reset of an L1 stream controller occurs, the output of the valid counter is set to zero and the output of the request counter is set to sixteen (number of cache lines per L1 stream). For the particular stream, the functional reset interface between the L1 stream controllers and the AFU is deasserted. At this point, the L1 stream controller is waiting to have at least two valid cache lines available before servicing AFU read requests. The reason is that the AFU is able to read across a cache line boundary and in order to service those, both cache lines need to be present.
As mentioned earlier, streams are directly mapped onto the memory organisations. During a functional reset, also a subset of the EA of the first cache line for this stream is sent from L2 to L1. The part of the global pointer which indicates the current cache line gets assigned this address as a starting value.
Each L2 stream controller keeps track of the current address and end address of each stream. When the end of the stream is reached, the corresponding L1 stream controller will be notified. At that point, the conditions for accepting an AFU read request change since now a read request will not only be accepted when there are two or more valid cache lines, but also when there is only one valid cache line present. After the final cache line has been fully read, the functional reset interface is asserted to indicate the end of the stream has been reached.

7.3.2 Read Port Module

Figure 7.8 shows a high level diagram of the read port module. The purpose of a read port module is to calculate the BRAM address and update the global stream pointer based on the requested stream by the AFU. An AFU read request consists of a valid bit and the requested stream number. Since the AFU read request drives two separate control flows, both have to be synchronised. That means that no new AFU read request can be accepted until both flows have finished and are ready. In order to achieve this, the base combine module is used from the library. In order to update the L1 global stream pointer, each read port module decodes the stream identifier requested by the AFU as a one-hot signal. All decoded stream identifiers together form a matrix which is transposed and fed to the respective L1 stream controllers as discussed previously.

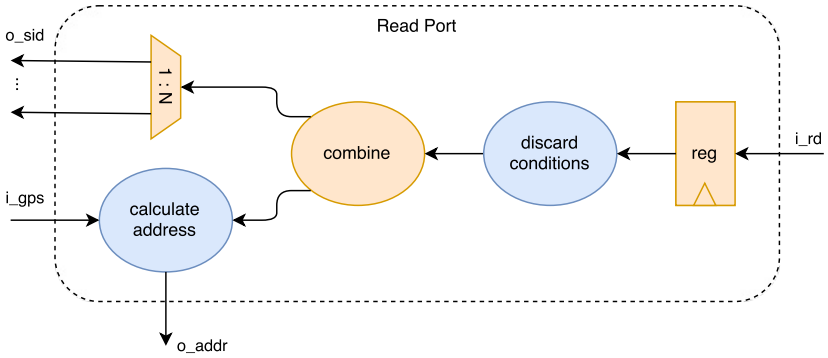


Figure 7.8: Diagram of a read port module showing the essential submodules.

BRAM Address Calculation

The calculated address per read port depends on the requested stream and the requested streams of the previous read ports during that cycle. What this means is that the read ports have a predefined order, in the desired configuration ranging between zero and seven. When multiple read ports request the same stream, assuming all upstream modules are ready, read port zero will read the first unread element in the data stream, read port one the second unread element and so forth. This means that the address for read port zero equals the global stream pointer, provided by the respective L1 stream pointer module. If read port one requests the same stream, its address is equal to the global pointer plus one, since the previous read port already reads the data element at the global pointer address. In a general, this relationship can be expressed as shown in Equation 7.3.

$$A(p,s) = \begin{cases} g(s[p]) & \text{if } p = 0 \\ g(s[p]) + \sum_{n=0}^{p-1}(s[p] == s[n]) & \text{if } p > 0 \end{cases} \tag{7.3}$$

Here A is the BRAM address, p is the read port number, s is an array with the stream identifier of every read port in this cycle and g is an array of the global pointer of each stream, provided by the L1 stream controllers. Calculating the address for any read port is basically a function of the read port number and all requested streams for that cycle. This expression consists of two parts, one statically generated as hardware, and one dynamically assigned during operation. Indexing the array of requested streams is done statically in hardware and is therefore indicated by [] brackets. The global pointer array is dynamically indexed using the result of the requested stream array and is indicated by () brackets. The expression also shows that, depending on the read port identifier, different logic for address calculation is generated from the same template.

Preventing Deadlocks

With the previously described implementation, deadlocks are possible in one or multiple read port modules. In such a case, a read request for some reason can not be serviced and therefore requests for that particular read port will not make any progress. In the process, this will stall any other read requests until the deadlock has been resolved, or more common, stall for an infinite amount of time. There are three distinct scenarios where a deadlock in a read port module can occur.

- Servicing a read request before functional reset results in reading invalid data and should therefore be discarded.

- Servicing a read request after a stream ended also results in reading invalid data. When the L1 stream has ended, which also implies L2 has ended, servicing read requests should be discarded.

- Termination of a stream mid-cycle occurs when during a single cycle multiple read requests are made for the same stream, but after a subset of the requests (in increasing port number order), the stream ends and the successive read ports will read invalid data.

In either case, the read request should be discarded to prevent a deadlock. This is done by testing various conditions and de-asserting the valid read signal before it reaches the `combine` cell shown in Figure 7.8. The reason is that when a read is discarded, the two upstream control flows are not allowed to notice the request or invalid data will be read and the global pointer will be updated while invalid data was read.
Figure 7.8 also shows a discard conditions module, which tests the deadlock conditions. While a properly designed AFU should use the provided functional reset interface to decide if a read request for a particular stream is valid, the implementation assumes a naively designed AFU. The first two scenarios can be solved by checking the output reset end signal provided by each L1 stream controller. This signal (`invalidate_rd`) is asserted when the stream has not yet been functionally reset or when it has terminated. Therefore, if for the requested stream this signal is asserted, the read request should be discarded. A discard simply de-asserts the valid signal from the ready-valid signal pair.
The third scenario requires to check multiple conditions. Reading out-of-bounds only occurs when the L2 stream controller is finished and the last valid cache line in L1 is being read. If within a single cycle a read port requests the last element from the last valid cache line and a successive read port requests the same stream, the requested element is out-of-bounds since the stream has ended. If the cache line offset carry bit of the calculated address is asserted in the same cycle as the previous two conditions, the read request should be discarded.

An implication of this approach is that if the AFU makes a request and it is discarded, there is no way of knowing in the current implementation. Currently the only guarantee is that each read port operates in-order, but responses can come back to the AFU with different time offsets between read ports, depending on the amount of back-pressure on upstream modules. The current implementation provides the stream identifier as a response with the requested data element. Since streaming accesses are always in the same order, the AFU can determine the order of the response data.
The problem could be solved by, for example, providing the AFU with a discard signal per read port which will be asserted when this occurs. Another solution could be to associate a unique identifier (UID) with each AFU read request, which might as well be the address the BRAM was indexed with, since it will not be reused unless that index was read. The UID will be immediately returned to the AFU, after which the AFU can act accordingly.
However, in order to minimise wasted cycles of processing discarded read requests, the AFU could pay attention to the output reset end signal, which is asserted when the L1 stream has ended. Therefore no more valid data is present for that stream and no more requests have to be made. This solution is sufficient for the first case, but not for the second case since this output signal is not updated until after the current cycle, since a register has to be updated. For the third case, at most the number of read ports minus one reads are discarded. Another solution is to reset the AFU with the same start and end EAs as the L1 and L2 control modules. Then counters are used to keep track of the number of received responses for each stream within the AFU.

7.4 BRAM Organization

The memory organization has been discussed extensively. The idea is to have a BRAM column for each AFU read port, and each read port has access to identical data. Due to the fixed configurations of BRAM primitives and the need for four write channels, each BRAM column consists of four slices, where each slice holds one forth, or sixteen streams, which each hold sixteen cache lines per stream. The BRAM primitives are double-pumped in order to utilize them fully.

7.4.1 Ready-Valid Memory Interface

A BRAM primitive is inherently not directly compatible with the design methodology used. Memory primitives, either read or write, consist of three signals: enable, address, and data. In such a case a latch oe module is used, which is similar to a register with a so-called output enable signal, to act as a read enable input of the BRAM. This signal is triggered when the input is valid and the upstream module is ready.
This solution works when the operating frequencies of the control and data path are the same. When using the design methodology with back-pressure, it is important to control the enable signal of the registers within the BRAM as well or things will not work properly when there is back-pressure. For example, when using a two cycle read memory latency, two `latch_oe` modules could be cascaded. However, since our data path is operating at twice the control frequency, a different scheme has to be used, while still supporting back-pressure. By switching to a credit-based interface, using source and sink compatible modules, back-pressure does not have to be supported in the data path but is supported by using a FIFO in the credit source module. Any additional registers within the BRAM read path can be enabled all the time, since they do not have to support back-pressure. Implementing a more sophisticated register enable signal is strictly a power optimization.

7.4.2 Double-Pumped and Credit-Based BRAM

Due to the requirement of double-pumping the BRAMs, switching to a credit-based interface is easier since it removes the requirement of controlling the pipeline registers within the BRAM in order to allow for back-pressure. This is at the cost of a six entry, 16B wide FIFO to act as the credit sink. Also multiple register stages are used in the data read path to be able to route the output signals back to the AFU. Examples are the additional pipeline register within the BRAM primitive and the FIFO within the `credit_snk` cell.
Figure 7.9 shows a high level diagram of the implementation of a single BRAM column using a credit-based interface. This module will be instantiated as many times as there are AFU read ports and the respective write channels are tied together.
The BRAM slice shown in the figure is a concatenation of eight BRAM wrapper modules to form a cache line wide memory. A BRAM wrapper module is an abstraction of the `clk2x` domain. It is basically the BRAM primitive instantiation, immediately followed by an 8:1 MUX at half the data width (8B) to select the requested element of 16B from the cache line of 128B. Then there is a free-running register at `clk2x` for the enable signal of the read pipeline register. For clarity, the write interface of each BRAM slice is not drawn, but in the desired configuration, four write channels are present.
The control path is shown in the upper half of the figure and operates at `clk1x`. The lower half is the data path and operates at `clk2x`. Since the `credit_src` cell does not depend on a clock for the valid logic, an input register is added. The toggle flip flop acts as the LSB for the cache line read address. Instead of using the latch oe module to generate a read enable signal, a

register is used where the output ready signal is constantly high. The reason is that the FIFO inside of the credit sink module handles received data from the BRAM. The ready-valid outputs of the `credit_src` cell generate an act signal which is the read enable of the BRAM slice and it toggles the read address LSB flip flop. The `vlat` cell shown at the output of the BRAM acts as an alignment register to obtain the full data width again before it goes into the credit sink.

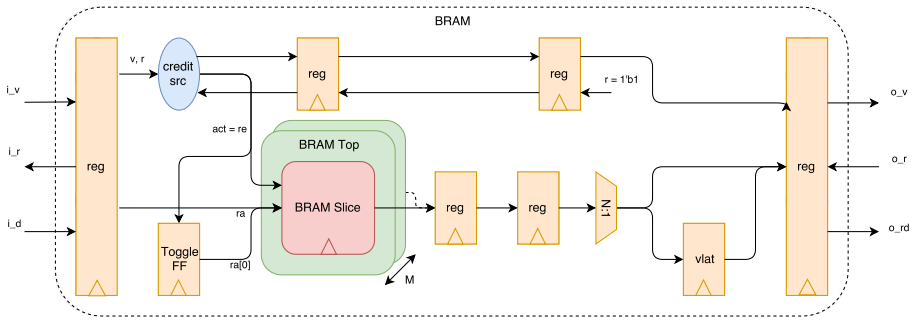


Figure 7.9: Diagram of double-pumped BRAM column using a credit-based interface.

7.4.3 Absence of Write Channel Back-Pressure

Usually the ready signal is supplied by an upstream (output) or downstream (input) module, but an example of a fixed ready signal is the input response from L2 when a new cache line has been written in the BRAM organisation. This ready signal is always asserted, which means that response can always be serviced right away. The reason is that there is no back-pressure present on the write interface of the BRAMs, since the only case where back-pressure is desired is when the read and write port have a conflict. This will never occur in this implementation since a read to a specific address is only issued when that particular cache line is known to be valid, and a cache line is only written to a particular address when it is known to be invalid.

7.5 L2 Control

The second level of the design, shown in Figure 7.10, consists of the L2 stream pointer logic, Round-Robin multiplexers and URAM organisation. The input request signal `i_req` is connected to the L1 control output request signal `o_req` and is as wide as there are streams. When one or multiple of these bits are asserted, a new cache line for that stream has to be fetched from the URAM organisation and written into the BRAMs.
However, since there are multiple L2 stream controllers within the same slice, a Round-Robin multiplexer grants read access to only one stream per slice. In the desired configuration, this is a sixteen-to-one Round-Robin multiplexer. In general, there are M slices, each with there own output address signal `o_addr`.
When a new cache line is fetched from the URAMs and written into the BRAMs, the corresponding L2 stream pointer decrements its valid counter and therefore requests a new cache line from the host. Since there is only one channel, for example OpenCAPI, to make these requests on, the same Round-Robin multiplexer modules are used to arbitrate between requests from all streams. Each slice instantiates its own Round-Robin multiplexer module after which a final Round-Robin multiplexer arbitrates between the slices.

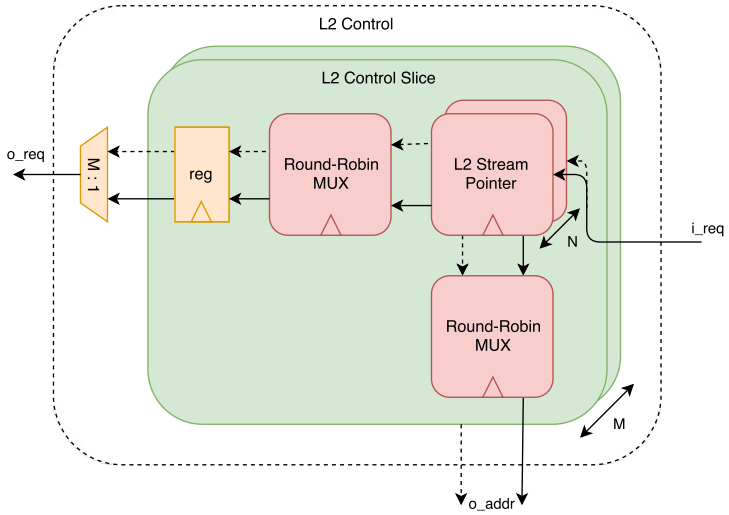


Figure 7.10: Diagram of the L2 control and data path showing the essential submodules.

7.5.1 L2 Stream Pointer

Figure 7.11 shows the L2 stream pointer module, which is similar to the L1 stream pointer module. The input request signal `i_req` is connected to the output request signal `o_req` from the L1 stream pointer module. This signal is asserted when a cache line has been fully read by the AFU and indicates that the respective L2 stream pointer should fetch a new cache line from the URAMs and write it into the BRAMs.

Global Cache Line Pointer

The input request signal is first connected to a gate module, which only passes the input signals if the enable signal is asserted. In this case, the enable signal checks if there is at least one valid cache line available in the URAM for this stream. Then, the act signal `s0_rd_act` is generated and enables the cache line pointer, which keeps track of the current address of a particular stream. When a valid input signal occurs, the respective URAM address to be read is presented at the output signal `o_addr`. This output signal is used to index the URAM slice.

Valid and Request Counters

Similarly to the L1 stream pointer module, there is a valid and request counter present. Instead of keeping track of sixteen cache lines per stream, as was the case for the L1 control, the L2 controllers keep track of 256 cache lines, as shown in Section 7.2.1. Both counters operate in the same way as their L1 counter part, except that the act signal mentioned earlier drives the request counter increment and valid counter decrement signals. The reason is that when an input request is received, the valid counter has to be decremented since a cache line will be transferred to the L1 BRAMs. Also, the request counter is incremented since the transfer of a cache line entails an empty line in the URAMs.

Functional Reset Behaviour

On reception of a functional reset, the request first goes to a gate module, in order to assess if a functional reset is permitted. This is only the case when the respective stream has consumed all of the cache lines from the previous data stream and has no more valid cache lines in the

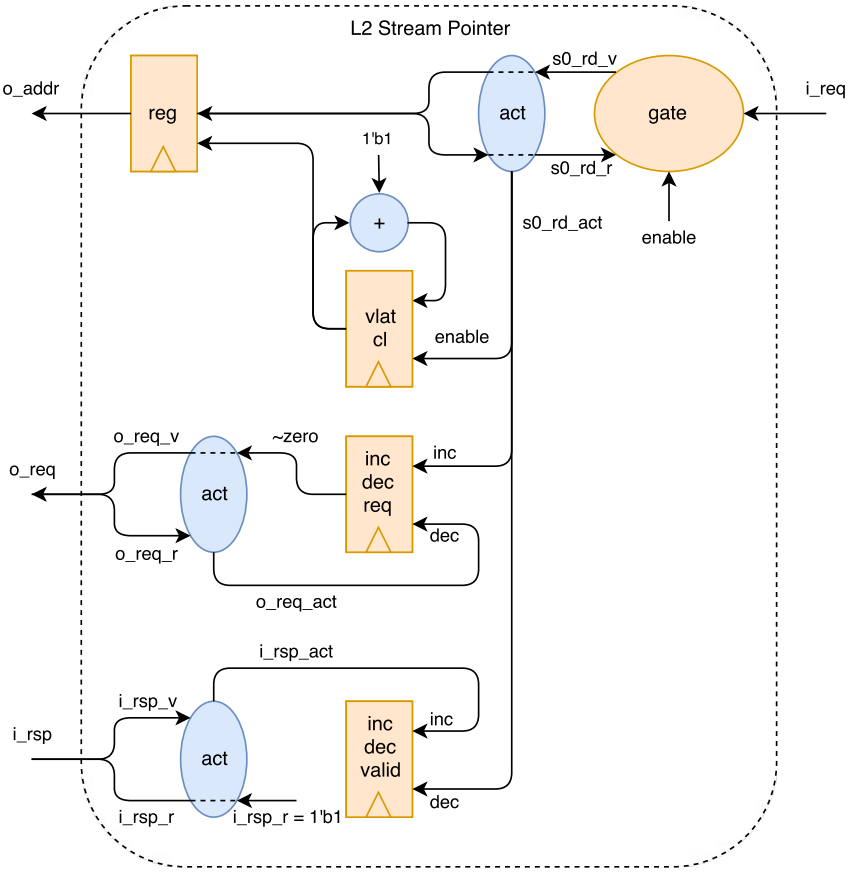


Figure 7.11: Diagram of the L2 stream pointer showing the essential submodules.

URAM, nor outstanding requests. Both counters are used to assess these conditions.
If the functional reset is permitted, the request will be forwarded to the respective L1 stream controller as well and an act signal will be generated. This signal is used to initialise registers for the reset begin and end EAs respectively. The begin EA will be incremented during operation and holds the next EA to be requested from the host to fetch a new cache line. This address is sent alongside an output request from this stream. The end EA is used to determine when the stream has ended and no more cache lines have to be fetched.
Also the cache line pointer is reset with the direct mapped address based on the begin EA signal. This same address is sent with the response data from the URAM to index the BRAM during the write operation. The valid and request counters are reset to their initial values, thus zero valid cache lines and 256 outstanding requests. In the case that a data stream contains less than the URAM capacity per stream (less than 256 cache lines), there is logic present which decrements the valid counter by one every cycle to slowly converge to the end of a stream condition.

7.5.2 Round-Robin Multiplexer

The Round-Robin multiplexer module consists of multiple Round-Robin multiplexers from the design library. Contrary to a traditional multiplexer, which uses an input select signal to select one out of multiple inputs, a Round-Robin multiplexer selects an input autonomously, according to a Round-Robin arbitration scheme, and produces an output select signal which indicates which of the inputs have been selected.

The Need for Pipelining

Round-Robin arbitration allows multiple requestors to share a common resource within a fixed time slot. All processes which are ready are serviced in a circular manner without the notion of priority. When a requestor has been serviced, it will go to the end of the line and will be the last to be serviced again, assuming that all requestors have a valid request.
For the modules used from the design library, this means that the multiplexer will first check if input N is valid. If it is, it will be selected and during the next cycle input N+1 will be assessed, and so on. If an input is not valid, the next input will be assessed until a valid one is found, or until a full circle has been made assessing all inputs, where none were valid.
Due to the possibility of not a single valid input, the arbitration logic could be assessing every input. Therefore, the critical path scales with the number of inputs, or requestors. In the desired configuration, each L2 slice has to merge requests from sixteen streams to share the associated URAM (the common resource). Naively a sixteen-to-one Round-Robin multiplexer from the design library would be instantiated, but due to the number of requestors, achieving the desired operating frequency will be challenging. Therefore multiple smaller Round-Robin multiplexers are instantiated and their output is captured in a register, as shown in Figure 7.12. Configuring each Round-Robin multiplexer in the top layer for four requestors results in an equally shared critical path on both sides of the register, neglecting wire delay. This results in a configuration where parameter N equals parameter M which equals four. Finally a second layer of a Round-Robin multiplexer is needed to merge the four outputs present after the register and present the downstream logic with a single chosen requestor. The associated data is presented at the output signal `o_req` and the selected requestor at `o_sel`.

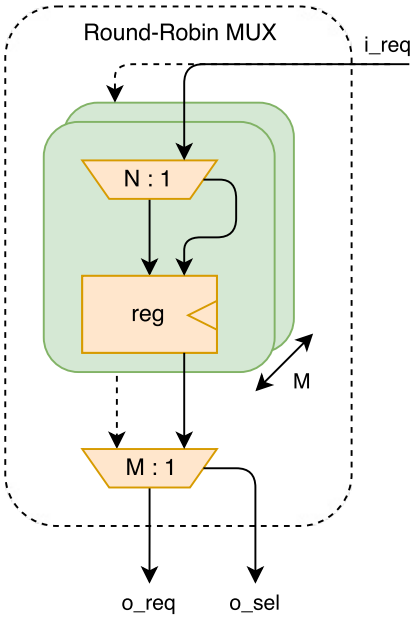


Figure 7.12: Diagram of the L2 Round-Robin multiplexer showing the essential submodules.

Use Cases within the L2 Control Logic

As shown in Figure 7.10, the Round-Robin multiplexer has two different use cases. First of all, it is used to merge the requested address from each slice of L2 stream controllers to access the associated URAM array. The data sent with a request consists of the current stream pointer, supplied by the respective L2 stream pointer module. The output select signal of the Round-

Robin multiplexer is the selected stream identifier and is concatenated with the stream pointer to obtain the URAM address.
This module is also used for merging the host requests from all streams, shown to the left of the L2 stream pointer modules in Figure 7.10. In the per slice use case, the data input of the Round-Robin multiplexer consists of the EA requested by each stream and the output select signal shows which stream has been chosen. When merging all slices, an additional four-to-one Round-Robin multiplexer from the design library is used, preceded by a register due to critical path considerations mentioned earlier. The input data of this final level of multiplexing is both the requested EA and the previously generated output select signal, or stream identifier. The output select signal of the final Round-Robin multiplexer indicates the selected slice and when concatenated with the previously obtained stream identifier (the output select signal of the Round-Robin multiplexer module) it represents the final chosen stream.

7.6 URAM Organisation

Contrary to the BRAM organisation, the URAM organisation has no data duplication and consists of a module called URAM Top which is generated M times, or as often as there are write channels between L1 and L2. Figure 7.13 shows the organisation and additional submodules used.

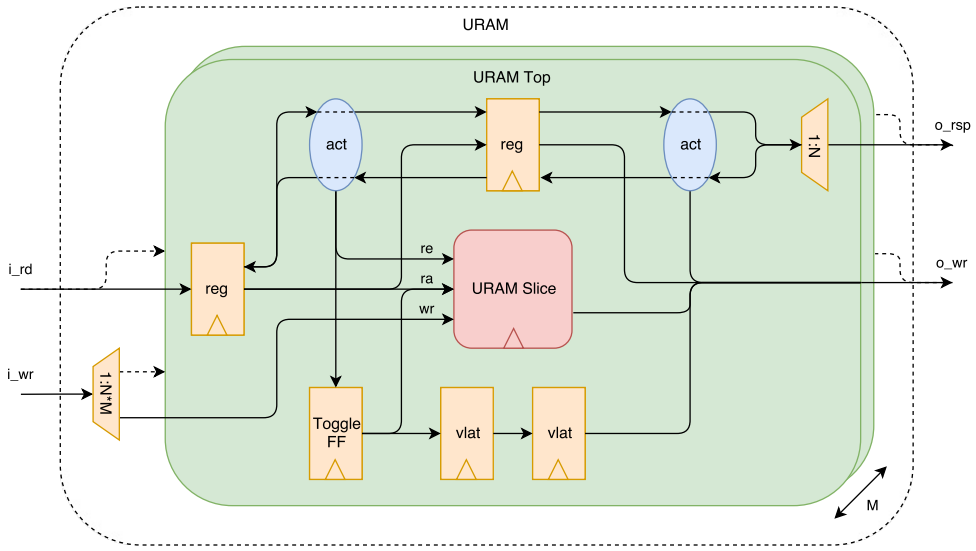


Figure 7.13: Diagram of the URAM array for M channels showing the essential submodules.

7.6.1 URAM Slice

A URAM Slice module consists of a concatenation of double-pumped URAM primitives. Each primitive is a 4k entry with 8 bytes per entry. By double-pumping, a 2k entry with 16 bytes per entry memory is obtained and now each entry functions as a data element. Currently the URAM primitive is configured to have a two cycle read latency, but depending on the implementation this can be changed to four cycles as well. An additional register stage is implemented to be able to route the output data from the URAM Slice to each BRAM array.

7.6.2 Write Interface

In order to write to the URAM Slice, the input write interface is present denoted by `i_wr`. This interface is double-pumped and consists of an address and half-sized data to be written. Part of the address indicates to which channel the data should be written, which is used as the select signal of the multiplexer.

7.6.3 Read Interface

The input read interface `i_rd` is connected to the output read interface of the L2 control module and consists of a ready-valid pair and an address (stream identifier and pointer). The interface is fed into a register after which an act signal is generated which acts as a read enable signal to the URAM Slice. It also drives the enable signal of a toggle flip-flop which operates at clk2x. Its function is to generate the least significant bit of the read address for the URAM Slice, which is concatenated with the address supplied by the input read interface. This bit is required due to the double-pumping since two addresses have to be read in a single clk1x cycle.
To keep the ready-valid pair synchronized with the URAM Slice, a register is used which operates at clk1x, shown above the URAM Slice in Figure 7.13. Also the address from the input read interface is registered here since it is required to write the obtained data into the respective BRAM arrays. The earlier mentiond LSB of the address is also registered, shown below the URAM Slice, but these modules operate at clk2x.
Finally, an act signal is generated which represents a write enable for the output write interface `o_wr`. The remaining signals of this interface are the address, registered in two different flows and the read data from the URAM Slice. The output write interface is directly attached to the input write interface of the BRAM arrays.
A response is sent to the respective L1 stream controller that a new cache line will be written into the BRAM arrays. Depending on the configuration, one URAM Top module contains N streams, which in the desired configuration is sixteen.

Chapter 8

Results and Discussion

This chapter explains the validation infrastructure built around the implemented design and shows the functional correctness of several corner cases. Afterwards it reports on the results obtained after synthesis and implementation and finally discusses these results.

8.1 Validation Framework

The validation framework consists of three modules to automatically test if the implemented design is functionally correct, shown in Figure 8.1. This simplifies validation of adjustments to the design, but also different configurations of the design.
The diagram follows the conventions mentioned in Section 6.3.1 but adds a yellow rounded rectangle for validation modules. The function of each of the modules will be briefly discussed.

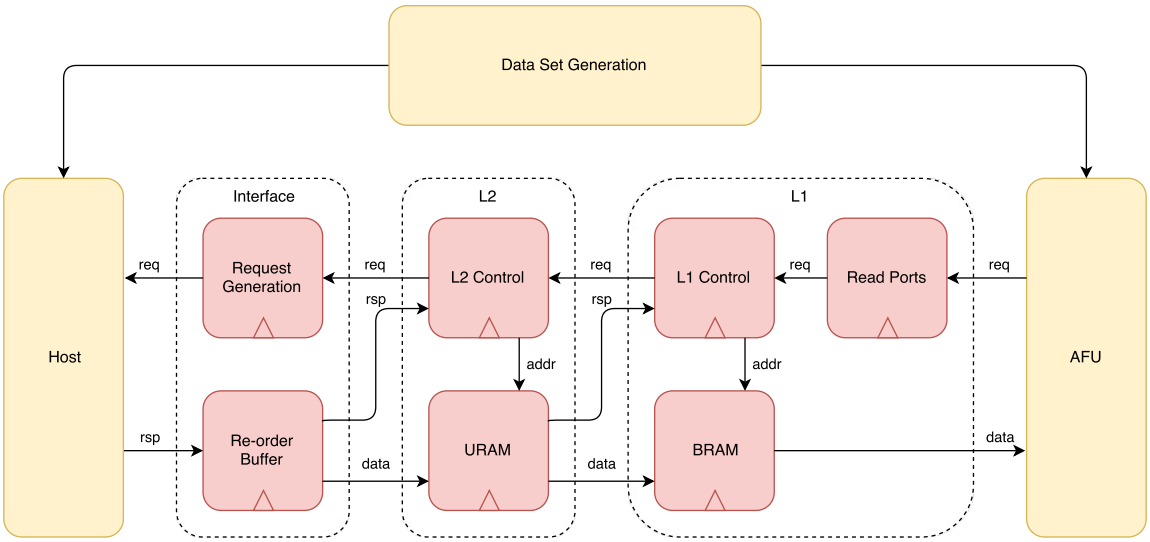


Figure 8.1: Diagram of the validation framework.

8.1.1 Data Set Generation Module

The function of this module is to generate a data set that acts as the stream data found in host memory. Cache lines have to be written in two half cycles and 16 B data elements are distributed between two physical addresses. Therefore, the data generation module initially generates a two-dimensional array with a number of entries equal to twice the number of data

elements required for all streams combined, with a data width of half a data element or 8 B. This array is then shuffled into an array usable for the Host module and for the AFU module. The Host module will write a new cache line to the URAM module in two half cycles. Therefore, the initially generated data has to be shuffled to obtain a two-dimensional array with the number of entries equal to twice the number of cache lines required for all streams combined, with a data width of half a cache line or 64 B. Each entry is a concatenation of eight half-sized data elements, either the even or odd indexed half data element from the original data set.
The AFU module also needs a shuffled version of the initial data set, but is different compared to the Host module. The reason is that the AFU module has to compare one or multiple data elements of 16 B, depending on the configuration received from the BRAM module or modules. The new two-dimensional array has the number of entries equal to the number of cache lines required for all streams combined, with a data width of a cache line or 128 B.

8.1.2 Host Module

The Host module consists of a register cell with a latency of one clock cycle that loops back the request made by the Request Generation module to the Re-order Buffer module. The number of cycles can be adjusted, depending on the host architecture.
When a valid request is received and the Host module is ready to accept, a task is initiated to write the next cache line for the requested stream into the URAM module. The Host module has a counter per stream to index the provided data set by the Data Set Generation module and updates the counter accordingly.

8.1.3 AFU Module

The AFU module generates read requests for the Read Ports module by initiating a task that has the read port and stream identifier as arguments. Each read request interface consists of a ready-valid signal pair and a stream identifier.
The module also validates data received from the BRAM array. Since each individual read port operates in-order, but read ports among each other do not, some logic keeps track of which read ports received valid data this cycle and updates counters per stream accordingly. These counters are used to index the shuffled data set provided by the Data Set Generation module. Since read requests can be discarded for various reasons, the AFU module also keeps global counters of the number of read requests made per stream and the number of valid data elements received. When the test bench terminates, a summary of these counters per stream is printed. It can then be assessed if reads have been discarded as intended, for example when reading after a stream has ended, or if a discard has occurred due to a bug.

8.1.4 Setup and Operation of a Stream

Before using the multi-stream buffer, the circuit has to be reset first. After this has occurred, the Host and AFU modules indicate that they are ready to receive requests and data. In one cycle, only a single stream can be functionally reset. This is done by sending a request to the functional reset interface consisting of a ready-valid signal pair, a stream identifier, and a begin and end EA. The EA should point to a 128 B aligned (required due to the direct-mapping of EAs to memory addresses). If the request is accepted, the L1 and L2 Control modules will start to request cache lines from the Host module by using the begin EA and fill their associate memory arrays. At any point after the functional reset request has been accepted, the AFU is allowed to make a read request for that particular stream. Depending on how many cache lines are valid in the BRAM arrays, the read request will be serviced right away or has to wait until

there is enough valid data to continue.
The begin EA is recalculated according to requests made to the Host module. When the begin EA surpasses the end EA, the stream has ended. This means that both the BRAM and URAM arrays still have valid data, so until there is no more valid data available, read requests for that stream will be accepted. The output functional reset interface will indicate when the stream has entirely ended. At this point, read requests made will be discarded and the AFU should use the read port to request unfinished streams, or functionally restart the stream again.

8.2 Functional validation

With the validation framework in place, changing the implementation and its functional validation is as easy as pressing a button. This allows to quickly functionally verify different configurations and corner case access patterns. This section visualizes various access patterns, proves the L1 buffer size analysis, and validates corner cases.

8.2.1 Multi-Read Port Access Patterns

To not overwhelm the reader, the configuration is slightly tuned down to 32 streams, four read ports, and two write channels. Section 5.2.2 showed four distinct access patterns possible for a configuration with eight read ports and eight data elements per cache line. The configuration used has four read ports and the same number of data elements per cache line, but this still allows us to show the correct operation of these access patterns.
Figure 8.2 shows the waveform of the four access patterns. The signal list contains both clock signals, the AFU read request interface showing the ready and valid signals and the stream identifier associated with each read port. Similarly the data response from the BRAM arrays is shown, accompanied by the ready and valid signals, and the stream identifiers. Also the cache line offset and cache line number are shown for stream five, six and seven. Note that the time dimension is not to scale.

All Reads from a Single Stream

The leftmost marker indicates the start of an access pattern where all read ports request the same stream. The data response with the associated stream identifiers appears five cycles later. Currently the first cache line is read and the starting offset is three. This is due to the fact that this stream was read at an earlier stage to position the stream pointer at the desired offset within the cache line to demonstrate other access patterns. There is a cycle latency between the AFU read request and the offset update, since the AFU read request first flows through an input register.

All Reads from Different Streams

The second marker indicates the start of an access pattern where all read ports request different streams. The response contains data from the expected streams. Similarly as in the previous access pattern, the streams have been read at an earlier stage. Therefore the starting offset is six and since one read request is made per stream, the new offset is seven.

Crossing a Cache Line Boundary

The third marker indicates the start of an access pattern where a cache line boundary is crossed. Currently, the offset is seven. Therefore when four requests for this stream are made, the first

request reads at offset eight and the other three requests read the first three elements from cache line one. This can be seen from the cache line offset and cache line identifier signals.

Crossing Multiple Cache Line Boundaries

The last marker indicates the start of an access pattern where two cache line boundaries are crossed. Similarly to previous patterns, some requests are made at an earlier stage. Therefore, the current offset is at seven for both streams six and seven. After making two read requests per stream, both offset seven from cache line zero and offset zero from cache line one are read.

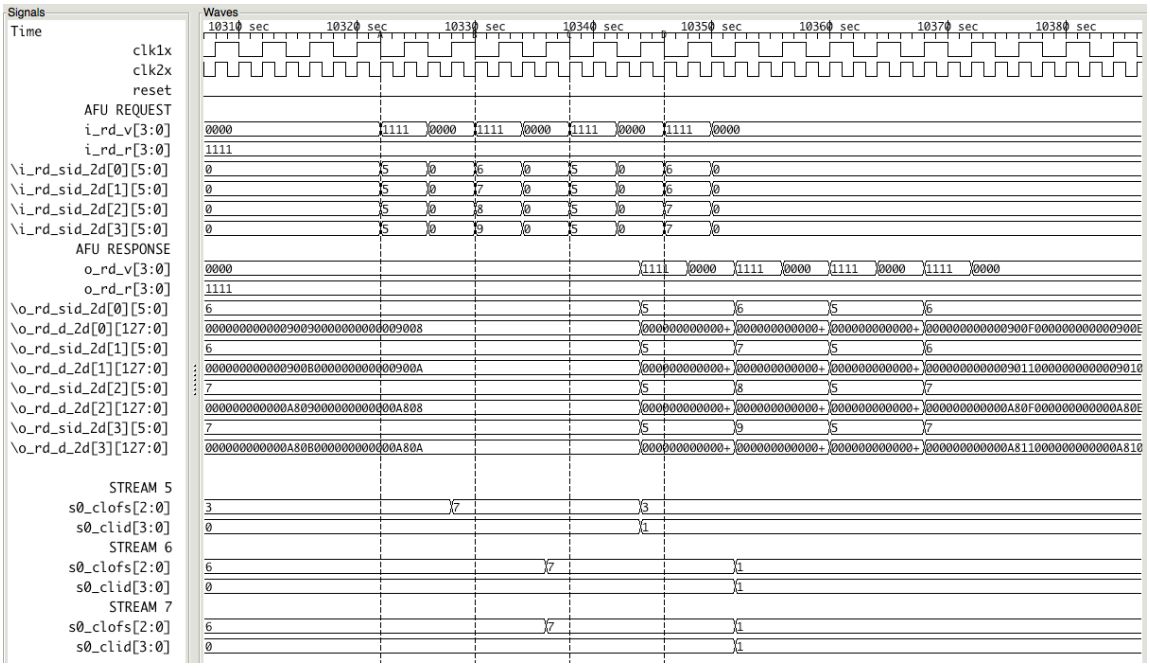


Figure 8.2: Waveform showing four distinct AFU access patterns.

8.2.2 L1 Buffer Depth validation

Section 7.2.1 analyzed the required L1 buffer depth for any configuration. When using the desired configuration parameters in Equation 7.1, latency L is the only unknown parameter. Chapter 7 discussed the implementation in detail and therefore the latency is known. Latency L is defined as the latency between the L2 Control module accepting an L2 request from L1, to committing the new cache line in the BRAM slice. The L2 stream pointer has a latency of one cycle, followed by one cycle latency of the Round-Robin multiplexer, followed in turn by three cycles in the URAM slice, and finally one cycle to commit the new cache line. Therefore latency L equals six clock cycles.
Table 7.1 showed that a configuration with four write channels should have a buffer size of $L + 16$ to accommodate for the worst case access pattern. The closest power of two is sixteen and therefore this many entries was chosen. However, taking latency L into account, the buffer size of each L1 stream should be 22 cache lines deep. This means that under the worst case access pattern and the assumption that no back-pressure will occur during run-time, six cycles of AFU read requests cannot be serviced. Depending on the access pattern of the AFU, this may or may not be a problem.

Functional Simulation

To illustrate this, a functional simulation was done and the results are shown in Figure 8.3. The signal window shows AFU read requests, the number of L1 valid and requested cache lines for stream fifteen, and the output of the Round-Robin multiplexer that arbitrates between sixteen streams to schedule read access.
The leftmost marker shows the moment L2 requests are triggered using the burst access pattern as described in Section 7.2.1. The next marker shows the start of the distributed access pattern. During the second cycle of the burst access pattern, the last element of each cache line is read. This is reflected by the number of valid cache lines signal `s0_ncl` in stream fifteen, since it is reduced from sixteen to fifteen. During the following cycles, a full cache line is read from stream fifteen and the valid counter decreases by one every cycle.

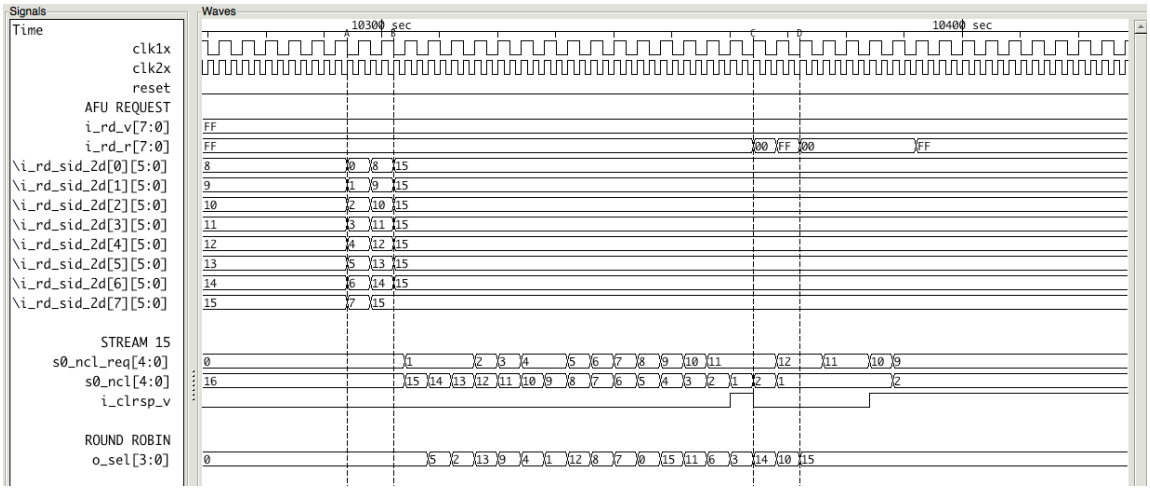


Figure 8.3: Waveform showing the worst case access pattern and the impact of the buffer size.

Discrepancy Between Analysis and Functional Simulation

The model used during the analysis assumed that L2 requests would be serviced in ascending order. Due to the nature of the Round-Robin multiplexer, the request for stream fifteen is scheduled earlier as shown in the waveform by signal `o_sel`, the output select signal of the Round-Robin multiplexer. As a consequence, a valid response `i_clrsp_v` for stream fifteen is received earlier than expected during analysis.
As expected, no AFU read requests are serviced for several cycles since the buffer is too small to handle this configuration. The third marker shows the first occurrence of this, when all read ports apply back-pressure to the AFU. Even though there is still one valid cache line present, read requests are only serviced when two or more valid cache lines are present to service access patterns that cross a cache line boundary, as mentioned before. Since the Round-Robin multiplexer scheduled stream fifteen earlier than expected in the analysis model, the valid counter is briefly incremented to two valid cache lines, which then removes the back-pressure. The forth marker indicates the second time back-pressure is applied for the same reason. In total, back-pressure is applied for six cycles. This is in accordance with the expectation to facilitate this access pattern without any loss in performance, the buffer should have had 22 entries. Instead the buffer contains sixteen entries and consequently no AFU read requests are accepted for six cycles.

8.2.3 Discarding Read Requests to Prevent Deadlocks

Section 7.3.2 discussed three conditions for which incoming read requests from the AFU should be discarded in order to prevent deadlocks. This paragraph validates this behavior by simulating the conditions.

Servicing Read Requests before Functional Reset

Section 7.3.2 mentioned that servicing read requests before a functional reset, or after a stream has terminated, results in a discarded request. No valid data is present and therefore no read requests are allowed.
Figure 8.4 shows the AFU request interface. The leftmost marker indicates a read request on read port one and the second marker indicates the same signal after the input register. Since the `invalidate_rd` signal is high, meaning that the request stream has not yet been reset (or has ended), the read request is discarded. Therefore the next downstream signal `s1_rd_v` is de-asserted.
Since this implementation also discards a read request when it is issued after the stream has terminated, no waveform is shown for this corner case.

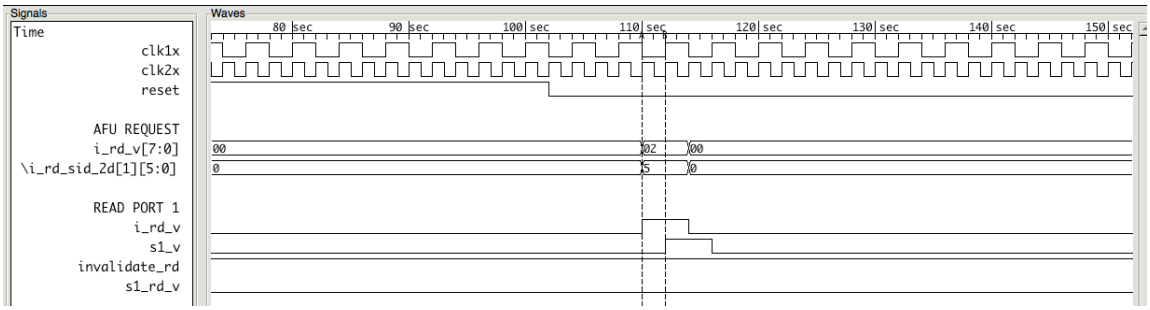


Figure 8.4: Waveform shows read request discard when issued before a functional reset.

Termination of a Stream Mid-Cycle

Another discard condition is when multiple requests are made during the same cycle, but within that cycle the last element from the stream is read. Therefore, one or multiple read requests have to be discarded since no more valid data is present.
Figure 8.5 shows the AFU request and response interface, followed by internal signals of stream five, and read port zero and one signals. The leftmost marker indicates an AFU read request on two ports, both for stream five. Stream five is currently at offset seven, with one valid cache line left. The second marker indicates the same signal after the input register (`s1_rd_v`). The out-of-bounds signal tests if the stream has ended and a request is made. If this is the case, as is in this example, the signal is asserted. This de-asserts the input valid signal `s1_rd_v_test` to the combine cell in the read port logic and therefore the read request is discarded.
Read port one, however, does not discard the read request because it will service the element at offset seven. Therefore the AFU response interface shows one valid output at the fourth marker. This valid signal belongs to read port zero, as expected.

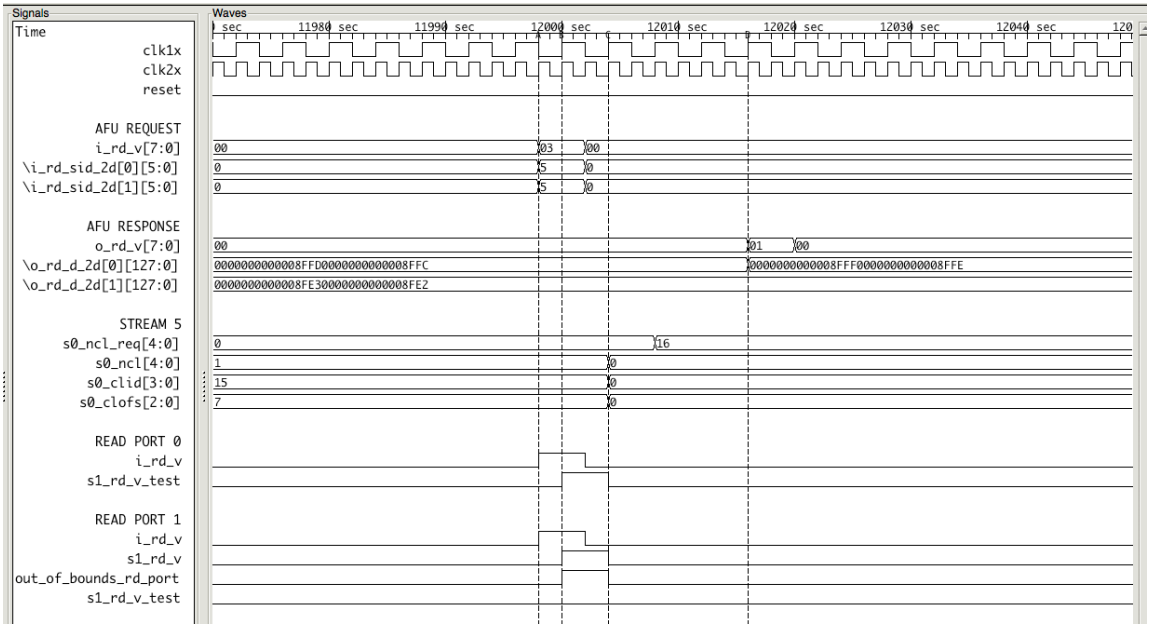


Figure 8.5: Waveform shows read request discard when stream terminates mid-cycle.

8.3 Synthesis and Implementation Results

Besides functional validation, the design has to be synthesized and implemented as well to verify the operating frequency and resource utilization. First the setup of the Vivado 2017.1 toolchain is explained and afterwards the obtained results for various configurations are presented.

8.3.1 Vivado Toolchain Setup

Since the multi-stream buffer is a sub-module in a larger design, obtaining implementation results such as resource utilization and timing is complex and with uncertainty. Typically an FPGA design is connected to the physical pins on the chip's package. Since this is not the case here, the tool has to be instructed neither to route wires to these pins, nor take the wires into account during timing analysis. Vivado provides a special mode for such an isolated approach called `out_of_context`. This mode is used to obtain the results presented below.
Typically, input and output signal constraints are provided, but this increases implementation time dramatically. It has happened that the implementation phase increased from 15 min to 5 hours for a small configuration of the design. Since all inputs go directly to registers and all outputs come directly off registers, no constraints are applied. Since an input signal constraint would only check the path from the pin to the directly connected register, such a test is basically meaningless. There is no logic and therefore it cannot be the critical path. The critical path is between the input and output registers and it is thus sufficient to specify clocks only.
All clocks are related by default. The two clocks used for the final design will come from the clock source. To make sure the tools include clock skew, an additional hierarchical design constraint related to the clock source is added. With this constraint the tools know which source is driving the clock. In the future, both clocks are expected to be driven by buffers, therefore a different buffer is chosen for each clock constraint.
One of the downsides of the `out_of_context` mode is that it will typically only indicate a best case scenario. When a sub-module is integrated in the final design, particularly when the

design consumes a significant part of the available resources, the reported performance of the sub-module will be at best the results obtained in the out_of_context mode [69].

8.3.2 Synthesis Results

Using the described Vivado toolchain setup, various configurations of the multi-stream buffer have been synthesized and implemented. Both the synthesis and implementation target the KU15P using the Vivado 2017. Since several versions are available within Vivado, the lowest-end part regarding temperature and speed grade is chosen to allow for variations (`xcku15p-ffve1760-1-e`).
In order to help FPGA designers with a specific workflow, Vivado provides various synthesis strategies. Strategies range from fast results to performance optimized. The latter is chosen for this project, in order to let the tool help as much as possible to obtain the target frequencies. For that reason, the `Flow_PerfOptimized_high` is chosen and turns off resource sharing, and decreases the maximum fan-out for example.
The timing constraints are set to 200 MHz and 400 MHz, respectively. When the tool finds a solution that fits the timing constraints, it will no longer continue to search for a possibly better alternative. Additionally, typically the timing constraints are higher than your actual target frequency because in an FPGA the wire routing delay will be dominant. To accommodate for both, usually a design is over-constrained. To find an optimum between the level of confidence that the design will work, and the run-time of the tool, the design is not over-constrained.
The synthesis results of various configurations are shown in Table 8.1 and Table 8.2. The same parameter naming scheme is used as in Equation 7.1. Table entry N indicates the total number of streams, C indicates the number of channels, P indicates the number of read ports, F indicates the `clk1x` constraint, WNS shows the worst negative slack, WHS shows the worst hold slack, and WPWS shows the worst pulse width slack.
To reduce run-time of the tool, the number of L2 cache lines per stream is decreased from 256 to 128. Getting data out of the memory columns and the fan-out from the URAM to the BRAM slices that increases with the number of read ports are the most difficult parts of the design to place and route. These complexities are still present, even with this reduction in cache lines.

Table 8.1: Synthesis timing and power consumption results for various configurations.

N	C	P	F [MHz]	WNS [ns]	WHS [ns]	WPWS [ns]	Power [W]
32	2	4	200	0.761	0.049	0.412	2.784
32	2	8	200	0.761	0.049	0.412	4.085
64	4	4	200	0.761	0.049	0.412	4.707
64	4	8	200	0.761	0.049	0.412	7.226

Table 8.2: Synthesis resource utilization for various configurations.

N	C	P	F [MHz]	CLB LUTs	CLB Registers	BRAM	URAM
32	2	4	200	11274 (2.16%)	13329 (1.27%)	72 (7.32%)	16 (12.50%)
32	2	8	200	14195 (2.72%)	15185 (1.45%)	144 (14.63%)	16 (12.50%)
64	4	4	200	23286 (4.45%)	25205 (2.41%)	136 (13.82%)	32 (25.00%)
64	4	8	200	28667 (5.48%)	28248 (2.70%)	272 (27.64%)	32 (25.00%)

For each configuration, the synthesis tool reports that the timing constraints are met and the resource utilization is in line with the expectations. The number of BRAMs is slightly higher

than expected. Since each BRAM slice requires a credit sink cell, a First-In-First-Out (FIFO) cell is used which uses a memory. Since the data width is equal to the data element width, two BRAMs are required per FIFO.
The longest path for each configuration is the internal path between a URAM array and an internal pipeline stage. However, the slack will dramatically decrease during implementation.

8.3.3 Implementation Results

After several initial implementation runs, it became apparent that small tweaks had to be made to the original design. The timing constraints where not met, even after several hours. While the synthesis results where promising with respect to meeting the timing constraints, the influence of the inherent location of memory primitives in columns became quickly clear. Table 8.3 and Table 8.4 summarize the obtained results for the same configurations as shown in the previous section.

Table 8.3: Implementation timing and power consumption results for various configurations.

N	C	P	F [MHz]	WNS [ns]	WHS [ns]	WPWS [ns]	Power [W]
32	2	4	200	0.006	0.030	0.412	3.226
32	2	8	200	0.012	0.030	0.412	4.997
64	4	4	200	0.014	0.031	0.412	6.713
64	4	8	200	-1.082	0.030	0.412	10.949

Table 8.4: Implementation resource utilization for various configurations.

N	C	P	F [MHz]	CLB LUTs	CLB Registers	BRAM	URAM
32	2	4	200	11052 (2.11%)	13329 (1.27%)	72 (7.32%)	16 (12.50%)
32	2	8	200	13980 (2.67%)	15185 (1.45%)	144 (14.63%)	16 (12.50%)
64	4	4	200	22963 (4.39%)	25205 (2.41%)	136 (13.82%)	32 (25.00%)
64	4	8	200	28607 (5.47%)	28411 (2.72%)	272 (27.64%)	32 (25.00%)

The configurations besides the 64 stream, 8 read port configuration share a common critical path. The BRAM slices operate at 400 MHz, but when the number of channels is increased, more multiplexing is required. At this frequency and with half data element sizes of 8 bytes, this path becomes too long. To solve this, the relay stations used by the ready-valid methodology came to the rescue, and changing a parameter resulted in starting a new implementation run immediately. However, this is not enough to close timing on the 64 stream, 8 read port configuration. The critical paths are the write channels, that have to drive eight BRAM slices each, and the combinatorial L1 address calculation by the read ports.
It is important to realize that the synthesis results came not even close to the results obtained after implementation. During the design phase of the multi-stream buffer, memory arrays were carefully analyzed. However, the impact of routing across the FPGA to each memory column has a much bigger impact than expected.

8.3.4 Integration with the OpenCAPI DLX and TLX

Section 4.4.5 showed the latest resource utilization results from the OpenCAPI DLX and TLX. Table 8.5 shows the results of the combination of both modules with the multi-stream buffer configured as a 64 stream, 8 read port. While nearly 10% of the LUTs are consumed, the

other resources see no significant change compared to the results presented for the multi-stream buffer.

Table 8.5: Implementation resource utilization for the 64 stream, 8 read port configuration of the multi-stream buffer plus the DLX and TLX.

CLB LUTs	CLB Registers	BRAM	URAM
47633 (9.11%)	37803 (3.62%)	279.5 (28.40%)	32 (25.00%)

8.4 Discussion

The results obtained of various configurations of the multi-stream buffer have been presented. Designing FPGAs at 200 MHz is challenging, especially when a uniform access latency is required from roughly 25% of the total available BRAMs. While the proposed design exploits different memory primitives for the L1 and L2 buffers to provide an uniform access latency, physical restrictions limit the attainable performance, both in terms of operating frequency and AFU request-to-response latency.

8.4.1 Extracting Data from Memory Columns

As initially anticipated, extracting data from the URAMs and BRAMs is complex at the target operating frequencies. The reason is that memories are in columns and relatively far away from the CLBs, as shown in Section 4.4.1. BRAM and URAM columns contain internal configurable pipeline stages, in order to help close timing and move data to its consumer.
The proposed architecture has taken this into account from the start and is therefore preferred over other architectures. it moves the complex to route path from the latency critical AFU request-to-response path, to the data transfer from L2 to L1. In order to close timing, additional pipeline registers are required on the output data ports of these memories.
For this reason, the BRAM and URAM slices have been extensively tested and implemented during the design phase. Closing timing within and between these modules was expected to be complex. Therefore multiple pipeline stages are built-in. This is also a good example of the strength of the design methodology introduced in Chapter 6. The relay stations allow pipeline stages to be easily modified. This allows for an additional cycle of latency to close timing. If that is not enough, an additional `reg` cell for example can be instantiated, either up- or downstream.

8.4.2 Critical Paths

As expected, as the number of channels increase, the URAM slices have to drive more BRAM slices that are scattered around the FPGA. Adding an additional pipeline stage for each write channel closed timing for the 64 stream, 4 read port configuration. Timing is barely met, but keep in mind that the tool will not continue endlessly when it has found an implementation that fits the set of constraints. The 64 stream, 8 read port configuration was not able to meet the timing constraints and future work includes improving the write channels for this configuration and the combinatorial read port path.
Also the read port logic, inherently combinatorial since the address calculation of the last read port depends on the stream requested by all previous read ports. With this amount of BRAM primitives scattered around the FPGA, the combinatorial path is forced to span across the FGPA as well and therefore fails the timing constraints.
Another complex path is the multiplexing outside of the BRAM slices. A data element has to

be selected from up to four channels, where each channel consists of a multiplexer to select the correct data element from a cache line size wide BRAM array. When the number of channels increases, additional pipeline stages are required. This conflicts directly with the requirement to decrease the AFU request-to-response latency as much as possible.

8.4.3 AFU Request-to-Response Latency

A critical performance metric of the multi-stream buffer is the AFU request-to-response latency. This latency is defined as the number of clock cycles between the read port input register and the cycle the AFU receives a response with the requested data from the BRAM slice.
The implementation discussed in Chapter 7, and used in this chapter for validation, has a latency of five cycles. The read port logic requires one cycle to complete, while the BRAM slice requires four cycles. There is an input register, a control register parallel to the double-pumped BRAM primitves, and the credit sink cell which requires two cycles due to the internal FIFO.
After investigating preliminary implementation results, an additional regiser stage is required for the multiplexer as mentioned in the previous section. This directly influences the AFU request-to-response latency. Therefore in the current design, this critical latency is increased to six cycles. For smaller configurations, the additional pipeline stages can be removed, and possibly the input register of the BRAM wrapper module as well.
Depending on the workload and AFU, one might prefer a higher operating frequency versus fewer cycles in the AFU request-to-response path, or vice versa. If a large buffer is required, for example the 64 streams, 8 read port configuration, additional restrictions to the design could be applied. An example is to restrict each read port to a fixed subset of all the streams. This would work for the decompress-filter database operator for example, since it requests evenly from all streams. However, the access pattern of the merge-sort operator is random. A restriction such as this could hurt throughput.
Another possible solution is to decrease the number of channels, since this increases the AFU request-to-response latency, and increase the L1 and L2 buffer sizes. This results in fewer streams, but with more buffered data.

8.4.4 Possible Improvements

The complex paths mentioned require more attention for large configurations, in order to meet timing. However, also architectural improvements can be made. Most importantly, Section 7.1 introduced the Request Generation and Re-order Buffer modules. The Request Generation module has been validated, but not included in the validation framework yet. This is necessary to bridge to an interconnect specific interface. Similarly, the Re-order Buffer module has to be implemented and validated. Typically, responses from the host across the interconnect come back out-of-order, Without this module, either order has to be guaranteed by the host or the AFU will fail to produce useful data. Based on the fact that interconnects often operate out-of-order, an interesting architecture to pursue is a full out-of-order design and thus omitting the Re-order Buffer module.
Another performance metric left untouched is if address translation misses occur. In order to hit the translation cache in the host, host requests for the same stream could be grouped together, instead of using the Round-Robin scheduler. The Round-Robin scheduler could also be modified to, after chosing the next winner, not move away from this request producer but instead see if a second request is made.

Chapter 9

Conclusions

A new class of accelerator interfaces has significant implications on system architecture. An order of magnitude more bandwidth forces us to reconsider FPGA design. Naively scaling the interconnect will become a bottleneck due to the traditional IO model, but also because traditional solutions are unfit. New standards are required to provide a shared memory space with the IO, and extend the coherence domain of the host processor.

Therefore, OpenCAPI is of interest due to the coherent, high-bandwidth and low-latency interconnect it provides. Such an interconnect enables tightly coupled FPGAs in the data center. This allows for acceleration of emerging workloads and new usage models. Since very little public information about OpenCAPI is available, an overview of the interface is provided for those who want to gain a better understanding of it.

In this work, feeding such emerging FPGA accelerators is studied by generalizing across multiple common streaming-based access patterns and providing a data element granularity interface with multiple read ports, instead of a typical cache line granularity interface. In order to fully utilize the available bandwidth, multiple streams are required. Buffering cache lines under OpenCAPI assumptions requires re-evaluation of traditional solutions and approaches. The proposed architecture exploits different memory primitives available on the latest generation of Xilinx FPGAs. By combining a traditional multi-read port approach for data duplication with a second level of buffering, a hierarchy typically found in caches, an architecture is proposed which can supply data from 64 streams to eight read ports without any access pattern restrictions.

A correct-by-construction design methodology was used to simplify the validation of the design and to speedup the implementation phase. At the same time, the design methodology is documented and examples are provided for ease of adoption. With the design methodology, the proposed architecture has been implemented and is accompanied by a validation framework.

The Vivado toolchain was used for synthesis and implementation using the out-of-context mode. Various configurations of the multi-stream buffer have been tested. Configurations up to 64 streams with four read ports meet timing with an AFU request-to-response latency of five cycles. The largest configuration with 64 streams and eight read ports fails timing.
Limiting factors are the inherent architecture of FPGAs, where memories are physically located in specific columns. This makes extracting data complex, especially at the target frequencies of 200 MHz and 400 MHz. Wires are scattered across the FPGA and wire delay becomes dominant.

FPGA design at increasing bandwidths requires new design approaches. Synthesis results are no guarantee for the implemented design, and depending on the design size, could indicate a very optimistic operating frequency. Therefore, designing accelerators to keep up with an order of magnitude more bandwidth compared to the current state-of-the-art is complex, and requires carefully thought out accelerator cores, combined with an interface capable of feeding it.

Bibliography

[1] "Intel® Core™ i7-800 and i5-700 Desktop Processor Series," Intel Corporation, Jul. 2010, accessed 2017-07-01.

[2] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, "CAPI: A Coherent Accelerator Processor Interface," IBM Journal of Research and Development, vol. 59, pp. 7:1–7:7, Jan. 2015.

[3] R. Solomon, "An Introduction to CCIX," accessed 2017-10-15. [Online]. Available: https://www.synopsys.com/designware-ip/technical-bulletin/introduction-ccix-2017q3.html

[4] B. Thompto, "POWER9 Processor for the Cognitive Era," Presentation, 2016, accessed 2017-05-09. [Online]. Available: https://www.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.921-.POWER9-Thompto-IBM-final.pdf

[5] "OpenCAPI™ Exhibit SC17," Presentation, OpenCAPI Consortium, 2017, accessed 2018-01-21. [Online]. Available: http://opencapi.org/wp-content/uploads/2016/09/OpenCAPI-Exhibit-SC17.pdf

[6] OpenCAPI 3.1 Transaction Layer Specification, OpenCAPI Consortium Std. Version 1.0, January 27, 2017.

[7] J. Stuecheli, "OpenCAPI™ - A New Standard for High Performance Memory, Acceleration and Networks," Presentation, April 10 2017, accessed 2017-05-24. [Online]. Available: http://opencapi.org/wp-content/uploads/2017/04/OpenCAPI-4-10-2017-Jeff-Stuecheli-Preso.pptx

[8] B. Ronak and S. A. Fahmy, "Mapping for Maximum Performance on FPGA DSP Blocks," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems), vol. 35, pp. 573 – 585, april 2016.

[9] UltraScale Architecture Configurable Logic Block, UG574 v1.5, Xilinx, 2017.

[10] Xilinx. (2017) UltraScale+ FPGAs Product Tables and Product Selection Guide XMP103 v1.11. [Online]. Available: https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf

[11] M. Brobbel. (2016) capi-streaming-framework. [Online]. Available: https://github.com/mbrobbel/capi-streaming-framework

[12] M. Brobbel, "Open source AFU framework for streaming applications with CAPI connected FPGAs," Presentation, October 7, 2015, accessed 2017-02-10. [Online]. Available: http://slides.com/mbrobbel/capi-streaming-framework#/

[13] (December 21, 2017) SNAP Framework Hardware and Software. OpenPOWER Foundation. [Online]. Available: https://github.com/open-power/snap

[14] Bruce Wile, "The CAPI SNAP Framework Deep Dive," Presentation, October 28 2016, accessed 2017-03-20. [Online]. Available: https://openpowerfoundation.org/wp-content/uploads/2016/11/BWile-The-CAPI-SNAP-Framework-Deep-Dive.pdf

[15] Kangli Huang, "Multi-way Hash Join based on FPGAs," Master's thesis, Delft University of Technology, Delft, 2018.

[16] Yang Qiao, "An FPGA-based Snappy Decompressor-Filter," Master's thesis, Delft University of Technology, Delft, 2018.

[17] Xianwei Zeng, "FPGA-Based High Throughput Merge Sorter," Master's thesis, Delft University of Technology, Delft, 2018.

[18] Jian Fang, Yvo T.B. Mulder, Kangli Huang, Yang Qiao, Xianwei Zeng, H. Peter Hofstee, Jinho Lee, and Jan Hidders, "Adopting OpenCAPI for High Bandwidth Database Accelerators," in Third International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC'17), Denver, CO, Nov. 2017.

[19] J. Stuecheli, "POWER9," Presentation, 2016, accessed 2017-03-15. [Online]. Available: https://www.ibm.com/developerworks/community/wikis/form/anonymous/api/wiki/61ad9cf2-c6a3-4d2c-b779-61ff0266d32a/page/1cb956e8-4160-4bea-a956-e51490c2b920/attachment/56cea2a9-a574-4fbb-8b2c-675432367250/media/POWER9-VUG.pdf

[20] "AMD EPYC 7000 Series Processors: Leading Performance for the Cloud Era," Advanced Micro Devices, Jun. 2017, accessed 2017-09-21. [Online]. Available: http://www.amd.com/system/files/2017-06/AMD-EPYC-Data-Sheet.pdf

[21] P. Gupta, "Xeon+FPGA Platform for the Data Center," Presentation, 2015, accessed 2017-06-21. [Online]. Available: https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf

[22] Rambus Incorporated, "The role of FPGA acceleration in the data center and beyond," Blog, September 20 2016, accessed 2017-06-23. [Online]. Available: https://www.rambus.com/blogs/the-role-of-fpga-acceleration-in-the-data-center-and-beyond-2/

[23] D. A. M. A. Greenberg, J. Hamilton and P. Patel, "The Cost of a Cloud: Research Problems in Data Center Networks," ACM SIGCOMM, vol. 39, pp. 68–73, jan 2009.

[24] M. Bohr, "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper," IEEE Solid-State Circuits Society Newsletter, vol. 12, pp. 11–13, Winter 2007.

[25] W. Gropp, "CS598 Lecture 15: Moore's Law and Dennard Scaling," Spring 2016, unpublished. [Online]. Available: http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture15.pdf

[26] G. M. Amdahl, "Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities," in AFIPS Conference Proceedings, 1967, pp. 483–485.

[27] J. L. Gustafson, "Reevaluating amdahl's law," Commun. ACM, vol. 31, no. 5, pp. 532–533, May 1988. [Online]. Available: http://doi.acm.org/10.1145/42411.42415

[28] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," Computer, vol. 41, no. 7, 2008.

[29] "Intel® Xeon Phi™ x200 Product Family," Intel Corporation, accessed 2017-10-10. [Online]. Available: https://ark.intel.com/products/series/92650/Intel-Xeon-Phi-x200-Product-Family

[30] N. P. Jouppi, C. Young, N. Patil et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit™," in 44th International Symposium on Computer Architecture (ISCA), Toronto, Canada, Jun. 2017.

[31] Merv Adrian, "Big Data," Teradata Magazine, pp. 38–42, jan 2011.

[32] David, Howard and Fallin, Chris and Gorbatov, Eugene and Hanebutte, Ulf R. and Mutlu, Onur, "Memory Power Management via Dynamic Voltage/Frequency Scaling," in Proceedings of the 8th ACM International Conference on Autonomic Computing, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/1998582.1998590

[33] M. Greenberg, "LPDDR3 and LPDDR4: How Low-Power DRAM Can Be Used in High-Bandwidth Applications," Presentation, 2013, accessed 2017-10-17. [Online]. Available: https://www.jedec.org/sites/default/files/M_Greenberg_Mobile%20Forum_May_%202013_Final.pdf

[34] F. Kruger, "CPU Bandwidth – The Worrisome 2020 Trend," Article, March 23, 2016, accessed 2017-05-03. [Online]. Available: https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend/

[35] Giefers, Heiner and Polig, Raphael and Hagleitner, Christoph, "Accelerating Arithmetic Kernels with Coherent Attached FPGA Coprocessors," in Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, ser. DATE '15. San Jose, CA, USA: EDA Consortium, 2015, pp. 1072–1077. [Online]. Available: http://dl.acm.org/citation.cfm?id=2757012.2757062

[36] Allan Cantle, "Enabling Coherent FPGA Acceleration," Presentation, March 17, 2015, accessed 2017-04-03. [Online]. Available: https://openpowerfoundation.org/wp-content/uploads/2015/03/Cantle_OPFS2015_Nallatech_031315_final.pdf

[37] Neil Parris, "AMBA 4 ACE and Hardware Cache Coherency - Top 5 Questions," Article, October 14, 2013, accessed 2017-12-28. [Online]. Available: https://community.arm.com/processors/b/blog/posts/amba-4-ace-and-hardware-cache-coherency---top-5-questions

[38] "Atomic Read Modify Write Primitives for I/O Devices," White Paper, Intel Corporation, Aug. 2008.

[39] B. Benton, "CCIX, Gen-Z, OpenCAPI: Overview and Comparison," Presentation, March 2017, accessed 2017-05-03. [Online]. Available: https://www.openfabrics.org/images/eventpresos/2017presentations/213_CCIXGen-Z_BBenton.pdf

[40] TLP Processing Hints, PCI-SIG Std., September 11, 2008.

[41] PCI Express® Base Specification, PCI-SIG Std. Revision 3.0, November 10, 2010.

[42] J. Ajanovic, "PCI Express* (PCIe*) 3.0 Accelerator Features," White Paper, Intel Corporation, Aug. 2008. [Online]. Available: https://www.intel.sg/content/dam/doc/white-paper/pci-express3-accelerator-white-paper.pdf

[43] Address Translation Services, PCI-SIG Std. Revision 1.1, January 26, 2009.

[44] Atomic Operations, PCI-SIG Std., January 15, 2008.

[45] "Understanding Performance of PCI Express Systems," White Paper, Xilinx, October 28 2014.

[46] Rostislav Lisový, Michal Sojka and Zdeněk Hanzálek, "PCI Express as a Killer of Software-based Real-Time Ethernet," in RTN'2013, The 12th International Workshop on Real-Time Networks, Paris, France, Jul. 2013.

[47] PCI Express® Base Specification, PCI-SIG Std. Revision 4.0 Version 0.3, February 19, 2014.

[48] Cayla McGinnis, "PCI-SIG® Fast Tracks Evolution to 32GT/s with PCI Express 5.0 Architecture," News Release, June 7, 2017, accessed 2018-01-04. [Online]. Available: https://www.businesswire.com/news/home/20170607005351/en/PCI-SIG®-Fast-Tracks-Evolution-32GTs-PCI-Express

[49] "Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems," White Paper, IBM, September 29 2014.

[50] Bruce Wile, "Base CAPI Performance Information," accessed 2018-01-07. [Online]. Available: https://www.ibm.com/developerworks/community/blogs/a661532e-1ec6-442f-b753-4ebb2c8f861b/entry/base_capi_performance_information?lang=en

[51] Coherent Accelerator Processor Interface User's Manual, IBM, Armonk, NY, January 29 2015, version 1.2.

[52] AMBA AXI4 to IBM CAPI Adapter, XAPP1293 v1.0, Xilinx, August 8, 2016.

[53] T. Fuchs, private communication, 2017.

[54] "OpenCAPI™ Forum SC16," Presentation, OpenCAPI Consortium, November 16, 2016, accessed 2017-06-26. [Online]. Available: http://www.opencapi.org/wp-content/uploads/2016/11/OpenCAPI-Overview-SC16-vf.pptx

[55] "CCIX Consortium," CCIX Consortium, accessed 2017-10-14. [Online]. Available: https://www.ccixconsortium.com

[56] AMBA® AXI™ and ACE™ Protocol Specification, AMBA Std. Issue D, October 28, 2011.

[57] AXI Interconnect v2.1 LogiCORE IP Product Guide, PG059, Xilinx, December 20, 2017.

[58] OpenCAPI 3.0 Data Link Layer Specification, OpenCAPI Consortium Std. Version 1.0, October 10, 2016.

[59] C. Wollbrink, private communication, 2017.

[60] UltraScale Architecture Memory Resources, UG573 v1.6, Xilinx, 2017.

[61] Xilinx. (2016) Virtex UltraScale+ Product Brief. [Online]. Available: https://www.xilinx.com/support/documentation/product-briefs/virtex-ultrascale-plus-product-brief.pdf

[62] "UltraRAM: Breakthrough Embedded Memory Integration on UltraScale+ Devices," White Paper, Xilinx, Jun. 2016. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf

[63] B. Li, Z. Fang, and R. Iyer, "Template-based Memory Access Engine for Accelerators in SoCs," in Proceedings of the 16th Asia and South Pacific Design Automation Conference, ser. ASPDAC '11. Piscataway, NJ, USA: IEEE Press, 2011, pp. 147–153. [Online]. Available: http://dl.acm.org/citation.cfm?id=1950815.1950857

[64] UltraScale Architecture DSP Slice, UG579 v1.4, Xilinx, 2017.

[65] Luca P. Carloni and Kenneth L. McMillan and Alberto L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 20, no. 9, pp. 1059–1076, 2001.

[66] A. K. Martin, private communication, 2017.

[67] Andrew. K. Martin. (November 1, 2017) dicells. [Online]. Available: https://github.com/akgmartin/dicells

[68] Y. T. B. Mulder. (January 22, 2017) multi-stream-buffer. [Online]. Available: https://github.com/ytbmulder/multi-stream-buffer

[69] "How to synthesize/implement a part of the design (out of context)?" accessed 2018-01-20. [Online]. Available: https://forums.xilinx.com/t5/Implementation/How-to-synthesize-implement-a-part-of-the-design-out-of-context/m-p/823023