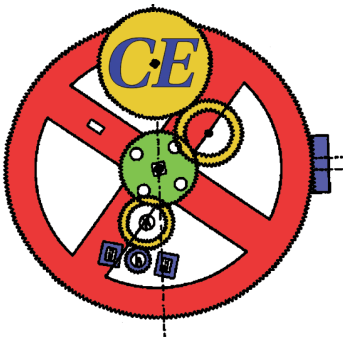


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2016

MSc THESIS

Accelerating Software Pipelines Using Streaming Caches

Koray Yanik

CE-MS-2016-17

Abstract

The trend of increasing performance by parallelism is followed by the adoption of heterogeneous systems. In order to allow more fine-tuned balancing between used thread- and instruction level parallelism, the heterogeneous ρ-VEX platform was developed.
Pipelining has been a part of microprocessor development for decades to increase throughput of a data-path, where a task is split in stages which are distributed over several functional units who work in parallel. In software the concept of pipelines does exist, but mostly speaks about data-flows as here stages do not operate in parallel. This thesis proposes a step towards making this a possibility by mapping software pipelining on heterogeneous multi-core systems.
This work documents the design, implementation and verification of a hybrid write-back and streaming cache scheme that aims to cut down overhead of inter-context and inter-core data communication, with the idea of allowing software pipelines to map stages over cores in the same microprocessor with different functional units, in order to fine-tune this mapping.
A prototype design is first implemented in a high level behavioral simulator, after which it is implemented in VHDL, tested functionally to conform to a test-suite and a set of testing pipelines developed for this project separately. The VHDL design is implemented on the ML605 Virtex-6 platform, and in its current state conforms to all test-cases but not yet the pipelines, and a slight slow-down is measured in practice.
Even though the prototype currently increased the run-time of a customly developed benchmarking pipeline from $3.3928 * 10^{-4}$ seconds to $3.7858 * 10^{-4}$ seconds, there is room for improvement and it enables more research in a new direction of transparently core-to-stage mapped software pipelines, which we define as horizontal software-pipelining, as opposed to traditional software pipelines who still execute code sequentially, hence vertically.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Accelerating Software Pipelines Using Streaming Caches

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Koray Yanik
born in Nijmegen, Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Accelerating Software Pipelines Using Streaming Caches

by Koray Yanik

Abstract

The trend of increasing performance by parallelism is followed by the adoption of heterogeneous systems. In order to allow more fine-tuned balancing between used thread- and instruction level parallelism, the heterogeneous ρ-VEX platform was developed.

Pipelining has been a part of microprocessor development for decades to increase throughput of a data-path, where a task is split in stages which are distributed over several functional units who work in parallel. In software the concept of pipelines does exist, but mostly speaks about data-flows as here stages do not operate in parallel. This thesis proposes a step towards making this a possibility by mapping software pipelining on heterogeneous multi-core systems.

This work documents the design, implementation and verification of a hybrid write-back and streaming cache scheme that aims to cut down overhead of inter-context and inter-core data communication, with the idea of allowing software pipelines to map stages over cores in the same microprocessor with different functional units, in order to fine-tune this mapping.

A prototype design is first implemented in a high level behavioral simulator, after which it is implemented in VHDL, tested functionally to conform to a test-suite and a set of testing pipelines developed for this project separately. The VHDL design is implemented on the ML605 Virtex-6 platform, and in its current state conforms to all test-cases but not yet the pipelines, and a slight slow-down is measured in practice.

Even though the prototype currently increased the run-time of a customly developed benchmarking pipeline from $3.3928 * 10^{-4}$ seconds to $3.7858 * 10^{-4}$ seconds, there is room for improvement and it enables more research in a new direction of transparently core-to-stage mapped software pipelines, which we define as horizontal software-pipelining, as opposed to traditional software pipelines who still execute code sequentially, hence vertically.

Laboratory : Computer Engineering
Codenumber : CE-MS-2016-17

Committee Members :

Advisor:	Stephan Wong, CE, TU Delft
Chairperson:	Stephan Wong, CE, TU Delft
Member:	Arjan van Genderen, CE, TU Delft
Member:	Johan Pouwelse, DS, TU Delft

Dedicated to Dear

Contents

List of Figures

List of Tables

List of Acronyms

AHB AMBA High-Performance Bus

AMBA Advanced Microcontroller Bus Architecture

ASIC Application Specific Integrated Circuit

BRAM Block RAM

CPU Central Processing Unit

DSP Digital Signal Processing

EDIF Electronic Design Interchange Format

FPGA Field Programmable Gate Array

GPP General-Purpose Processor

GPU Graphics Processing Unit

HDL Hardware Description Language

ILP Instruction Level Parallelism

IP Intellectual Property

ISA Instruction Set Architecture

MMU Memory Management Unit

PLA Programmable Logic Array

RAM Random Access Memory

RISC Reduced Instruction Set Computer

RTL Register Transfer Level

SIMD Single Instruction Multiple Data

SoC System on Chip

TLP Thread Level Parallelism

UART Universal Asynchronous Receiver/Transmitter

VEX VLIW Example

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

VLIW Very Long Instruction Word

VLSI Very Large Scale Integration

Acknowledgements

I would like to take a moment to thank everyone who has supported me in the last year, while writing the work that is before you now.

My first thanks goes out to Stephan for advising me, but also for working with a chaotic student like me who wants to do too much in too little time. Thank you for bearing with me, as I know I am not always an easy student to work with. Thank you to the entire committee as well, for reading this big pile and all the additional effort you are putting into it.

I would like to thank Joost for helping me with good ideas but also the simulator and other software aspects. I would like to thank Jeroen for providing me with more good ideas, and helping me with the ρ-VEX internals, as well as being the VHDL oracle.

Many thanks to my friends and colleagues in the MSc room: thanks to you guys it was a great place for general discussions and advice, but most of all it just was a fun place to work.

Also I would like to thank my family for their many forms of support. It has been a long road, but I finally made it.

Last but certainly not least, I would like to thank Dear. Even though you might not know it, I value everything you did to help me. You were my motivation and my focus during the long days!

Koray Yanik
Delft, The Netherlands
December 6, 2016

1 Introduction

Imaging applications have become commonly available for mobile and embedded devices. Typically, an image processing application has a very high bandwidth requirement and as such gave rise to a special kind of processor that focusses on bandwidth instead of latency: the streaming processor[2][3][4].

This processor is often seen in a Graphics Processing Unit (GPU), are optimized for regular and large applications, and exploit massive parallelism in combination with high bandwidth memory to gain maximum performance[5]. A shift has been observed from fixed function dedicated GPUs to general purpose programmable GPUs when performance increases allowed flexibility to become a second design goal[6]. This is partially the case because of the desire to customize the way the graphics were produced, but also because high-throughput, often pipelined work-flows are not just applicable in computer graphics, but also in other fields (with similar data profiles) such as image processing or physics simulation.

The concept of pipelining is used for a long time to speed-up throughput of datapaths, by separating a longer process into smaller stages that can be processed independently. The time for a data-item to enter and leave the pipeline remains the same (referred to as the latency), the interval at which data items can enter and leave the pipeline is now bounded by the stage length instead (the throughput). A pipeline is easily visualized via the analogy to a production pipeline at a car factory.

Pipelining is also used in software contexts, but here it is mostly used to describe a data-flow and operations that transform data as stages, since they are still executed sequentially; the mapping to parallelized workers is not directly possible on a General-Purpose Processor (GPP). Software pipeline applications are also called streaming applications often. One could map stages to different computational cores, but that raises two objections: inter-core data communication is expensive, and what are we to gain from such a mapping?

We propose an alternative approach to speed-up high bandwidth, often pipelined, applications on general purpose processors. Traditionally, streaming processors operate in a Single Instruction Multiple Data (SIMD) fashion to exploit a large level of Thread Level Parallelism (TLP) where several threads all execute an instance of the entire pipeline in parallel. As pipeline stages often have different computational profiles and thus requirements, we propose mapping the pipeline on a stage per thread. This is done to target heterogeneous platforms where these thread units can be customized, such as the reconfigurable ρ-VEX architecture. The most limiting factor for this approach is the high-bandwidth data-flow between the different threads, and this is the problem that we target in this thesis.

We thus aim to improve the performance of the ρ-VEX (reconfigurable VEX) processor in software pipeline applications. To achieve this, the data caches are extended with

two features: we allow data streaming between cache blocks (in different cores or contexts), and we change the write policy to a hybrid write-back write-through schema. This all has to be implemented without penalizing the reconfigurable nature of the processor.

Data streaming in this thesis is defined as allowing a context to read data from another context. Naturally, this can decrease the miss rate drastically in applications where contexts need to share data often.

The data cache's current write policy is write-through, with a write buffer of one deep[7]. Especially in a system where several contexts are actively computing and thus writing values in parallel, this buffer will be filled up quickly, stalling contexts. We aim to cut down on this memory traffic in certain cases by allowing the user to specify a special range inside the main memory that will operate under a write-back schema instead. This raises several concerns in terms of coherency and consistency, but these will be addressed as well.

1.1 Context

GPPs have been getting faster since their initial introduction. With the race to increase the clock frequency to the limit we run into temperature problems, but we are also getting close to the limit of decreasing the transistor size without making the manufacturing yield and device reliability challenging[8][9][10]. We have already observed the shift from single-core to higher degrees of parallelism over a decade ago[11].

Next to a demand for more performance we see the same happening with energy efficiency. As designs grow smaller and more energy efficient, we have seen a rapid adoption of high performance smart-phones in the last years. Energy efficiency can also be gained by more parallelism: doing more work at the same clock tick means the rest of the system does not need to be powered for two or more clock cycles.

Unfortunately, the usage of parallelism is limited by the application. We can identify two methods of parallelism: TLP and Instruction Level Parallelism (ILP)[12].

The former is where we can give work to more than one execution unit (for example a Central Processing Unit (CPU) core). Traditionally, this is most often applied to process several data elements that are independent of each other in parallel, by executing software threads on different cores[12].

The latter is where we can issue more work to the execution unit to do in parallel. This is mostly limited by data hazards in the instruction stream: it is for example impossible operate on a register at the same time as loading its content from the memory. It is achieved dynamically at run time by super-scalar processors, or statically at compile-time by Very Long Instruction Word (VLIW) processors (where each instruction is actually a bundle containing several instructions to be issued in parallel)[13][14].

As the amount of TLP and ILP depends on the application, it is useful to be able to design the processor to match the profile. The ρ-VEX (reconfigurable VEX) processor architecture is designed to suit this goal by allowing to dynamically reconfigure (at runtime) the mapping of functional units to execution units (often these are called cores in a conventional multi-core system, but the ρ-VEX these are called contexts), in a VLIW fashion. This architecture is developed by the Computer Engineering group at the Delft University of Technology[15].

A different class of processor platforms are heterogeneous platforms[16]. These platforms contain different execution units optimized for different tasks. In this kind of platform, often a general purpose processor is augmented with special-purpose processors to speed up certain specific tasks.

The issue slots (or lanes) in the ρ-VEX do not necessarily need to be balanced: they can have different functional units, making this architecture suitable to create a heterogeneous design. If we combine this property with dynamic reconfiguration for ILP and TLP distribution, we think it is an interesting platform to map a software pipeline on.

Many computational problems in several fields like image processing or deep learning can be formulated in a pipeline fashion. Roughly speaking, a data item enters the computational pipeline, which consists of several stages, each operating on the data item to transform it into a different kind of item. Each consecutive stage operates on the output of the previous stage. The strength of pipelines is that we can issue a new item in the pipeline much quicker than it would take to finish the entire pipeline, if each stage is able to process independently from each other. A key factor in pipeline performance is however to keep them balanced, as its throughput is always defined by its slowest stage, which will act as a bottleneck.

A software pipeline or streaming application usually does not increase performance, since the stages do not translate to parallel resources; they are merely sequentially executed statements. We will refer to this as *vertical* pipelining in the remainder of this work. GPPs can have one or more computational cores, who can work in parallel, but traditionally they are the same, so the idea of pipelining does not seem logical at first: spreading the stages over cores will not change anything in terms of speed-up, while one does incur communication overhead. The straightforward way of exploiting parallelism in vertical pipelines is to start an independent pipeline on each separate core.

Now let us again imagine a streaming application, but now consider a heterogeneous platform where we can customize the parallelism and computational units of each core separately. Under this assumption, allocating stages to different cores might not be that odd.

We think the ρ-VEX platform might prove to be very cost-effective as a streaming platform, due to its dynamic reconfigurability and the flexible arrangement of its lane groups to design a heterogeneous platform that matches the data-flow of our streaming application.

1.1.1 Problem Statement

When processing software pipelines, most of the time this is done in a vertical fashion where all the pipeline stages are processed on the same core, and parallelism on multi-core systems is only achieved by mapping the entire pipeline on all cores, to allow the CPU to process more than one item at the same time. This is done because the memory hierarchy of traditional processors has a big penalty on communication between cores, but also because there would not be a direct benefit from alternative mappings since all cores in conventional processors are the same.

The ρ-VEX allows for flexible scaling between ILP and TLP, but also allows for

different functional unit mappings between contexts. Pipeline stages are not always balanced, and do not always require the same amount of functional units. An example is a standard GPU pipeline, where the early stages are in floating point (world coordinates) while later stages operate in integers (screen coordinates). Also different stages can have differing levels of ILP: this is especially the case with the programmable pipeline stages of GPUs.

This gives rise to the idea where we customize our processor in such a way that each core (or context, when considering the ρ-VEX) is optimized for a pipeline stage: in our GPU example we only add floating point units to the contexts that handle the early stages, but we can also assign lanes to stages according to their ILP (and even decide to merge lanes).

To make this idea feasible, the penalty of communication between cores has to be addressed. This is expressed in the following research question.

How can streaming applications be efficiently executed on heterogeneous multi-core platforms, for example the ρ-VEX?

A systematic approach will be applied in order to find an answer to this question. We will discuss our methodology next.

1.1.2 Methodology

To answer this question, a few things need to be clarified first. In this section we will list all the steps that we will perform in the course of this thesis, or in other words, our methodology, which will lead us to an answer to the research question.

- The first step will naturally be to study related concepts to build the fundamentals on which this thesis is written. A literature study is done first, where we focus on computational pipelines and memory architectures, in order to understand better the reason(s) and/or problems why pipelines are not executed horizontally.
- An abstract design of the concept that we wish to use to solve this problem needs to be formed.
- A simulation model needs to be developed to evaluate the concept before an actual implementation is created.
- Once the model proves to be promising, an actual implementation has to be designed to target the ρ-VEX.
- After implementing, it naturally has to be verified, not only for correctness but also for costs and gains.
- This will be done by a behavioral simulation of the implemented hardware design (model), but also by testing the implementation when mapped on an Field Programmable Gate Array (FPGA).
- After we have experimental results, we can form a conclusion and hopefully answer the research question.

1.2 Overview

Chapter 2 will go into more detail about the background information required to understand the remainder of the thesis. A reader that is well knowledgeable in relevant fields of caches, memory architectures and hardware design might want to rely on it as a reference, or skip this chapter altogether.

In Chapter 3, we will design (in high level concepts) our solution or mechanism that we intend to use to gain a speed-up. High level design decisions will be made here, which will apply to both the behavioral simulator and the hardware implementation.

Chapter 4 shall discuss the actual implementation work and attempts to justify all design decisions made here. It starts of by introducing the interface that will be exposed to the user in order to use the newly added mechanisms, continues by creating a functional model to verify the idea and then the design for the hardware implementation will be formed.

After that, Chapter 5 does experimental verification to confirm everything works as expected and attempts to measure both costs and gains, in order to quantify and justify the results.

In the end we will arrive at Chapter 6, which shall give a summary that ends with a conclusion. It will also list the contributions made by this thesis, and gives some recommendations for future work while also justifies why they didn't fit in this thesis.

2 Background

In the previous chapter we have shown our problem statement and discussed the context of this work. At this point we are not yet able to propose a solution however, as we still lack the required theoretical foundation. In this chapter we will discuss all the necessary topics and background information that are required as a base for our design and implementation phases.

In Section 2.1 we will discuss, in broad terms, typical memory hierarchies of both single- and multi-core processor architectures, and the role of caches. Following that, in Section 2.2 and Section 2.3, we will zoom in on caches more, discussing several classes of caches and multi-core cache concepts, respectively.

After discussing memory related topics we will turn our attention to how we can use parallelism to gain more performance in Section 2.6. Next, we can discuss our target architecture, the ρ-VEX, and how its dynamic reconfigurability can help us to exploit parallelism in a flexible way, in Section 2.7. A key component that we discuss in more detail is again the cache.

As we intend to design a model to functionally test the system, we will discuss hardware simulation and emulation as well as the existing ρ-VEX simulator sim-rvex in Section 2.8. The actual implementation needs to be tested, verified and its properties measured as well. The used technology for this will be a Field Programmable Gate Array (FPGA), as explained in Section 2.9.

We shall discuss software pipelines in Section 2.10 and some particular example cases so we know a bit more about our actual target.

Finally, Section 2.11 will conclude the chapter with a small summary to place everything in context.

2.1 Memory Hierarchy

Memories have always been an important bottleneck in computer systems: the difference between improvement rates of microprocessor speed and memory access speed has been growing so large that we can speak of a memory wall, which is even regarded as something obvious[17].

A typical processor is much faster than its memory, so memory performance is critical. Next to speed, size and cost have been important factors too. Unfortunately, making a memory bigger negatively impacts its speed, and vice versa[18]. To alleviate that problem, computer architectures have used memory hierarchies consisting of several levels of memory. We will limit us to the part of this hierarchy inside the processor core, as this is the only part that is relevant to this thesis. Anything outside the Central Processing Unit (CPU) core will be regarded as main memory, which contains all the data.

2.1.1 Caches

Computer programs typically exert two properties of locality: spatial and temporal locality[4]. The former simply expresses that when a certain data item or instruction (basically, an address) is accessed, it is likely that data items or instructions around it will be referenced next. This stems from the fact that programs are written sequentially, and most of the time branches are local too. Data is also commonly stored in a structured way such that related data is close to each other. Temporal locality expresses that the same instruction or data item is also very likely to be re-used. This is simply because typical programs execute loops most of the time.

This property is exploited by the cache of a processor system. A cache is simply a (typically small, fast) memory that holds a copy of another memory for quicker storage[19]. The idea is that it is faster but smaller than the higher level of memory in the hierarchy, and thus it acts as a small buffer.

A cache can try to predict what memory locations are going to be used soon, but simply keeping memory locations that are requested (and maybe a few more around it) inside a cache can already be a good way to improve performance, due to the principles of locality.

To gain an effective trade-off between memory size and access time, high performance systems typically have more than one level of cache [19] (typically the cache layer that is n layers away from the CPU is called the Ln cache), where caches close to the CPU are faster and smaller. Each layer of cache acts as a buffer that stores a small part of the higher memory.

This hierarchy is not only limited to the CPU core, as effectively the main memory of desktop computers (typically Random Access Memory (RAM)) acts as a cache for their storage (disks).

2.1.2 Multi-Core Systems

Until now we have discussed only single-core systems. Caches generally do not have enough bandwidth for more than one processor[19], and arbitration would slow down accesses on the cache, which is one of the most critical systems for performance. Each processor thus requires its own private cache for most optimal performance. In a multi-level cache, this arbitration can be more acceptable in higher levels, making shared caches possible.

Unfortunately, private caches introduce a new problem called coherency. Different processors should have the same view of all data in their cache, so one way or another a write by a processor in its own cache should be reflected in all other caches that hold the same cache line. This problem is made more manageable by shared caches, as it limits the problem to a smaller set of private caches (below it). This gives rise to the concept of having hierarchical caches[20].

A simple example with four computational cores and three levels of caches is shown in Figure 2.1. Often caches are identified by a number Ln, where n is the distance from the processor core. The organization of the cache hierarchy and how to merge shared caches is a typical trade-off problem. The more sub-caches a higher cache is shared with, the more arbitration contention it might get. There is also less space available for each

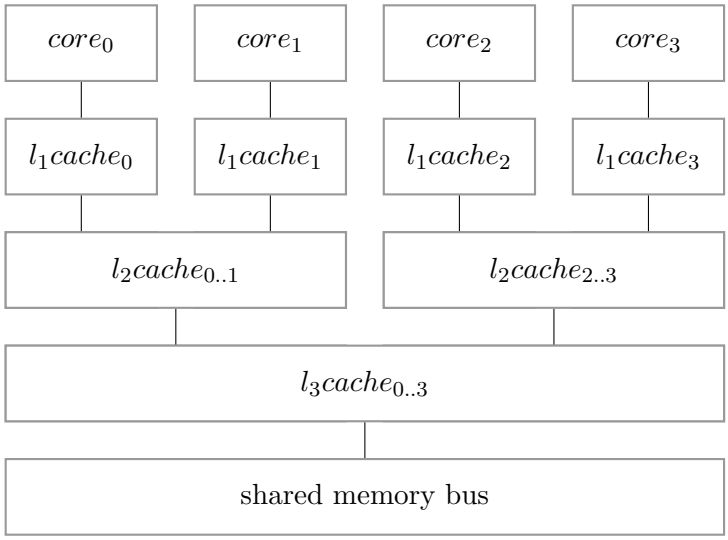


Figure 2.1: An example multi-core memory hierarchy with three layers of shared cache.

sub-cache (as they are not always interested in the same data). On the other hand, having less sub-caches in a shared cache increases the required amount of layers, which might in turn increase latency, and we have more coherency issues.

The Intel Core micro-architecture has a private L1 cache per core and a shared L2 cache, while the Intel Sandy Bridge micro-architecture (second generation Core i series) and newer have private L1 and L2 caches per core, while the L3 cache (referred to as LCC cache in Sandy Bridge) is shared over all cores[21]. Having an additional higher, larger cache will probably increase the miss penalty (as an additional level needs to be traversed) but it might increase the chance that a line is still found inside the cache (in this bigger layer), which has a larger transfer time but smaller than a miss would inflict.

2.2 Caches

Caches are very important for overall performance as they help to alleviate performance penalties imposed by memory accesses, which in turn are very slow. Caches can however be implemented in several ways, and as we can see these implementations can differ in performance and complexity. A cache can generally be grouped using three properties: the schema it uses to map full addresses to cache addresses (also called its associativity), its replacement policy and the schema it uses to handle writes to the main memory[12].

2.2.1 Addressing Schema

By definition, a cache is smaller than the memory it is caching. This means the cache is not an injective function: more than one memory address maps to the same cache location. When we read an address (from the complete addressing range, thus let us define this as m_a) from a cache, we thus need to determine first if the cache contains this item or not (called a cache hit or miss respectively). For this purpose, the address is

separated in two parts: the tag and offset. The offset is used to determine the location inside the cache where the data item will be stored, while the tag is stored so that we can reconstruct the original address and thus identify the cache item. Equations to calculate these two components are shown in Equation 2.1. As we initially start up the system with random data in the caches, there should also be a notion of if the data item is valid or not. This is most often stored in a separate bit.

$$offset(m_a) = m_a \bmod |c| \tag{2.1a}$$

$$tag(m_a) = \frac{m_a}{|c|} \tag{2.1b}$$

The following addressing schema's are defined[4]:

- Directly mapped: each address is directly mapped to one cache location. This is the most simple schema to implement, as we simply store each item at its offset, and a read hit is registered when the stored tag is the same as the requested address' tag bits (thus we need a single tag comparator).
- n-way set associative: here each memory location can be stored in n different cache locations. In other words, we have n smaller caches (sets) to consider for each address. This means we need n comparators in parallel, but also that each requires $\log n$ less bits for the tag, as this is already implied by the set that services the access. Popular choices for n are 2, 4 and 8, and we can see that a directly mapped cache is actually just a special case where n is 1.
- Fully associative: each memory location can be placed in any location in the cache. Obviously, this requires as much cache sets as data items, and is thus only practical for small cache sizes. This is again a special case of the n-way set associative schema where n is the cache size.

The second property is the replacement policy. This is a direct result of the set-associativity: remember that an n-way set associative schema allows the same location to be stored in n locations. This also means that we need to decide in what location to initially store an item when we read it from the memory (or when we write it to the cache). In most cases the cache will already be filled up, so in this case we also have to replace another item. The rules by which it does this is regarded as its replacement policy[12]. It is plain to see that the replacement policy does not apply to a directly mapped cache, since a decision between only one set is easily made. A few example policies are least recently used, most recently used or random replacement.

The last consideration is how to handle writes to the next level (a different cache or the main memory, depending on the architecture). Just keeping everything in cache without updating the next memory is a bad idea since we know that more than one address maps to the same location, so if we write to an address (and thus a cache item), it has to be updated to the next memory before it is overwritten by a different item. This schema is even more important in a multi-core set-up, for reasons that we will discuss in Section 2.3.

Here there are two commonly used schema's: write-through or write-deferred[22], also called write-back[4]. The first always updates the main memory when the processor writes to an item in the cache. This is straightforward to implement, but can lead to unnecessary data traffic if the program would write to the same memory location more often. Write-back is a more just-in-time approach, where data items are only updated to the next memory just before they are replaced. This requires a little bit more work to implement, as we need to keep track of if an item has been written to or not. Both schema's can also have a write buffer to avoid stalling the CPU when the cache is performing a write: instead it will only stall the CPU when its current command requires a write to the next memory while the buffer is already full.

2.3 Multi-core Cache Concepts

Up until now we have only considered caches in a single-core environment. In multi-core systems, it is impossible to serve all cores with the same shared cache, simply because caches have a limited number of bandwidth[19], so we would have to arbitrate accesses which would kill performance.

This means that we have different caches per core, and that in turn means that the same item can be in more than one cache at the same time. Now imagine the situation where one core writes to a memory location. This updates a corresponding cache item, and depending on the write policy, may or may not update higher caches or the main memory. The update should however also be done to all other caches that have this item cached. This is the problem of cache coherency[23].

2.3.1 Bus Snooping in Write-Through Memory

A solution to the coherency problem, in the case of a write-through policy, is the write-once strategy defined by [24]. This strategy is also often referred to as a write-invalidate snoopy cache protocol[22]. While this strategy also incorporates the notion of reserved data (which has ben written in exactly one cache while still consistent with the main memory), we can use the idea of snoopy caches. The principle of snoopy caches is that they monitor accesses done by other caches as well. This can be used fairly straightforward to implement coherency in write-through caches: just invalidate a cache line when any other cache writes to it.

In this implementation, all caches snoop the bus: they observe any access that is done on the bus. If a write is done, they all invalidate (set the valid bit that indicated that a line was read from the next memory) the data line which holds the same address. Since the policy is write-through, we know that all writes from a CPU to the cache will always directly go to all the higher levels in the cache and to the main memory, thus all the caches will automatically invalidate the item at all times. This also means that the update will always be reflected in the main memory, and thus they can re-fetch the item on the next read. It won't directly work with a write-back solution, because the write will not be done directly thus the situation can exist where a cache contains an updated value but other caches and the main memory are unaware of this.

2.3.2 Bus Snooping in Write-Back Memory

An extension to this is performed in the Intel 64 and IA-32 architectures on write-back memory[25]. When a processor tries to access a location that is modified (valid and dirty[1]) in a different processor's cache, the second processor signals the first, offering the modified location directly to both the accessing cache and the main memory. This ensures coherency in a write-back system, while bypassing the main memory architecture as well, but requires a more complicated protocol.

2.4 Cache Example: Intel

The Intel 64 and IA-32 families offer fine grain control over how data is cached in the system[25], on a system-wide, memory range and memory page basis via the Memory Management Unit (MMU).

Caching can globally be restricted by a single bit in the CR0 control register. When caching is not restricted, write-back is the default policy, but this can be overridden either on page-by-page basis or by physical memory ranges.

As mentioned earlier the cache mode can be set locally as well, in a memory range or page basis. These features are meant to complement each other. Using memory ranges can be useful for specifying fixed I/O port or frame-buffer ranges, but the amount of ranges that we can specify is fixed and limited. On the other hand, managing cache modes on a page by page basis is useful in other situations and only extends the existing page information registers by a few bits. When both a region and a page overlap with conflicting cache modes, generally the most restrictive of the two is used. A total of six different caching modes are offered, with write-back and write-through among them.

Intel uses memory type range registers (MTRR) to assign a cache mode to physical memory ranges. To assign cache modes to pages, the Page Attribute Table (PAT) is used instead.

2.5 Cache Example: ARM

In the case of ARM processors it depends on the family. For example, the Cortex R7 only supports write-back[26], while the Cortex R5 family it is a part of the Memory Protection Unit or MPU (which is optional, so implementations without this feature exist)[27].

If the processor has an MPU, it can have either 12 or 14 memory regions. Each region is defined by a base address and size, and we can set cache policy to write-through or write-enable for each region.

For the Cortex A8, it is part of the MMU like with Intel, but it is implementation defined whether or not all cache policies are supported[28]. There are eight Memory Attribute Indirection Registers (MAIRs) available, which can define a memory type each (containing the cache policy among other attributes)[29]. Three bits inside each the translation table entry point to one of these eight MAIRs.

[1]The Intel architecture uses a different protocol for coherency however, called MESI, but this is beyond the scope of this project.

2.6 Parallelism

Parallelism is the key to performance in current computing architectures. There are several kinds of parallelism and they can be exploited in different ways. We will focus on two forms of parallelism defined by [12]: Instruction Level Parallelism (ILP) and Thread Level Parallelism (TLP).

- ILP: Instruction Level Parallelism occurs when several instructions can be executed in parallel.

- TLP: Thread Level Parallelism is found in applications when we can split the work in several tasks that can run independently from each other (aside from an occasional synchronisation or data share).

TLP is offered in multi-core systems, where applications can run several threads on different cores. It is up to the programmer to expose and exploit this kind of parallelism, and he is also responsible for avoiding unsafe situations such as race conditions.

ILP is offered in two different ways. In one way, the compiler generates a sequential instruction stream as usual, unaware of the ILP capabilities of the processor. The instruction stream is analyzed by the processor at run-time, who can re-arrange instructions or issue them in parallel as long as it concludes that it is safe. A processor with this ability is called a superscalar.

On the other end of the spectrum is the implementation where the compiler takes care of extracting ILP in a safe manner. The processor on the other hand has multiple issue-slots in which instructions can be issued (in parallel). The compiler can thus issue an instruction stream with one or more instruction per instruction word. Processors that work like this are called Very Long Instruction Word (VLIW).

Advantages of superscalars are that the compiler can be kept more simple, applications with the same Instruction Set Architecture (ISA) can be executed with parallelism regardless of what specific processor it was compiled for, and that certain dynamic optimisations can be performed that might not be able to be done by a static, compile-time check. Of course, this comes at a greater cost, as the hardware to arrange all this is very complex and energy-consuming. Superscalars are therefore mostly seen on the desktop and high end processor markets.

The VLIW is more or less the opposite: the hardware can be very simple (only some duplication required), small and power-efficient. The compiler has to be more complicated, and more tailored towards the specific processor, so a program compiled for a different processor from the same ISA might not provide the same performance. It might also lack some optimisations that can only be done at run-time. Because of its simplicity, lower cost and better energy-efficiency, these processors dominate the embedded- and Digital Signal Processing (DSP) processor market. Because our target processor architecture is a VLIW, we will go into a bit more detail next. Superscalars are out of the scope of this thesis.

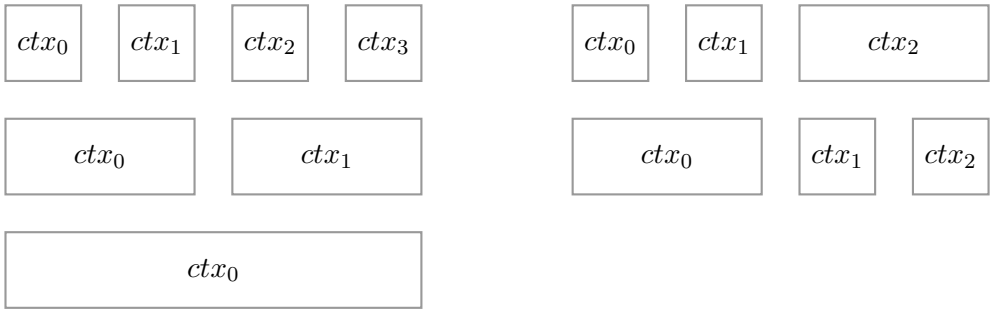


Figure 2.2: All possible configurations of an eight-way ρ-VEX core.

2.6.1 VLIW Processors

As mentioned earlier, VLIW processors can issue several instructions per instruction word. We will introduce some basic VLIW terminology next. An instruction word as issued to the complete processor is called a bundle, while each of the sub-instructions that it contains are called syllables. The amount of syllables a bundle can contain (and thus instructions that can be issued in parallel) is defined as a VLIW processor's issue width. The unit that executes a single bundle is called a lane, and thus a VLIW processor by definition contains as much lanes as is its issue width.

2.7 ρ-VEX

The ρ-VEX (Reconfigurable VLIW Example (VEX)) is an architecture based on the VEX architecture, developed by the Computer Engineering group at the Delft University of Technology. It was originally designed and implemented in [15], but the current version is created by [7].

It is based on the VEX architecture: a static compile-time configurable clustered VLIW processor architecture for simulation, analysis and evaluation purposes[30], based on the ST200 architecture[31]. One can define clusters of cores, each with a configurable number of lanes, functional units and registers. This design adds flexibility to design a core that suits an application with a specific ILP.

The ρ-VEX design is based on it, but also allows a number of contexts (comparable to cores) to be mapped to the lanes, each with its own set of registers, adding the possibility of TLP. This mapping can dynamically be reconfigured at run-time (hence reconfigurable VEX). With this, the core can dynamically re-adapt to changing ILP and TLP requirements. Note that this architecture currently does not support clusters, but platforms with multiple ρ-VEX cores have been created.

2.7.1 Dynamic Reconfigurability

As mentioned earlier, the ρ-VEX offers dynamic reconfigurability to exploit ILP and TLP in different ways. This is done by allowing the user to change the mapping of contexts to lanes at run-time. We can consider a context as a core in a conventional multi-core system.

In practice, this means we can allow a single context to use all available lanes if the program has a very high ILP, but we could split the lanes in two or more contexts if this is not the case, to enable more TLP. The default ρ-VEX configuration has an issue-width of eight (and thus also eight lanes). Lanes are coupled by two, thus we have four lane-groups available. Likewise, the default configuration has a maximum of four contexts.

Contexts are usually defined by how many lanes they are coupled with. In the default configuration we can have one eight-way context, two four-way contexts, four two-way contexts, or two two-way and one four-way contexts. All possible configurations are listed in Figure 2.2. Note that there is one possibility missing: because of technical limitations (the routing is implemented via a binary tree structure for performance) a configuration with a four-way context in the middle is forbidden.

By default the system starts up with a single context, who can in turn write a new configuration request vector to a control register. A few cycles later, any other context will be started depending on the vector.

2.7.2 Dynamic Caches

Of course, the cache of the ρ-VEX has to suit this reconfigurable design as well. In this section we will describe the cache and its dynamic routing network. The instruction cache has a similar routing network and is not relevant for this thesis, so it will be left outside of our scope.

Let us begin with the data caches itself. Each lane group has a dedicated cache block, which however has to be routed to one or more lane groups, depending on the current configuration. The cache blocks themselves operate in a write-through write-allocate fashion, and operate in a partially set-associative manner with bus-snooping as coherency mechanism. Write-through is chosen because of the choice for bus-snooping.

Each block that is coupled to the current context (and by that we mean that the lane group that owns the block is coupled to the current context) can hold an entry relevant for that context. This means the request has to be forwarded from any lane-group to all cache blocks that are coupled to its context. Likewise, the result has to be forwarded back to each lane group that was coupled to it, because it could have been issued by any lane group. Note that because the request could have been issued by any coupled lane group and can be handled by any coupled block, we can allow at most one cache access per lane group in a context. If this restriction is violated, the requests will be ignored and the caches will issue an error to the lane groups.

When writing, the bits directly following the tag are used to select a block that should handle the write. This makes the cache behave as a direct-mapped cache and thus we do not get the performance of a truly set-associative cache, but it is nevertheless done to improve locality when the cache is reconfigured.

2.7.2.1 Routing Network

As mentioned earlier the cache needs to be able to route a request from any lane group to any cache block that is coupled to the same context, and the result has to be routed from

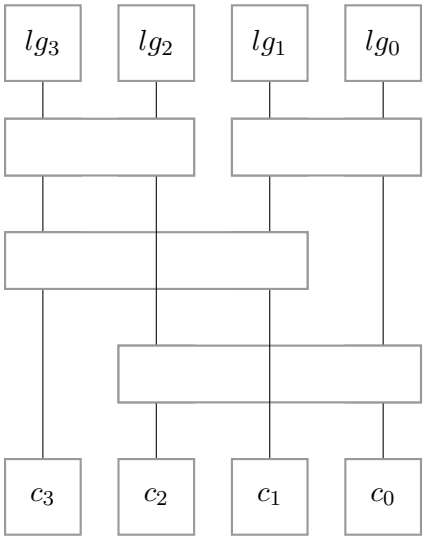


Figure 2.3: The input routing network in a ρ-VEX core with four lane groups.

any cache coupled to this lane group (since any cache could have handled the access) back to all lane groups that are coupled to the context.

To implement this, the routing network as shown in Figure 2.3 has been implemented to broadcast requests from lane-groups to cache blocks. The blocks inside this routing network exhibit behavior described in Equation 2.2. If the inputs are decoupled from each other, the inputs are directly forwarded to the outputs. Otherwise, if the inputs are coupled, a request on either of the inputs will be forwarded to both outputs. The output network is constructed in a similar fashion, but reversed. Also, the switching nodes forward hits instead.

$$(out_l, out_r) = \begin{cases} (in_l, in_l) & \text{when } decoupled(in_l, in_r) + request(in_l) \\ (in_r, in_r) & \text{when } decoupled(in_l, in_r) + request(in_r) \\ (in_l, in_r) & \text{otherwise} \end{cases} \tag{2.2}$$

2.7.2.2 Cache Blocks

The last component that requires more clarification are the cache blocks themselves. Each lane group has its own cache block, that may or may not be part of a larger group of cache blocks joined together in a context. The routing network takes care of routing commands to all involved blocks and routing back the correct response from all joined blocks to the issuing lane group, so each cache block can just handle requests that are issued indepedently.

A slightly simplified view of the overall structure of a cache block is depicted in Figure 2.4. The memory blocks have a delay of one cycle, so several registers are added to ensure all signals arrive in the controller correctly timed. The cache itself contains four key components:

- The valid block: this contains a single bit per cache line determining if the contents of the line is valid or not. This memory has two separate ports: one for reading and setting the valid bit, and one for clearing the valid bit (invalidating it). The ports are configured in this way because the main cache controller who is connected to the same read address validates lines when they are read from the memory, while a write to the memory from a different source (cache block) has to invalidate lines so this is most likely a different address. It also is able to be flushed (entirely set to 1) in a single cycle.

- The tag memory: this holds the tag for each cache line, which is compared to the tag of the currently requested line address: when they match, a hit is registered. It has two read ports: one for the normal address, and one to check if the cache line addressed by the invalidation logic is in our cache or not (a hit on the address), because we only need to invalidate it if it is the same line.

- The data memory: this block contains the actual data that forms the cache line.

- The controller: this responds on read and write misses to handle them correctly, providing signals that write or read to the memory bus and updates the correct memory blocks.

2.7.2.3 Effects of Stalling on Cache Blocks

Note that this is a slightly simplified view: the way the commands are issued to the cache blocks is slightly more complicated. The inputs to the cache are only valid when the lane group of which this cache is a part of is not stalled. The last valid input remains active while the lane group is stalling. For example, when a read is issued, one cycle later the hit signal should go up if there is a hit. If the hit signal does not go up, the main controller will start handling a miss. The lane group will stall when it registers a miss. The inputs should now remain the same.

To ensure this, all inputs to the cache are connected to registers, which are only active when the stall signal is low. This would normally delay all the inputs to the cache blocks by one cycle, which means cache accesses would take at least two cycles. To avoid this, an additional multiplexer is added that selects the original, combinatorial input when the stall signal is low, but the registered variant when the stall signal is high as shown in Figure 2.5. If one would look back to Figure 2.4, actually all inputs are handled like this. The inputs that are explicitly registered in this figure are in reality connected to the output of the register, bypassing the multiplexer.

A timing diagram for two example requests is shown in Figure 2.6, for both an immediate hit (Figure 2.6a) and a hit after four cycles of miss handling (Figure 2.6). In this figure, V denotes a valid request, while X denotes that the value is ignored and might be anything. Signals req_i, req_r and req_o denote the request issued by the lane group, the request stored in the register and the request issued to the cache, respectively.

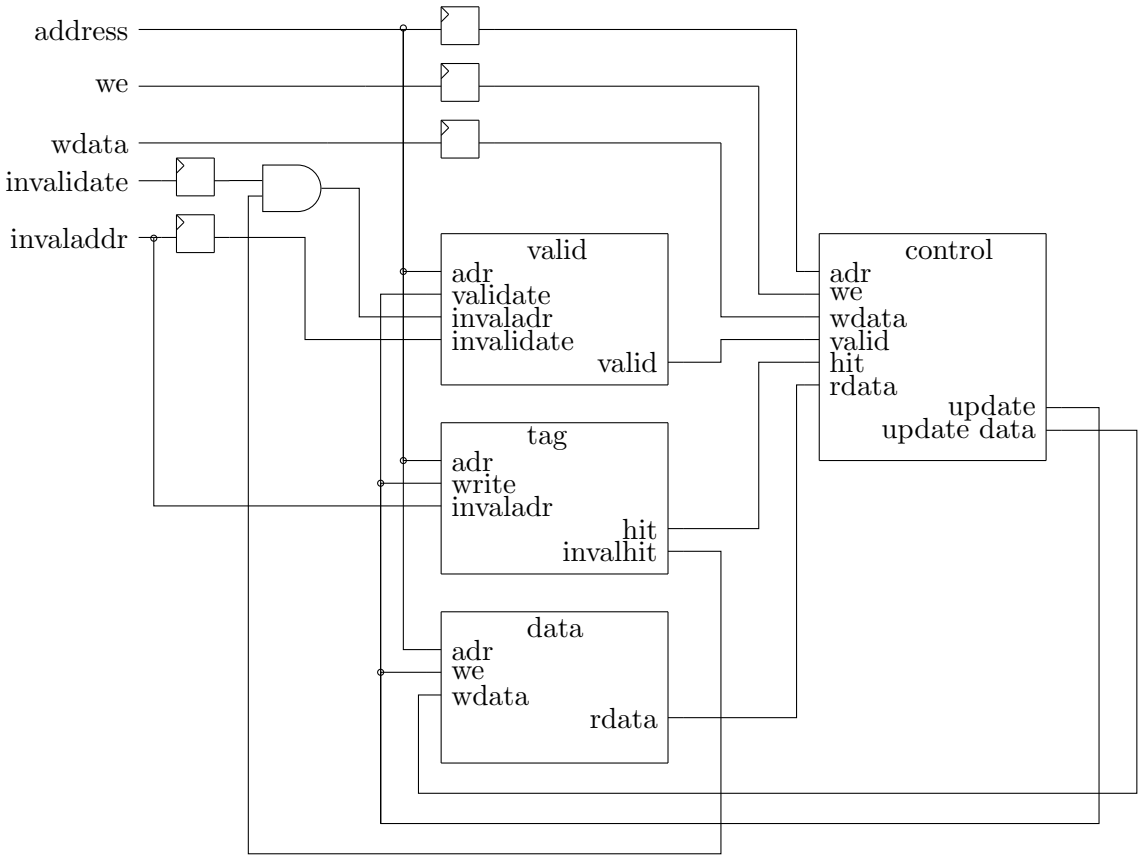


Figure 2.4: A simplified view of the cache memories and main controller inside a data cache block.

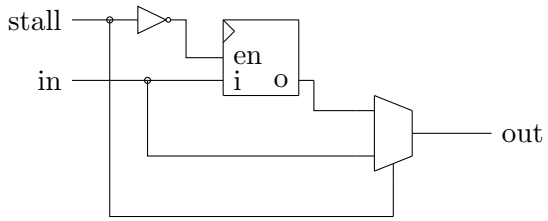
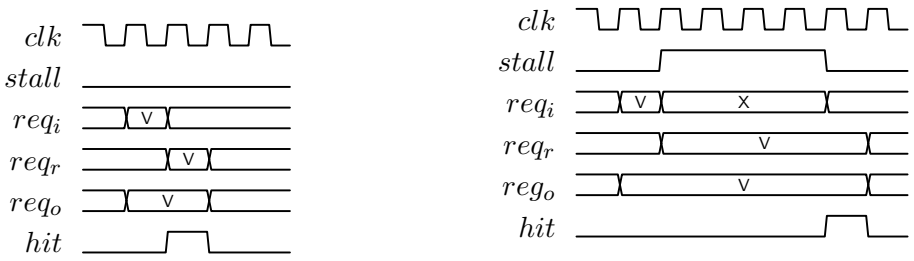


Figure 2.5: Circuit used to keep inputs valid when the lane group is stalling.

2.8 Simulation

We already mentioned that we aim to start by developing a high-level model, to experiment and test our ideas with. Simulation is a well-known and often used method to validate or test circuits and hardware designs[32]. Simulation is done by inferring how a circuit behaves when certain test-stimuli are applied to it, which is in practice done via models of the circuit.

Hardware models and abstraction levels are discussed in more depth in Section 2.9.3.



(a) An example where the cache reports a hit.

(b) An example where the cache reports a miss for four cycles.

Figure 2.6: The timing behavior of requests to the cache, in both a hit and miss example.

For now it is sufficient to know that we can simulate different models of the same circuit, that are designed at different levels of abstraction. The key problem is that at a lower level of abstraction, we have more details to consider. One one hand, this means that problems that relate to these details cannot be found at higher levels of abstraction, but on the other hand simulating is more computationally extensive. One more important issue is conciseness of the model: if the model does not match the desired design (which may or may not be caused by assumptions made over the details that are abstracted away from), a simulation will give undesired results too.

One can do a simulation of a circuit-level model, where the electrical properties of the circuit are derived via equations. This is very precise but costly, and is less viable for larger circuits. Another example is logic level simulation, where the model is simulated with logical variables, by doing assignments on clock cycles, which is more suitable for larger circuits. A higher level of abstraction is considered a functional-level simulation, where the functional behavior of the circuit is simulated. This can be in a modeling language aimed at hardware design (we will again go into more detail on those in Section 2.9.3), or even a general purpose programming language.

As we will see in Section 2.9.3, higher level models can be used to automatically generate lower level models. It is thus often the case that a simulation can be done after each generation step, to iteratively confirm the more detailed model still functions correctly. Simulation is sometimes also called emulation (this term is more common with models done in high-level programming languages).

2.8.1 sim-rvex

Two simulators of the targeted ρ-VEX platforms exist, but we will focus on only one: sim-rvex. This is a modified version of the st200 simulator, targeting the ST200 VLIW microprocessor by HP Labs and STMicroelectronics, on which the VEX architecture is also based[31]. The ST200 processor is in turn a part of the Lx architecture, which is described in [33].

This simulator is a behavioral simulator, written in the high level programming language C. It has support for the dynamic run-time reconfigurability of the ρ-VEX, and provides the full ρ-VEX instruction set. It is not perfectly cycle accurate, especially when considering data cache accesses some things are estimated. Furthermore, since it

is a behavioral model, it abstracts away from details in the cache. Only tags are stored to simulate hits and misses, so cache line simulation needs to be made more detailed for our purposes.

2.9 Hardware Prototyping

After developing a simulation model, we can move on to an actual implementation. As it is unlikely that the first design will work perfectly, a prototyping platform is required. Hardware production such as Application Specific Integrated Circuit (ASIC) fabrication have always been costly, both in time and money. It is thus not surprising that hardware prototyping platforms have been developed, to alleviate this problem. An early example of such a platform is a Programmable Logic Array (PLA), but these were only of limited use. This eventually evolved into the technology of FPGAs, which among many other uses are very suited for rapid prototyping[34][35]. In this section we will discuss the idea behind FPGAs and the basic work-flow that we will use to implement our prototype.

2.9.1 Field Programmable Gate Arrays

FPGAs are essentially programmable hardware devices, on which the logic can be changed post-fabrication[34]. They fill the gap between on one side the high performance but fixed function ASICs and on the other side the flexible but relatively slow general purpose processors. ASICs are entirely fixed at fabrication and can not be changed, while a General-Purpose Processor (GPP) has a large set of instructions that can be exploited to implement virtually any application. FPGAs fall in the middle, as their logic can be changed, making them more flexible than ASICs. As functionality is implemented in hardware, they are generally faster than GPPs, but overhead as a result of the programmability cause them to be slower than ASICs.

They also distinguish themselves from ASICs in term of cost: an ASIC has a high initial cost (mask production), but a low per-unit cost, while FPGAs have (from the user's perspective) no initial cost but a higher per-unit cost, making them more suitable for low-volume products.

They have many possible applications where the speed of a GPP would be lacking, but an ASIC is either too expensive or not flexible enough. Reasons can be for example that one wants to be able to update functionality (expanding functionality, updating a used data-set or fixing bugs) or remove it (licensing and Intellectual Property (IP) protection)[34].

2.9.2 Technology

Many different (commercial) technologies and architectures of FPGAs exist. Three critical components of FPGA architecture are the programming technology, the logic blocks and routing architecture which are discussed in depth in [36].

The programming technology decides how to make the hardware actually programmable, some examples are memories or anti-fuse technology. This has implications

such as if the design is re-programmable more than once (anti-fuse programming is permanent), if it is volatile (and thus requires to be re-programmed every time on power-up), but also the area, production process and electrical properties.

The basic building blocks on an FPGA are its logic blocks, and their granularity. Some FPGA designs use fine grain blocks, that consist of only simple transistor blocks or gates, while others use course grain blocks that are built from larger structures such as multiplexers or LUTs. Using fine grain blocks have an advantage that it is easier to use small blocks efficiently and thus to maximally utilize the available resources, but the disadvantage is that they require more wire segments and switches to be route-able, making it more costly and slower. Course grain blocks on the other hand are a bit more difficult to effectively use as they are larger. They also require more inputs to each block, requiring more effort to route.

The last factor is the routing architecture: an FPGA is basically a large amount of logic blocks, that all need to be routed together in an efficient way to express more complicated logic. The routing architecture has important as the interconnects are typically the biggest power- and area consumers on FPGA chips.

2.9.3 Modeling and Design

We described FPGAs as our target technology. However, regardless of the target, making a manual Very Large Scale Integration (VLSI) design on a physical level is not feasible. Generally a hardware model is implemented and later synthesized. We will describe this in a bit more detail in this section.

A model is essentially an abstracted description of an entity. It focuses on certain aspects while discarding irrelevant details, making it compacter and thus easier to comprehend. Many kind of hardware models exist, with different focuses. [32] classifies them globally in abstraction level and view. As abstraction levels, they define architectural (describing an architecture of operations), logic (describing the evaluation of logic functions) and geometrical (as a mapping of geometrical entities). Furthermore, they define the views as behavioral (how does the circuit behave), structural (in terms of interconnected components) and physical (its physical implementation).

The idea behind synthesis is that we can iteratively generate a more detailed model by adding additional required information[32] (often referred to as constraints, as they make the model less general, and thus constraint it). Examples are architectural- and logic level synthesis, where structural views are generated from architectural and logic level models respectively: both bind hardware resources to either the architectural operations or the logic functions that define the model (and thus it requires more information about the available hardware resources).

The earlier definition of hardware models are fairly broad: one can design visual models (for example logic gate schematics, data-flow graphs or state machines), but they tend to grow large quickly. A more commonly used model in hardware development is a Hardware Description Language (HDL) due to its conciseness[32].

HDLs generally support both behavioral and structural views, but also architectural and logic level models. A common design flow is that the designer writes a behavioral or structural model (or a combination of both), and both architectural and logical synthesis

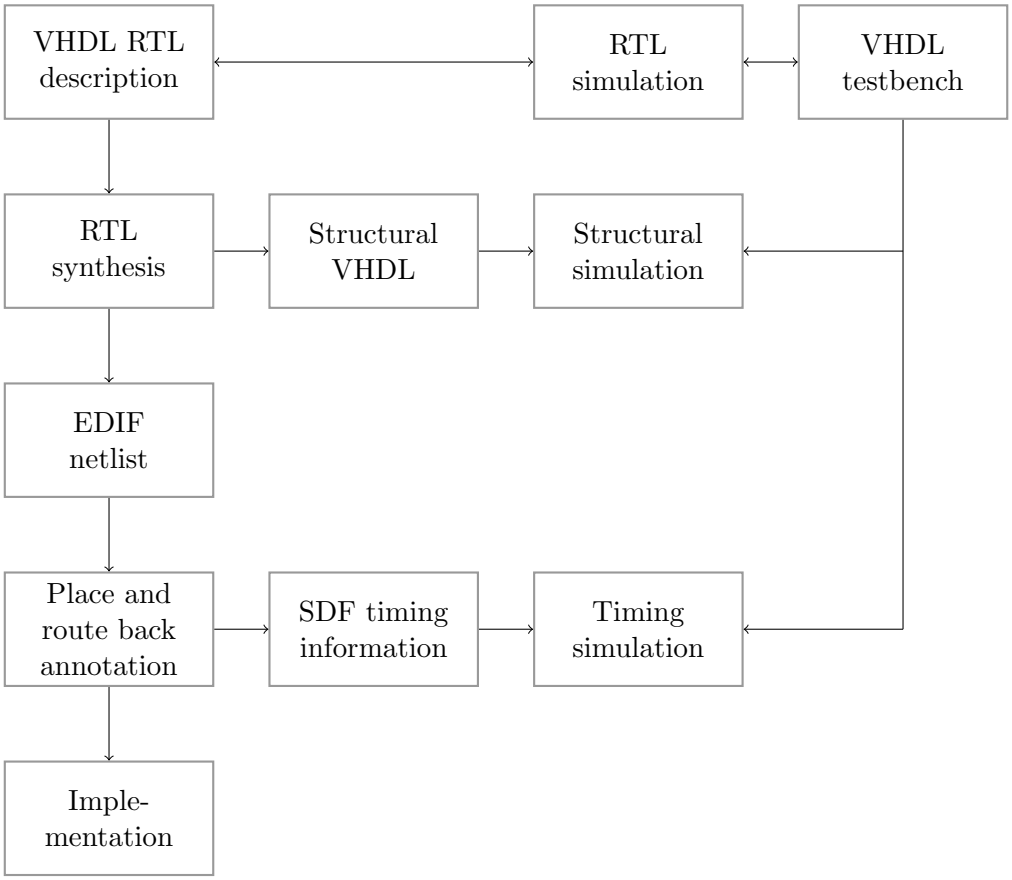


Figure 2.7: RTL synthesis design flow, taken from [1]

tools are used to generate a fully structural, logical model purely describing interconnects of basic resources, referred to as an Register Transfer Level (RTL) model or net-list. This final model is then implemented by a final synthesis tool targeting the desired technology, creating a physical implementation. The latter tool is most likely provided by the FPGA vendor, but often all these synthesis steps are supported by the same tool.

An example RTL design flow is described in [1], and shown in Figure 2.7. In this case VHDL (VHSIC Hardware Description Language) was used as HDL while Electronic Design Interchange Format (EDIF) was used as net-list representation format, but the design flow applies to any HDL or format. Likewise, it describes a synthesis flow starting with an RTL description and RTL simulation, but these steps can also be performed on a behavioral model.

One starts off with a behavioral or RTL model, and a testbench, which is a top-level model used to test the design (referred to as a unit under test) by feeding test stimuli to its inputs. After a behavioral simulation verifies correctness of the model at this level of detail, a structural model can be generated and simulated again in the testbench. Again after simulation proves this level of abstraction still behaves correctly, one generates a netlist and performs placement and routing; which generates timing information. The new model augmented with timing information can once again be simulated, after which

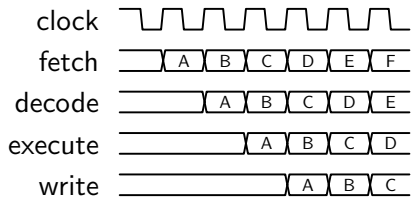


Figure 2.8: A pipelineing example where several instructions are fed through a four-stage pipeline.

the final implementation can be generated.

2.10 Pipelines

Pipelineing has been used to speed up processors since the shift to Reduced Instruction Set Computer (RISC) architectures in the 80s[37], however it is not limited to hardware. When a larger process can be cut in chunks (called stages) that can be processed in parallel, the throughput of the system is no longer determined by the long process but by that of the smaller stages.

It is best explained via the analogy of an car-assembly pipeline, where a large number of workers each add one component to the car that passes along on the pipeline. It might take a day to assemble a new car, but at the end of the pipeline a new car is delivered at a much quicker interval.

This is the difference between the latency and throughput of a system: the latency is the time it takes for a certain car to enter and leave the pipeline, while the throughput is defined by how many cars leave the factory. While the latency remains the same, the throughput is now determined by the slowest stage in the pipeline (which serves as a bottleneck). It is thus important that a pipeline should be as balanced as possible.

2.10.1 Hardware Pipelines

Pipelines are used in computational data-paths extensively, for example the data-path of a (micro)processor. Some common stages in processor data-paths are instruction fetch, instruction decode, instruction execute and write-back.

Firstly, the instruction is fetched from the instruction cache or main memory. Next, this instruction is decoded to generate control signals for the rest of the processor. In the following stage, these control signals are used to actually perform the instruction. Some processors fetch the operands for the instruction in this cycle, while others introduce an additional stage for this before the execute step, if the path through the registers and functional units is too long. In the last stage, the result is saved.

An example where several instructions are scheduled is shown in Figure 2.8. We can see in this example that instruction A finishes four cycles after it is issued, which implies a latency of four cycles, corresponding to the number of stages. However, after this initial delay where the pipeline is still being filled, a new instruction finishes every cycle.

2.10.1.1 Hazards

Pipelining is not without danger, as there are situations where we cannot issue certain instructions immediately after each other. These situations are called hazards. [4] defines three classes of hazards:

- Structural hazards: the hardware does not support a combination of instructions that are planned to execute at the same time. An example would be a system with no separated instruction and data caches: the instruction fetch and data execute phases might both need to be able to read at the same time. The pipeline and architecture of a processor needs to be designed in such a way that these hazards cannot exist.
- Data hazards: this happens when an instruction relies on data that is still being computed in the pipeline. For example, if one would perform two additions immediately after each other, the second addition would need to wait until the former is ready writing back the result before it can fetch its operands. A solution to this, which is called bypassing or forwarding, is by keeping the value in the pipeline and effectively forwarding it to the next stage directly.
- Control hazards: this happens every time when a conditional branching or control operation is executed. It cannot fetch the next instruction, since at this point in time it is not yet known what th next instruction will be. There are many solutions to this, for example branch prediction.

2.10.2 Software Pipelines

Pipelines are not limited to assembly lines and processors (or other hardware with datapaths). They can be used in software too, even though the final result should map on hardware to some extent to gain any performance gain from it.

Splitting a large function in several sub-methods that each operate on a data item can be considered a pipeline, but if this code is executed on a conventional, single-core processor, the stages are all executed sequentially and from a performance perspective nothing is gained.

Nevertheless, pipeline-like software does exist, even without performance gain, since some applications map to a pipeline quite naturally. Consider for example image processing: often several steps are performed on an image in a certain order. A traditional example is edge detection, which is often preceded by a noise reduction operation since edge detection is very noise sensitive by nature.

In some cases, a software pipeline actually maps onto different computational resources and thus a throughput gain can be achieved. This is the case when programming Graphics Processing Unit (GPU)s, in both graphical or general purpose computing contexts. Since GPUs evolved from 3D graphics rendering pipelines, where the stages used to be fixed function but slowly became programmable, computational pipelines map quite naturally on them. We will discuss the classic fixed function 3D graphics pipeline as defined in OpenGL 1 in the following section, focusing on the pipeline stages while not discussing the nature of the operations too much, as this is not really relevant.

2.10.2.1 Graphics Pipeline

OpenGL was designed as a software interface to provide standardized access to GPUs[38]. Version 1 describes a pipeline that mapped naturally on GPUs of that time, with fixed function stages. In later versions of the standard made (most of) the stages programmable, firstly to allow more customization of how the graphics would look in the end, but eventually evolving them towards highly parallel general purpose processors.

The user supplies commands to a buffer called the display list, which are retrieved and processed in order by the pipeline. More precisely, the user provides several high-level state changing functions or issues vertices (four-dimensional vectors, comparable with coordinates in a projection). These vertices are interpreted in order as primitives such as triangles or lines (consuming three or two vertices, respectively).

The traditional pipeline consists of roughly four stages. The first stage approximates (more complex) geometrical primitives with other primitives (for example if one draws a square, this can be approximated with two triangles). This stage is sometimes not explicitly described.

The second stage operates on the issued vertices, where world coordinates are transformed to screen coordinates by multiplying with the model-view and projection matrices (one describes the location and rotation of the camera, the latter the perspective of the camera). Points which fall off-screen can be clipped out and discarded (this might result in more points being added when primitives are on-screen partially). This generates coordinates perpendicular to the screen, so they contain an x and y coordinate, but also a depth value.

The third stage rasterizes primitives (most often triangles by consuming three vertices per triangle) and generates several fragments: these are representations of screen-points or framebuffer coordinates.

The last stage operates on all fragments, does depth testing to discard them if they are behind something else, calculates their color by interpolation, texture look-ups or alpha testing when they are translucent. The end result is written to the frame-buffer.

2.11 Conclusion

We now have laid down the foundation on which we can build the rest of this thesis. We now know more about memory hierarchies, the cache and how both affect communication between different cores. An introduction to parallelism and how the ρ-VEX can exploit it in different cases has been given. We also know how to simulate hardware functionality as a validation model beforehand. Our target will be an FPGA design as they allow for a very cost-effective prototyping work-flow. Lastly, we discussed software pipelines as our target application.

3 Design

In this chapter we will analyze the problem of communication latency between different cores or contexts again. We will try to find a solution for it and make a high level design in this chapter.

In Section 3.1 we will very briefly elaborate on the communication latency, or penalty, that is inflicted when we share data between two computational cores. It defines them as compulsory misses in streaming data-flows. Section 3.2 tests the scalability of the streaming concept and concludes it has the unintended side-effect of increasing unnecessary write traffic. For both compulsory misses in streaming data-flows but also the increase in write traffic, we introduce a new concept to tackle it.

The former is tacked in Section 3.3, by our streaming cache design. The latter is mitigated by a hybrid between write-back and write-through cache policies in Section 3.4. In Section 3.5 we conclude with a summary of the most important parts of this chapter.

3.1 Penalty of Streaming

Recall in Section 1.1.1 where the idea of running a pipeline application horizontally was introduced, mapping states on different contexts of the ρ-VEX. The same section identified memory communication overhead as one of the reasons why this idea would not work in the current form.

In the case of the ρ-VEX, the producing context (the one that produces or writes the data) writes his data to the cache. One cycle later, the cache will in the case that the bus is free handle this write or stall the Central Processing Unit (CPU) until the bus is free. This takes several cycles. Once this is completed, the consuming context (the one who consumes or reads the data) can start its read. Since the former context just finished writing it and we know that both contexts are different (as was our assumption setup), we know that it is always a miss, a so called *compulsory miss*. This means we have to read it from the main memory, taking several cycles again. An example is shown in Figure 3.1a: here, core 0 wants to read data generated by core 1. This data has to traverse through a cache, over the memory bus to the main memory, and then can be read back to another cache and is finally available. Overall, this means that one context to context stream can take as much as two write penalties[1] and a read miss penalty.

We would like to allow the programmer in certain cases to bypass this large penalty, by giving contexts read access to caches that are owned by other cores as well. We define this as streaming caches, which will be our method of decreasing the compulsory miss-rate in streaming applications. An example is shown in Figure 3.1b, where the same scenario is depicted. However, core 0 is now able to directly read from cache C_1, cutting

[1] The real worst case is even more problematic, if one or both of these writes had been a byte or half-word write, the write policy would have forced us to do an additional read before each write too.

(a) An example without streaming. The data has to travel over the memory bus to and from the main memory.

(b) An example with streaming. The transfer latency is now no longer affected by the bus accesses.

Figure 3.1: A schematic view of the data-path between two cores, with and without streaming.

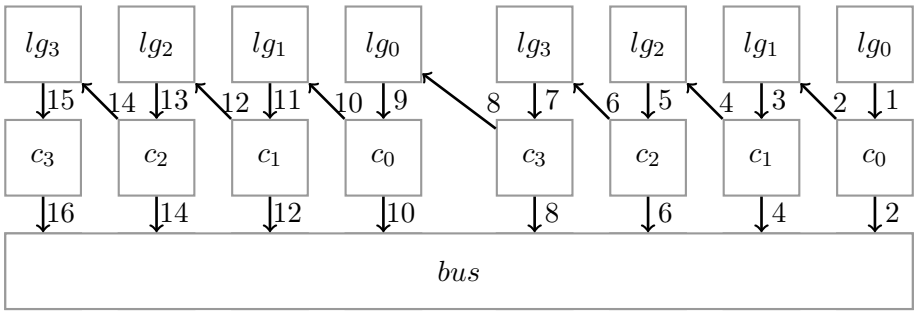


Figure 3.2: A long streaming pipeline example with two cores, each with four contexts. The routing networks of the cache are not drawn for simplicity.

down the latency significantly (especially considering the memory bus traffic is the most expensive part in terms of latency).

3.2 Scalability

The idea of streaming was proposed to create chains of contexts that process data in a pipelined fashion, in order to reduce compulsory miss-rates. In a single core system this can already mean a chain of four contexts, but we also propose to allow streaming to more cores, giving us the possibility of very large pipelines. We should also consider the scalability of this idea before we start implementing streaming.

Let us consider a long streaming pipeline. An example configuration with two cores, each having four contexts, is shown in Figure 3.2. We assume that each context is doing a stream-read every consecutive cycle, so this would be a nice worst case (from the perspective of data traffic). The arrows indicate the data-flow, where the number indicates the cycle number that this memory operation is issued. For example, lg_0 of the right context stores a value in its cache the first cycle. The second cycle, this update is presented to the memory bus (which is shared for all contexts). Also, in the third cycle, the data is available for streaming to lg_1, and so on. Note that the data flows to the left,

which is the opposite of what we have talked about before. This is because up until now we have been discussing streaming from the perspective of the request to the cache, the response (which is the data) flows in the other direction.

One fundamental issue that we can observe from this figure is that we put an enormous strain on the memory bus, by issuing a new write every consecutive stream, which is in the worst case every even cycle! As the memory bus has a write-buffer of only a single item, this means that the second memory operation will already stall lg_1. Even though the data is updated in the cache and thus lg_2 could in theory already stream-read this data, it will also immediately stall because it attempts to update this stream read to the memory too. This goes on until every lane group is stalled because they want to write their updated value. The example is with two cores, but even a single core can already reach this situation easily.

By adding streaming caches, we might have reduced our compulsory miss-rate, but in the process we increased write traffic, which potentially stalls the core.

Let us reconsider the nature of our data-flow. We are working on a pipeline, where each stage takes the output of the previous stage as its input. This data is of a temporary nature, and is not relevant for any stage except the immediate successor. Our next proposal to mitigate the strain on the memory bus by combining the streaming functionality with a method of marking data as temporary: we want to store this data with a write-back policy instead. Since using write-back implies that bus snooping no longer works as coherency policy, we can't simply change the entire cache to a write-back policy. But again, our data is considered temporary and only important to the stream source. We thus argue that it is safe to disable consistency for this data. In other words, to counter the increased write-traffic caused by streaming, we add an additional hybrid write-through write-back cache scheme as well.

3.3 Streaming Caches

In this section we describe our concept of streaming cache. We will start by giving the rationale behind the concept in Section 3.3.1. In Section 3.3.2, we define terms related to stream data-flow directions to ensure consistency in the rest of the work. After that we will look into the routing network of the ρ-VEX cache, and see what implications this will have on the streaming concept, in Section 3.3.3. To avoid the most serious implication from that section (having to double the routing network depth), we provide an additional restriction on streaming in Section 3.3.4. Since the cache blocks can only handle one request at a time, we need to do arbitration as described in Section 3.3.5. Because of how we changed the routing network, we need to do one additional routing step for the stalling signals, which is described in Section 3.3.6. The streaming operator is defined in Section 3.3.7. A brief note on streaming to external cores is given in Section 3.3.8 and on writing in Section 3.3.9.

3.3.1 Rationale Behind Streaming Caches

As mentioned earlier in Section 2.1.2, traditional processing systems usually have a large cache hierarchy. In multi-core systems, data often has to be shared between the different

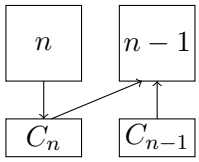


Figure 3.3: The data-flow when context $n-1$ requests streaming data from context n: data is read from cache C_n to context $n-1$.

computational cores. The way this is usually done is by having a hierarchical cache structure where higher caches are shared between two or more lower caches, merging them. This means that the data has to travel from one core to another, it has to go up in to the cache hierarchy until it is at a shared cache between the source and destination cores, and then can travel down again to the core itself (see Figure 3.1a). This means we have a large penalty consisting of several levels of cache write and cache reads. In the worst case, it might even have to go all the way to the highest cache, if the cores are the furthest away from each other in the hierarchy.

The streaming cache approach is introduced in an attempt to bypass this hierarchy by allowing cores to stream-read directly from the data cache of a different core (see Figure 3.1b). Note that we only say stream-reading: writing directly to a different core's cache will not be supported as the scenario where cores are able to modify the cache of other cores will very likely lead to unintended side-effects. Stream reading can be exploited to greatly speed up communication between different cores, especially if a pipeline structure is implemented in software.

Of course this is not straightforward. Note that we need to ensure that this works for any context configuration. The entire routing network exists to ensure a read gets issued to the correct cache block, but also to get the response back to all lane groups that could have issued it. Allowing to stream from any context to any arbitrary other context will require a crossbar, which has an area cost that grows quadratically with the number of nodes, which does not scale well[39]. We shall thus restrict ourselves to streaming from and to neighboring contexts. This is a realistic restriction: remember we are trying to speed up horizontal pipelines. Also, we can choose how to map our program on the different contexts ourselves, so there is no limitation.

3.3.2 Streaming Direction

Let us also clarify the direction of streaming. When we speak of streaming sources and streaming destinations, we define the stream source as the context that issues a stream. This also means that the context who's cache blocks respond to the stream are regarded the stream destination. Note that this is unintuitive from a normal memory access, as the data moves from the destination context to the source. Furthermore, in our examples we will issue stream requests right. See Figure 3.3 for a clarification. Since we also draw our lane groups numbered from high to low, this means that when a stream request is issued by context n, it is sent to the caches of context $n-1$.

This direction is an insignificant choice for the most of the story as we could mirror

everything to work in the opposite direction, only at Section 3.3.4 will this choice be restricted as the only correct way.

We can already see that the original network depicted in Figure 2.3 is incapable of this directly. For example, there is no direct path between lg_2 and c_1 possible, unless lg_2 and lg_3 are coupled.

3.3.3 Fully Reconfigurable Network

Let us first consider a true reconfiguration compatible streaming network. The difficulty is that there are two configuration aspects in both the input and output network: we have to consider both the configuration of the source context that issues the stream, and the configuration of the destination context which holds the target caches.

Let us say we keep the original network to generate requests for each coupled context. We could go for a naive implementation and shift these signals by right once, but then there are two issues:

- The request is not correctly offered to the original target cache(s) any more. If we have a context with a single lane group, our complete request is shifted to the next context. In any other situation, the request will no longer be issued to the lowest indexed cache block.
- The request is only forwarded to the next cache block, which is only correct if the destination context consists of a single lane group. In any other situation, its other cache blocks will not receive the request.

A solution to this would be to duplicate the routing network and place it directly below. This is shown in Figure 3.4. This network now acts as a distribution network to re-distribute the shifted stream request. This solution is very impractical however, due to the simple reason that this routing network is already at the critical path of the processor. Even worse, the same has to be done at the output network to shift the result back once by left! Doubling the input and output routing networks would thus have a serious impact on the frequency at which the core can run.

3.3.4 Restricting the Source

The problem is fundamental: we cannot use a single network that can consider both the source and destinations configurations. To work around this problem, we propose a restriction to the source configuration. If we restrict streaming accesses to only one fixed lane group per context (the first or last, so always a boundary lane group), we can implement streaming with only one network that will take care of the destination routing as it originally does.

Note that the original routing network already enforces that at most one lane group is allowed to do a memory operation per context at the same time, as mentioned in Section 2.7.2, to keep it route-able. The assembler must ensure that this restriction is met at all times. It must only be restricted further to a fixed lane group.

Here the direction in which we stream finally matters: assuming we number our lane groups in descending order, we can chose to stream either left (a stream is issued from

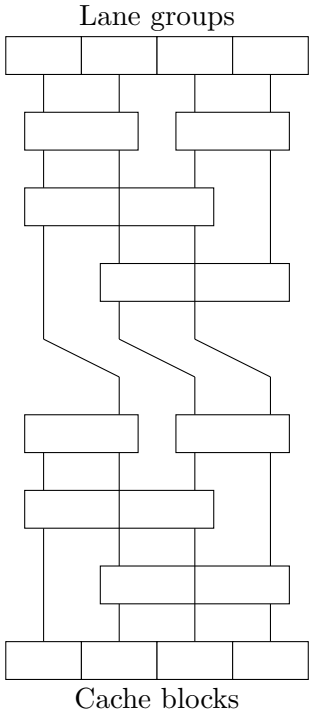


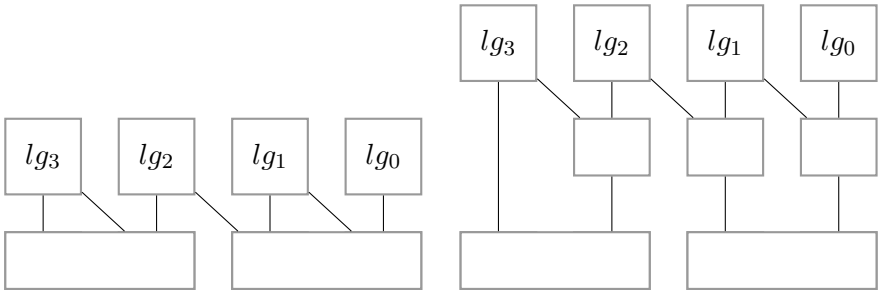
Figure 3.4: The routing network shifted once and then duplicated, to allow fully dynamic streaming.

context n to the cache of context $n+1$) or right (issued from context n to $n-1$). In the former situation, we must restrict the assembler to always put stream requests in the lane with the highest index, while the latter implies the lane with the lowest index.

In Section 3.3.2 we mentioned earlier that we should stream from left to right. The ρ-VEX uses generic bundles[40], which are set up in such a way that a binary compiled for an eight-issue ρ-VEX can still be executed by a four- our two-issue ρ-VEX. This is needed to support four- and two-way contexts. If a four-way context executes an eight-way bundle, it will do this in two cycles, issuing four syllables per cycle.

- The default ρ-VEX configuration only has a memory unit in each even lane.
- The assembler has flags to inform it about the capabilities of each lane, which it has to know to issue all instructions correctly. The default configuration matches that of the default ρ-VEX, so only even lanes get memory operations. The assembler always schedules memory operations on the lowest available lane, practically forcing memory operations in the lowest syllable for every possible configuration.

Since the caches can only handle one read at a time, we are allowed to have at most one memory operation per context. This is also enforced by the routing network. We can go even further by passing a configuration in which only the syllable has a memory unit.



(a) Adding dedicated signals to the routing network.

(b) Arbitrating the memory ports before the routing network.

Figure 3.5: Two examples of restricted streaming. Note that the rest of the network is not drawn as it remains more or less the same (except for the final layer of the output network, which does the reverse of this).

With this restriction, it is no longer required to take the configuration of the source context into account. A stream request can simply be shifted once before it enters the routing network, which will then distribute it over the cache blocks depending on the configuration of the destination context.

The next question is how we offer the streaming request to the routing blocks and to the cache. A simple approach is that we add extra logic and signals to the routing network, dedicated to a streaming request, as shown in Figure 3.5a. We could also try to use the request signals in the original routing network blocks, giving us minimal extra logic. We would only need to arbitrate the read ports for the first layer of the routing network. This is shown in Figure 3.5b.

3.3.5 Arbitrating in the Routing Network

Cache blocks generally can handle only one request at a time, and the cache blocks of the ρ-VEX are no exception. This is furthermore enforced by the routing network, which will assert an error if more than one simultaneous requests are offered to a coupled routing block, and by the compiler/assembler who will not generate code that does not uphold this. This causes two issues with our streaming mechanism, which will describe in this section.

The first issue is that we need to ensure that none of the switching nodes can ever get two memory operations when their inputs are coupled: this will be considered as the pre-routing arbitration. The second issue is that we simply need to ensure that each cache only gets one request at a time.

The former situation arises when we choose the arbitration approach as shown in Figure 3.5b. Let us assume lg_0 and lg_1 are coupled in one context. Now lg_2 issues a streaming request, but in the same cycle lg_0 also issues a memory operation. We now have two memory operations offered to the same coupled switch node, which will generate an error. To solve this the arbiter nodes need to consider memory operations by all coupled lane groups. This is difficult to calculate fast, without relying on the routing network itself, requiring again a duplicate network, so this has to be carefully thought

	Effort in network	Effort in blocks	Performance (delay)	Performance (throughput)
Routing arbitration	+	+	-	-
Block arbitration	-	+	+	-
Dual-Port	-	-	+	+

Table 3.1: The three discussed designs and an estimate of four metrics, as explained in Section3.3.5.

out in the implementation.

The latter issue would automatically solved by solving pre-routing arbitration, as we only allow streaming when all coupled lanes are not issuing memory requests. However in the approach that the routing network (as depicted in Figure 3.5a) will have streaming related inputs, we push the arbitration problem towards the end of the input network. This problem will be called post-routing arbitration, and can be solved in three ways:

- Duplicate the cache memory to create a second port. This approach is not seriously considered as the valid bit memory is currently already a large consumer of area on the Field Programmable Gate Array (FPGA)[7].

- Add an additional read port to the data cache, if this is still available. In that case it might be free as the type of Block RAM (BRAM) available on the target FPGA platforms already have two ports[41].

- Do arbitration over the cache ports, which can cause stalls but should be relatively cheap.

Summarizing we have several approaches to the arbitration problem: we either arbitrate in front and at the end of the routing network or we delay the arbitration to the cache blocks themselves. In the latter case, we can also try to add an additional read port to all cache blocks to bypass or reduce the arbitration problem. These three designs are listed in Table 3.1, with a relative estimate of four different metrics. Routing arbitration requires little changes in the routing network or the cache blocks, but suffers from an increased delay (levels of logic between the core and the caches) and a lower throughput (as the cache blocks are arbitrated). Block arbitration requires more effort on the routing network but less in the blocks, suffers from less increased delay but still requires arbitration over the blocks. The dual-port design requires most changes in both the network and blocks, but adds no additional levels of logic and also requires no arbitration and thus should have the highest throughput.

As we see in these metrics, there is no clear winner, especially since these metrics are only estimated. We cannot accurately estimate how much effort is required unless more implementation details are examined, or how much the delay differs without having a structural model. We shall thus discuss implementations for all three designs in Chapter 4.

3.3.6 Stalling Synchronization

We also need to consider how to stall everything correctly. The cache is responsible for stalling lanes in certain cases: when its handling a read miss, or the lane is issuing a write while the write buffer is already full.

The ρ-VEX requires that this stall signal is the same for all lane groups that are coupled in a context together, and it will not behave correctly otherwise. The output network is responsible for routing the stalling signals such that this property holds. We must ensure that this property also holds when streaming.

A few different classes of stalling behavior can be identified in streaming. Since a single-port cache block can only service one read at a time, in cases where a stream is issued to a context that is already doing a memory request, we might have to stall the streaming source context. The opposite might also be true, as we might grant a stream request in arbitration, making the destination context stall. The former is relatively easy, since the output routing network distributes stalling signals to all coupled lane groups. The latter needs extra care as the stream response is only shifted back once, to the lane group that issued the stream.

3.3.7 Streaming Operator

Up until now we have always assumed that the data that we request exists in the destination cache, and thus we are able to stream this. In other words, we have always assumed a streaming hit. This is expected for a pipeline, where earlier stages generate data for later stages to handle, but we should also consider all other situations too. For example, the later pipeline stage could have been delayed and the earlier stage already replaced his cache line with a different item. In this chapter we will define the behavior of the streaming operation more clearly.

We want to be able to bypass the memory hierarchy in the case that the previous pipeline stage already has data available in its cache. This means that we should check if a hit occurs in its cache, and in this case take this data instead. This also means that if we have a hit in both our own cache and our destination's cache, we should prioritize his data. In the case of a miss in the destination's cache, there can be two situations: either the destination did not change anything, or it was overwritten by a different cache line. Both cases mean that the memory holds the most recent version, thus we can safely process the memory read as normal.

Summarizing, the streaming operation is defined by the following function $stream_n(a)$ in Equation 3.1a as a stream operation by context n at address a, where $c_n(a)$ and $m(a)$ denote the values read from the cache of context n and the memory at address a respectively. When deciding if we have to update the cache line, we now have to consider both hit signals, as shown in Equation 3.1b. A circuit design depicing the same boolean logic equations is shown in Figure 3.6.

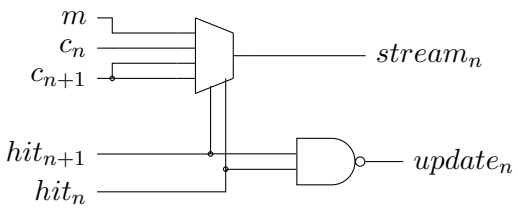


Figure 3.6: A circuit design of the streaming operator and update signals expressed in Equation 3.1. $stream_n$ is the selected output data, c_n the value read from the cache line (both for cache block n), and m denotes the value read from memory.

$$stream_n(a) = \begin{cases} c_{n+1}(a) & \text{when } hit_{n+1}(a) \\ c_n(a) & \text{when } hit_n(a) \\ m(a) & \text{otherwise} \end{cases} \tag{3.1a}$$

$$update_n(a) = \overline{hit_{n+1}(a) \cdot hit_n(a)} \tag{3.1b}$$

3.3.8 External Streaming

If we take one more look to the abstract designs for streaming in Figure 3.5, we see a quite regular structure, except for the first and last lane groups. We could make this design completely regular by adding an additional input and output on the left and right sides respectively. With this idea in mind, we could even generalize streaming from context to context into core to core. Assuming all cores share the same memory, there does not seem to be anything that could not be generalized in this way. We need to expose the streaming signals to outside the core, and keep this requirement in mind when implementing.

3.3.9 Writing

We mentioned earlier that stream-writing will not be supported. This means that a write by the stream destination context to a streamed location will be handled like any other write. The context will write it to its cache and it will be committed to the main memory, while being invalidated in the source context. This behaves as expected, but should only be done with caution.

3.4 Hybrid Write-through and Write-back Support

In this section we describe a hybrid write-through and write-back design for the cache. Firstly we describe the rationale behind this concept in Section 3.4.1. In Section 3.4.2 we very briefly compare our situation with two existing implementation examples, to

see how they would apply here. Next, in Section 3.4.3 we discuss how to classify data as write-back instead of write-through. Section 3.4.4 states that we have to disable bus snooping for write-back data, and Section 3.4.5 discusses the situation where the write-back feature is disabled at run-time.

3.4.1 Rationale behind Hybrid Caches

We mentioned earlier in Section 3.2 that the current cache implementation is not directly suited for streaming. Using write-through caches to stream data can give us large amounts of unnecessary bus-traffic that will fill up the writing buffer quickly, while a full write-back cache will be difficult to implement from a coherency perspective.

The easiest way of working around this is to allowing some data to be stored in a write-back fashion with consistency (momentarily) disabled. What we propose is that a cache can keep values in its own cache, without letting them write back to the main memory.

This would normally be dangerous in a multi-core system from a coherency perspective, as it is asking for the situation where a context modifies a value but keeps it in its cache, followed by a read (on the same location) issued from a second context: this context will get the old value.

This problem can still occur in this proposed system, and the programmer has to be careful when accessing these special write-back values, or race-conditions can occur. By definition, using write-back in a multi-core or multi-context system is dangerous and can only be done safely if the write-back regions are considered private, or when we use the proposed streaming structure.

If we allow a context to read write-back stored values from its neighbors' cache, that means he can also safely read this value at all time. Consider the situation where ctx_n tries to read address a from the cache block owned by ctx_{n-1}.

- Address a has not been modified by ctx_{n-1}. This means the memory location is still the most recent copy of a.
- ctx_{n-1} did modify location a. It has to be in the cache, because when the line is removed from the cache it had to be written back to the main memory first (meaning it is no longer a local modification). From that we can assume that it is always in the cache. This means we can successfully stream it.

We must also consider the situation where two contexts have overlapping write-back ranges. This is strongly discouraged from the perspective of recovering consistency (this will be discussed in more detail in Section 3.4.5): if we only have one core that can locally modify an address, we can fairly easily recover from it: the local value should be pushed to the main memory. If we have overlapping write-back ranges, each context can have its own local version of an item, and it is now difficult to select one from these to commit to the main memory.

To summarize, write-back memory is only safe if it is considered private, or a range to stream from. Conversely, streaming is always considered safe, but will suffer from a penalty when done to normal write-through memory. As streaming is a safe way of

accessing write-back and write-back memory produces no writes on the bus, we propose a pipeline setup where each context defines an own, non-overlapping write-back range. Contexts are allowed to stream read data from the memory of the next context, and should store it (and possibly modify it) in their own write-back region.

3.4.2 Comparing to Existing Implementations

The idea of allowing both write-through and write-back in the same cache is not new. We discussed an example earlier in Section 2.4 and Section 2.5 of how modern Intel and ARM architectures allow these. We will now briefly see how these implementations would apply to our situation.

Both the ARM Cortex A8 and Intel allow caching to be set on virtual memory pages, requiring an entire Memory Management Unit (MMU). Since this unit already exists for many other reasons, it makes perfect sense to do so. The ρ-VEX design that we worked with did not have a MMU, and designing one would be an entire work on its own.

Both ARM and Intels designs work from the assumption that write-back is the default cache policy, and optionally offer a fall-back to write-through. Write-back in general performs better than write-through, so this again makes sense. The most obvious use-cases for falling back to write-through on a certain memory page is when writing to or reading from this page has side-effects, for example when it is mapped to peripheral devices, control registers or frame-buffer memory.

For us the relation works the other way: we have write-through memory and we want a part of it to be used in a write-back fashion. This is a means to a different goal as well, as we want to avoid write-traffic when streaming and it is thus temporary.

The biggest differences are simply that we do not have an entire MMU to work with, nor do we need multiple ranges, and truly implementing write-back would mean we need to solve coherency in a different way, which is also a project on itself.

3.4.3 Classification of Temporary Data

The first task is to classify the data that is of this nature so that we can exclude it from write-through. An important requirement is that it can be done dynamically, so statically reserving a part of the cache's address space is out of the question. Furthermore, programmability and additional hardware cost (disregarding the actual write-back logic) has to be taken into account.

An easy (from the users' perspective) solution would be to add some sort of code annotation or *pragma* that marks a variable as temporary, and should be treated as such. When a variable is marked as temporary, all memory access instructions that are related to it should be made aware of it as well so that they are treated as write-back (for example adding a bit to the opcode, or adding a separate set of opcodes). From the implementation perspective however this implies not only compiler patches but also changes to the instruction coding scheme, requiring a lot of hardware changes.

Another approach is not variable based, but memory address based. Instead of marking variables as temporary, we mark a memory region as write-back. This puts a bit more effort on the programmer, as he has to communicate to the system what memory range is write-back (by writing it into some memory mapped control registers)

and has to manage this memory manually, but from the implementation perspective seems more realistic as we simply treat memory operations to addresses within that range as write-back, requiring nothing more than a comparator. The additional programming complexity can also easily be alleviated via a special *malloc* that updates the register automatically.

3.4.4 Disabling Snooping

The principle of snooping was described earlier in Section 2.3.1. Recall that this is a straightforward way to enforce coherency between different processors in a write-through implementation by simply invalidating all lines that hold data when it is being overwritten in the main memory.

However, when we update a write-back value, it is not immediately written back to the main memory. Invalidating the corresponding address is thus not useful at all: the other contexts will only re-read the same value. We must thus at least delay the invalidation to the cache line eviction step.

3.4.5 Disabling Write-back

One more aspect has to be looked at however: what to do with the write-back region after we disable write-back. If we do nothing at all, contexts that modified data inside the write-back region will now have local copies of that that differ from the main memory in their caches, and such we have inconsistency. One can argue if this is really a problem as the intended goal was to allow for temporary data storage to enable the use of streaming caches, but the feature could be used in more different ways.

Three strategies to manage this problem will be implemented, giving the programmer some more control:

- Flush. Flush all dirty data to the memory. This requires us to check all cache blocks for their dirty bit and worst case send a large amount of data at once, which might actually cost us all the theoretical speedup. Nevertheless, it is a relatively safe default to avoid accidental data loss.
- Invalidate. Invalidate all dirty blocks in the cache. This forces re-fetches for all locally edited blocks. This is an easy way of avoiding the write traffic introduced in the previous strategy, where data that is important after leaving streaming should be written outside of the write-back area. This strategy gives a very easy to use clear separation of write-back area as context-local temporary memory.
- Nothing. Clear dirty bits and do nothing to restore consistency and coherency. For those who want complete control, for example when the programmer will re-initialize the memory.

Note that only the first strategy leads to a predictable memory state (assuming no other context has a local modification, leading to conflicts and race conditions). Strategy three obviously leads to inconsistent memory. Strategy two will lead to consistent memory, but it is hard to predict in what state exactly it will end up, as cache lines might have been evicted or not.

3.5 Conclusion

A design has been proposed in this chapter to serve as an alternative to a cache hierarchy for multi-core systems, in order to alleviate penalties observed when doing extensive data sharing between cores. This streaming cache design aims to bypass the cache hierarchy in a simple but effective manner for reading operations.

The penalties of accesses between different contexts or cores was discussed first in Section 3.1. An additional issue with scalability of the streaming solution was determined in Section 3.2.

The streaming concept was described in more detail in Section 3.3. Initially the rationale behind it was once again explained in Section 3.3.1. The streaming direction was defined in Section 3.3.2.

The reconfigurable routing network to the cache blocks was considered in Section 3.3.3. Two options were considered to ensure a path between the stream source and stream target: a duplicated routing network or by restricting the source context. The latter was chosen as the former would impose a severe penalty on both the critical path and the area usage, and it was discussed in more detail in Section 3.3.4.

The next decision was at which place to arbitrate between streaming and normal accesses, in Section 3.3.5. The choice was either to re-use the existing memory request signals by adding a pre-network arbitration block, or to add additional signals to the routing network for streaming. The former solution would guarantee that each cache block can only get one memory request at all times, allows us to keep the cache blocks and the routing network the same but will affect the critical path, while the latter solution moves the arbitration problem to the cache blocks (unless we can use dual-port cache blocks) which might affect the critical path again, and modifies the routing network extensively. Since no clear winner is determined in this chapter, the implementation of all three will be discussed in Chapter 4.

In Section 3.3.6 we looked at the effect of streaming on stalling signals. Stalling is required to be synchronized as well between all contexts that are involved in a stream operation that is blocked (due to arbitration, unless dual-port caches are used), but we also need to distribute any stalling signals due to the arbitration to all contexts that are coupled to the stream source.

The streaming operator was defined in Section 3.3.7 to ensure we always prioritize the source context, as we consider it a producer-consumer relationship.

External streaming (that is, streaming to contexts of different cores) was also considered briefly in Section 3.3.8.

In Section 3.4 the hybrid write-back write-through concept was described. Firstly the rationale behind it was discussed in Section 3.4.1. The concept was compared to existing work in Section 3.4.2.

In Section 3.4.3 the decision is made to allocate a range of memory as write-back instead of marking individual data items as such, in order to stay compatible with the existing tool-chain and instruction set, as both do not need to be aware of the new features. Inside this region, the main cache controller needs to behave as a write-back cache controller instead of write-through. For that we also need to ensure that bus-snooping is disabled and add dirty bits to the cache lines as mentioned in Section 3.4.4.

To ensure the user can also disable the write-back feature, some form of consistency recovery is required. A controller was proposed in Section 3.4.5 with three methods, giving the programmer control over data priorities.

4 Implementation

In this chapter we will work our way to an implemented design by first designing the interface that is exposed to the user, creating a high level simulation model and then discussing the actual design.

We need a well-defined way to expose the newly added features to the user, which will be described in Section 4.1. This interface is defined first so that it will be the same in both implementations, keeping them binary compatible.

In Section 4.2, the high level simulation model will be implemented. This is done in a few steps: we have to increase the level of detail of the existing simulator slightly which is described in Section 4.2. The write-back functionality is added next, in Section 4.2.2, followed by the streaming functionality in Section 4.2.3.

The hardware design will come up next, starting in Section 4.3: firstly we need to add a controller which processes the control registers defined in Section 4.1. It handles arbitration to allow multiple contexts to change values at the same time, among other things.

The hybrid write-back write-through concept is implemented in Section 4.4. Two required changes to the main cache controller are described in Section 4.4.1. The first is solved in the same section, while the second is discussed in Section 4.4.2, Section 4.4.2.1 and Section 4.4.2.2.

Next, the streaming cache concept is implemented in Section 4.5. The circuit to detect stream requests is described in Section 4.5.1.

Next, the three implementations as mentioned earlier in Section 3.3.5 will be discussed one by one. Routing-level arbitration is described in Section 4.5.2, while cache-level arbitration is considered in Section 4.5.3. Dual-port streaming is discussed in Section 4.5.3.5.

Cache-level arbitration requires more work in distribution of stalling signals (Section 4.5.3.1), the cache block arbitration itself (Section 4.5.3.2), and finally how to handle multiple stall sources per block (Section 4.5.3.3).

Streaming to external cores is discussed in Section 4.5.4, and the final controller required to restore consistency after write-back is disabled will be discussed in Section 4.6.

4.1 Interface

As mentioned before, we should now define the interface of both the streaming and write-back features. If we offer the same interface in both the simulator and the core, we keep binary compatibility between these two. Hardware features are usually exposed via control registers. We will first discuss streaming, and go into write-back after that.

4.1.1 Global Control Registers

The ρ-VEX has global and context local control registers. Contexts cannot directly write to a global control register, so if any kind of global configuration has to be done this is usually implemented by having a global and local register pair. When a local register is updated, the core hardware can choose to update the global version. The context configuration (the mapping between contexts and lane groups) is an example of a hardware feature that is controlled in this manner.

This approach is very useful as we can implement arbitration this way: if more than one context attempts to update the global register at the same time (meaning they both write a value to their local variants at the same time), the hardware can choose to grant one of them. Another useful feature is that the hardware can also do error checking: if a control register only has a limited amount of valid values, the hardware can refuse incorrect values (by not updating the global variant). In other words, if a context wants to do change a global hardware setting, it should update its context local register and check to see if this request was granted by monitoring the global variant.

4.1.2 Streaming

We mentioned earlier in Section 3.4 that for write-back data we require a method of classifying the data as such. The same holds for streaming: how do we mark specific data as streamable? The conclusion from that section can be extended to this situation as well, and such having an address range where we can store streamable data is a natural conclusion. In Section 3.4 we concluded that the write-back memory can only safely be read from by other contexts via streaming, while in a normal pipeline all contexts (except maybe the first one) stream-read from write-back ranges to write-back ranges. The first context could also safely put its stream data in its own write-back range and benefit from the reduced write traffic as well. Taking this all into account, we consider streaming a natural extension from the write-back ranges, and thus decide to combine both the streaming and write-back ranges into one range.

It has to be observed however that write-back ranges can be applied for more goals than just streaming. An example would be the stack: each context could put its stack in a write-back range, since the stack is by definition context local. This means that we define this shared range as the write-back range to make it generalized, and add an additional option to enable streaming from the write-back range.

In conclusion, for streaming we only add one global configuration register with a context local variant, as the streaming configuration and reconfiguration registers. Their mnemonics, if they are global or context local and their full names and usage are shown in Table 4.1. In these registers, only bits 7 to 0 are currently used: each bit corresponds to a context and is high when this context enables streaming from its write-back region, as shown in Table 4.2. The rest of the bits are reserved for future use.

4.1.3 Hybrid Write-back Interface

Next, we discuss the interface for the hybrid write-back and write-through caches. Remember that we have an address range and a consistency recovery mode to set. Disabling

Mnemonic	G/L	Name
DSRR	L	data cache streaming reconfiguration register
DSCR	G	data cache streaming configuration register

Table 4.1: The control registers added to the platform for the streaming cache behavior, with their mnemonic, a letter indicating which one is global and local and full name.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																								s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0

Table 4.2: The meaning of each bit inside the DSCR and DSRR registers: bit s_n is high when context n enables streaming from its write-back region. All other bits are currently unused and reserved for future use.

and enabling the write-back range quickly might also be useful in certain cases, especially when setting up, so that the programmer can set each field separately (depending on some branches in the code) and enable the feature in a single go when all fields are set up.

An easy and straightforward solution for the address range is by adding two registers that act as 32-bit memory locations that are the boundaries of the write-back region. As these are memory locations, they have to be in 32-bit. These values can easily be used as comparator values when selecting between write-through and write-back logic. A possible downside of this is that we need a third register to add more control bits like the enable and the consistency recovery mode. Note that registers are relatively expensive in the r-VEX platform.

Note that the addresses have to be data cache-line aligned (otherwise things will get ugly, since we then can have a cache line that's partially write-back and write-through). As the current implementation has data cache lines that are 32-bit wide, this could free up two bits per register to use as possible control bits. To enable/disable and encode the three consistency modes we would need only three bits, but this is not really future-proof.

A second method could be to use an address mask instead of an address. From an implementation perspective this would require a single 32-bit register and an and gate, but this is quite complicated to use for the user.

4.1.3.1 Using Address and Size Registers

Another method is by using an address and size register. One reason to go for this solution would be to avoid the need for two 32-bit values and we can easily allocate less bits to the size register, freeing them up for control bits. It is also very easy to use, since it is natural to use in conjunction to a malloc call (which requires a size and returns the address of the beginning, exactly the two values we need). A possible downside to this is that we will require an adder to calculate the end of the region for the comparators. However, adders are most likely cheaper than registers in the r-VEX.

See Table 4.3 for an overview of the new registers. As one can see there are two added: one that acts as a control and one that acts as an address register. Currently, 16 bits were assigned to the size, as is shown in Table 4.4. As we have three consistency

Mnemonic	Name
DWBCRR	data cache write-back control request register
DWBARR	data cache write-back address request register

Table 4.3: The control registers added to the platform for the write-back cache behavior, with their mnemonic and full name.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
en														c_1	c_0	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s	s

Table 4.4: The meaning of each bit inside the DBWCCR register: en is the enable bit to enable or disable the whole write-back feature in one go, the c bits are used to set the consistency recovery mode and the s bits are used as the size. All other bits are currently unused and reserved for future use.

recovery modes, assigning two bits is enough for now. Their association is shown in Table 4.5.

4.2 Simulation

In this chapter we work towards an implementation by adding the features to simrvex, a fully functional working simulator of the r-VEX processor. First we discuss changes made to the simulator to facilitate the changes, then we go into the hybrid cache functionality and after that we will discuss the streaming caches. The interface as described in Section 4.1 was implemented to allow binary compatibility with the later follow-up implementation. The implementation served to explore, analyze and verify the original concept before actually making a hardware implementation. The result of the analysis is given later in Section 5.2.1.

4.2.1 Preparation

Although the simulator is fully functional and simulates the behavior of the processor in an accurate way, the internals were not yet completely suitable for our purposes.

More specifically, the cache's storage was not simulated: only hit and miss penalties. This was most likely done to speed up the simulation time: since the cache operated in a write-through fashion, the cache is always consistent. Therefore, it doesn't matter if we

c_1	c_0	Strategy
0	0	Flush
0	1	Drop
1	0	Nothing
1	1	Reserved

Table 4.5: The values for the c bits and the corresponding consistency recovery mode strategy as discussed in Section 3.4.5.

read the data directly from memory or from a local data-structure containing the cache lines, as long as we keep track of tags to check if the cache access was a hit or a miss.

This proved to be not accurate enough for our purposes, as the hybrid cache scheme aims to disable consistency over a range of data. The simulator was thus first modified to simulate the storage of the data cache lines in a write-through fashion, and augmented with several consistency checks.

To do this, a cache line field was added to the cache block tag structure and when a read miss was determined, the cache line is fetched from the main memory to the cache line. Both in a miss and hit case, the value read is taken from the cache line instead.

Writes were originally always directly to memory, but now a slight change had to be made. In case of a write miss, the complete line has to be fetched first to the cache. It is updated there, and the cache line can be written back. Fetching the line is skipped though when the write is a complete cache line size.

Several functions had to be separated or modified as well. For example, the original implementation determined hits and misses inside the same function that invalidated cache lines, calculated which cache line had to service the access and incremented penalty counters. It also never exposed any of this information either, since the original implementation only simulated these penalties, and the read/write functions would just operate directly on the memory anyway. This function was modified to return hit or miss information and the cache block that would handle the request, and invalidation was removed entirely from this function (since the need to invalidate now depends on if it is a write-back access or not). Another version was also added that did not incur any penalties either, so we could just test for hits (which will be needed in the case of a stream request).

As the original behavior was to read directly from the memory, consistency checks were fairly straightforward to implement: just ensure that the value returned from the cache line was the same as a direct read to the memory location. Assertions were added to all memory reading and writing operations to verify that the cache line we selected (and would return) was always the same as what was returned in the original implementation (directly the memory). As this assertion was never violated during any test program executed, we know that the modifications did not change the functional behavior.

4.2.2 Write-back Simulation

Firstly, the control registers for write-back behavior (the range, if it is enabled and the consistency recovery mode) were added and each data cache block was extended with a dirty bit per line (default 0). For obvious reasons, the consistency checks were disabled if the address was inside the write-back region (as we are allowing inconsistency there on purpose).

When handling a read miss and thus a fetch or writing to a cache line, we checked if the target line was marked as both valid and dirty. In that case, it would be evicted first to its original location and invalidated in all other caches before being overwritten. The original address of the cache line to evict is not available directly any more but it can be calculated fairly easy. Recall Equation 2.1, where we shown how to calculate both the offset and tag components of an address. We can combine the inverses of both equations

to construct back the original address as shown in Equation 4.1a.

Unfortunately we only know the tag of our original source address (we will now call m_s), as this was stored in the tag field. We know the current memory access is performed on the same cache block, so their offsets are the same (Equation 4.1b). This means that we can fill in $tag(m_s)$ (which we read from the tag field) and substitute in $offset(m_a)$ to construct the original address as shown in Equation 4.1c.

$$m_a = tag(m_a) \cdot |c| + offset(m_a) \tag{4.1a}$$
$$offset(m_s) = offset(m_a) \tag{4.1b}$$
$$m_s = tag(m_s) \cdot |c| + offset(m_a) \tag{4.1c}$$

When handling a write, a check was added to see if it was inside the currently active write-back region. In that case, the cache line would not be written back to the memory, but only marked dirty. Invalidation of the line in all other caches is also skipped in this case.

The consistency recovery modes were also implemented as discussed. When write-back would be disabled, the cache blocks will iterate over all lines and either write back all dirty and valid lines, unset their dirty bit or do nothing, depending on the selected consistency recovery mode.

To summarize, for a write-back implementation we have the following requirements:

1. During a write, if it is in the write-back region, don't write to the memory, don't invalidate the line and mark the line dirty.

2. During a read or write miss, if valid and dirty, evict and invalidate the line.

4.2.3 Streaming Simulation

Streaming was fairly straightforward to implement in the simulator. When doing a read, the stream destination's cache block was tested first to see if it enables streaming and the currently requested address is within its write-back region. If that was the case, a hit was tested in the destination cache (without penalties): if it had a hit, its line would be fetched and returned instead. Otherwise, the function would continue as normal, to behave like a normal access.

We also added a performance count register that would simply count the number of accesses that had a hit in the neighboring cache but a miss in our own cache. While this is hard to relate to actual cycles and performance gain, it will still give us an idea about how often we can do a streaming bypass.

4.2.4 External Streaming

Testing the core to core streaming feature was considered, but unfortunately implementing it in the current model would be a considerable amount of effort. This is because the current simulator is not at all suitable for a multi-core setup, thus is would require a lot of work just to make multi-core simulations possible.

4.3 Control Register Controller

The first component that had to be added was the controller that would manage the newly added control registers for both streaming and write-back configuration. This component handles the arbitration for the stream mode reconfiguration and assigns the configuration that wins arbitration (and refuses invalid configurations). For write-back it calculates the end of the write-back region and it converts the write-back configurations from contexts to lane groups.

4.3.1 Arbitrating the Stream Configuration

When a context writes to any of the stream reconfiguration request register, the controller will take the one with the lowest context index. If any of the contexts issues an invalid configuration, an error signal is asserted, even though this is currently not yet used for any purpose.

4.3.2 Write-back Conversion

The core has a set of local control registers per context, so this also counts for the write-back related registers. As we have a cache block per lane group, this controller takes care of the mapping.

The control register values are latched when they enter the controller. Relevant information is extracted from here in the next cycle. The write-back enable, region start and consistency recovery mode bits are simply extracted. An adder is inferred per context to calculate the end of the write-back region, and a latch is used to detect when write-back is disabled (making a signal to activate consistency recovery high for one cycle).

Next, these values are converted from contexts to lane-groups, and then latched again. These values are exposed to the data cache. There is thus a two cycle after between changing write-back related signals.

4.4 Write-Back

As mentioned earlier, a lot of the potential performance gain from streaming caches might be lost again due to the write-through behavior of the caches. To negate this penalty, we propose a hybrid write-through and write-back cache feature in this section.

Our proposed schema is such that the user can specify a region of the memory to behave as write-back instead of write-through. This means that writes inside this range will not be written back to the main memory directly. This feature will be usable regardless of streaming, as there are more scenarios where this can be useful. Imagine for example that each core marks its stack area as write-back. This should be completely safe, because the stack is always local to each core. The stack is also typically written to very often, thus we can avoid a lot of write traffic to the main memory.

The idea is fairly simple. As we do not immediately write data back to the main memory, the cache needs to be expanded with an additional bit per cache line: the

V	D	Meaning
0	0	Value in cache is outdated and should be fetched from the memory.
0	1	Value in cache is modified but the main memory has also been invalidated (by a different cache). We chose to prioritize the value in the memory here.
1	0	Value in cache is up to date with the main memory.
1	1	Value in cache is modified, so before overwriting we should evict it to the main memory.

Table 4.6: Summarising the different combinations of the valid and dirty bits, and their meaning.

dirty bit. For a feature that is handled in a later section, we require the ability to clear all dirty bits in a single cycle. Fortunately, the valid bit storage is capable of doing this, so this component can be re-used. Whenever a line inside the write-back region is written to (and it is thus not written to the main memory) we mark it as dirty. Now the combination of the valid and dirty bits give us information about the state of the current cache line. See Table 4.6 for a brief summary of all combinations.

The situation not valid and dirty is tricky: here the core has modified the cache line (hence it is dirty), which it could only have done when the line was still valid. However, it was later invalidated by a different core. We have to make a choice in this situation, do we trust the local field and do we evict over the updated value, or do we trust the value that has been written by another core more and do we need to update the line instead? Since write-back regions serve as local storage for contexts, it makes sense to prioritize the value of the context that owns it. Either way, it cannot be perfect in all situations, so this situation should generally be avoided. As neither way is perfect as there is always a situation were it behaves incorrectly, we make the valid bit hold priority over the dirty bit: the dirty bit only has a meaning when the line is valid. This is mostly done to keep the design simpler.

The cache's state machine only needs to be expanded with a few states, transitions and one additional write buffer which we will call the evict buffer. The original state machine is depicted in Figure 4.1a, while the expanded state machine is shown in Figure 4.1b. This buffer will store if the current access is inside the write-back region or not (this will be used during write miss evicts), the currently selected cache line and its original address, reconstructed from the cache tag and the currently selected address in certain cases. We can use Equation 4.1 to calculate the evict target address. Note that the cache size ($|c|$) is a power of two, so there is no need for a multiplication: the destination address is simply the tag and offset bits concatenated.

$$evict = valid \cdot dirty \cdot \overline{hit} \tag{4.2}$$

4.4.1 Changes to the Cache Controller

Remember in Section 4.2.2 we defined two requirements for write-back memory:

1. During a write, if it is in the write-back region, don't write to the memory, don't invalidate the line and mark the line dirty.

2. During a read or write miss, if valid and dirty, evict and invalidate the line.

The first requirement is met easily. When writing, we just check if the address is inside the write-back region. If this is the case, we skip writing it to the main memory but mark it dirty. This is simply implemented by informing the Central Processing Unit (CPU) that we accepted the write as the existing state machine already did, but without putting the memory request on the bus. Instead of transitioning into a state that waits for the memory operation to complete, we transition to a state that waits for the CPU to finish its current tasks so it is ready to read the result (or directly back in the idle state if it was already finished). Invalidation is actually done by the cache block only when an address is written to the memory, and since we don't offer the write to the memory, it is also not invalidated.

4.4.2 Evictions

The second requirement requires a bit more work. When handling a read or write miss (and thus updating the cache line with a different one) we need to check if the line has to be evicted or not. Remember that we only need to evict when we are overwriting a valid, dirty cache line with a different cache line (which is implied by having a low hit signal). This is expressed in Equation 4.2. The actions that we need to perform aside eviction differ, so there were two different evict state paths added: one for an update evict (after a read miss) and one for a write evict (after a write miss). Nevertheless in both cases, we store the current cache line and its target address in the evict buffer, which takes one cycle. We can also already start writing the evict data to the main memory in the same cycle, and this time we get invalidation automatically.

4.4.2.1 Update Evicts

Now things get different for update- and write evicts, let's discuss updates first. In the case of an update, the first cycle we also clear the dirty bit. After that, we go to a new state that keeps issuing the write until it is acknowledged. We then return to the original state: the CPU is still stalled and issuing the original read, but the dirty bit has been cleared now. This means we no longer need to evict, and the read will now be processed normally.

A possible improvement to this behavior is to handle the update first (since we have the old data in the evict buffer anyway), so we can let the CPU continue. After that, we can handle the evict. Note that the CPU can issue another operation that requires a memory access in the next cycle (write, update or evict). We thus need to handle this evict like another write in the write buffer, and thus stall the CPU if it tries to do this while the evict is still in process.

4.4.2.2 Write Evicts

In the case of a write evict, we need to take a look if the data we are overwriting our line with is in the write-back region or not. If it is inside, we can keep the dirty bit as it is, since our new write has to be marked dirty again. If it is outside the write-back region, we have to clear the dirty bit.

We also store the write itself to the write-buffer as would be with a normal write. The next state (which is again single cycle) will keep the evict buffer data on the memory bus, perform the actual write to the data cache and issues a release of the CPUs stall for the next cycle.

The state after that will keep the evict buffer write on the memory bus until it is acknowledged, and uses the bit in the evict buffer that stored if the access was a write-back write or not to determine wether or not we should transition into the normal writing state. The normal writing state already assumes the data is in the write buffer and continues to present this write to the memory until it is acknowlegded.

4.5 Streaming

In this section we will discuss the streaming mechanism as implemented. Three designs have been considered in chapter 3.3.5. All these three designs will be discussed in this section.

- Routing Level Arbitration: arbitrating before and after the routing network, while using the original read request signals.
- Block level Arbitration: arbitrating just before the cache block, adding additional stream-read signals to the routing network.
- Dual Port: using additional stream-read signals and an additional read port to bypass arbitration all together.

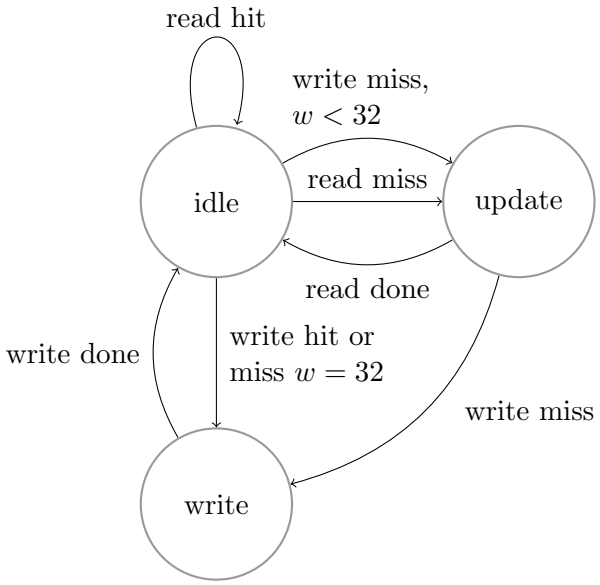
The first solution requires us to add an additional layer for arbitration before and after the routing network. Instead, the second and third solutions both require us to add some additional stream signals to the routing network. The second solution also requires arbitration at the cache block, while the third solution requires an additional read port on the cache. Before we discuss any of these solutions however, let us first look at how to determine if a lane group is doing a stream request or not.

4.5.1 Detecting Stream Requests

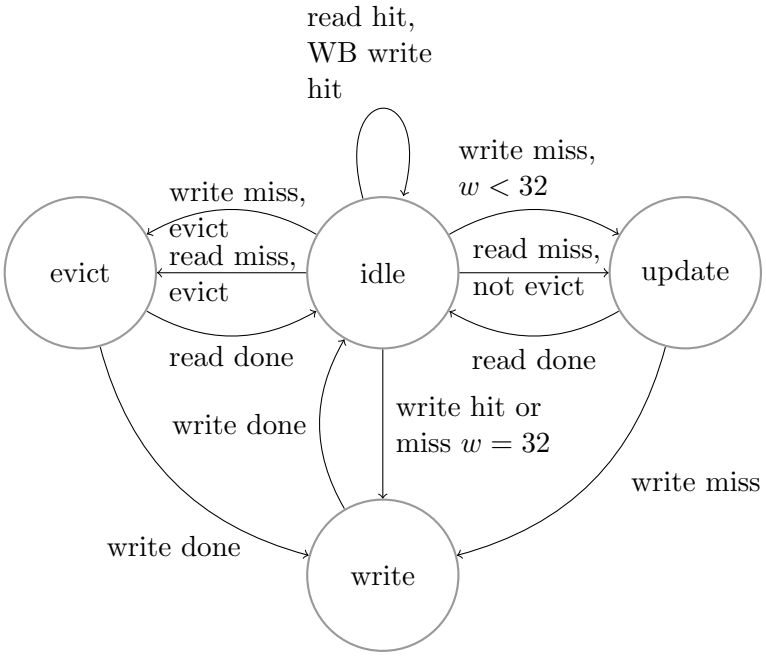
First we should determine if a lane group is actually doing a stream request. Let us consider that lane group n wants to do a stream request to the cache of lane group $n-1$. The routing network will ensure that this stream request is routed to all coupled cache blocks. Obviously, $n-1$ should have streaming enabled and n has to issue a read request to an address inside the write-back region of $n-1$.

But let us now consider the situation were caches n and $n-1$ are coupled together, their context has streaming enabled and n does a read inside its own write-back region. Unfortunately, all the conditions for a stream read are met. We thus need one more piece of information, since a stream read should only occur on a boundary between contexts, more specifically the lowest indexed lane group of a context.

Note that the ρ-VEX internally keeps track of the configuration by holding a decouple bit per lane group. This bit is high when this lane is considered a master lane group,



(a) A state machine for a write-through cache.



(b) A state machine for a write-back cache.

Figure 4.1: State machines for write-through and write-back implementations.

which means it is the highest indexed lane group of the current context. Any lane groups with a low decouple bit are considered slave lanes to the first higher indexed lane group with a high decoupled bit.

When we stream right, our source is lane group n and our destination is $n - 1$. This means that the stream is issued at a boundary only when the destination lane is

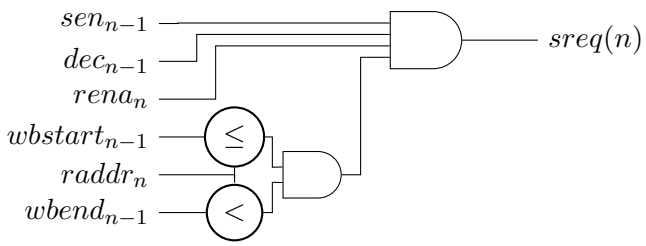


Figure 4.2: Determining if a read request is a stream request. This circuit needs to be instantiated for each lane-group.

decoupled: is is the highest indexed lane group in its current context, which implies that the next lane group is from a different context.

Let us define some symbols for all these properties and an equation. $sreq(n)$ is our function that is high when a stream request is detected for lane group n, while sen_n implies that streaming is enabled for n. dec_n is the decouple bit of n, $rena_n$ is the read enable bit of lane-group n. $raddr_n$ is the read address issued by n, and the set WB_n contains all write-back addresses as defined by the write-back region of the context linked to lane-group n, and LG is the set of all lane groups.

Taking this all into account leads us to Equation 4.3 and Figure 4.2.

$$\forall_{(n \in LG)}[sreq(n) = sen_{n-1} \cdot dec_{n-1} \cdot rena_n \cdot raddr_n \in WB_{n-1}] \tag{4.3}$$

4.5.2 Routing Level Arbitration

In this implementation we rely on an arbitration layer before and after the routing networks. The main advantage of this approach is that we do not need to touch the existing arbitration network and cache blocks: for them, it just looks like a normal access. The input layer (which is added in front) will decide whether it is safe or not to do a stream access for a context, and in that case grant it, meaning it has to shift a read operation from the stream source's lane group to the stream destination's lane group. It also needs to stall caches that are coupled to contexts related to stream arbitration in certain cases.

The output layer (which is added at the end) will route back the granted stream output from the destination to the source if its a hit, and might need to stall lanes when their caches are stalled due to stream arbitrations.

As mentioned in Section 3.3.5, in this implementation we need to solve the problem of pre-routing arbitration. A stream access can only be forwarded from source to destination if no other lane group in the destination context is issuing a memory operation. This is slightly unfortunate, as the mapping between contexts and lane groups is done by the routing network itself, but as we are arbitrating in front of it we obviously cannot use this network. Calculating if a context is doing a memory operation or not has to be done fast in order to avoid penalising the critical path too much.

This calculation was done by inspecting the write and read enable signals of all lane groups that are coupled to a context. Let us define CTX_n as the set containing all lane

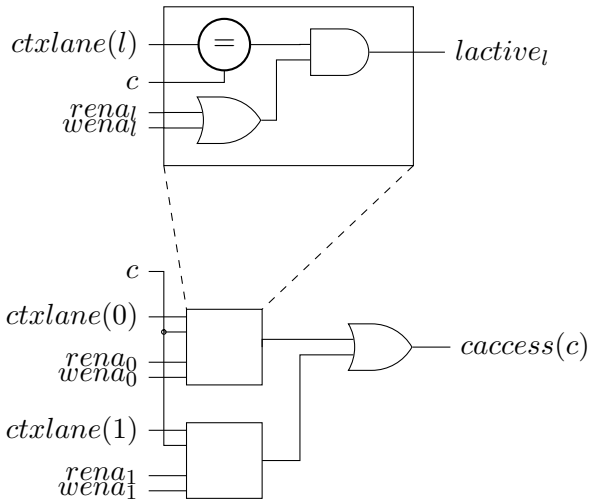


Figure 4.3: An example of the *caccess* circuit for context c when the design has two lane-groups.

groups that are coupled to context n, and $wena_n$ denotes the write enable signal of lane group n. Lastly, $ctxlane(n)$ is defined as a function that returns the context of lane group n. We can now define the signal $caccess_n$, which implies that context n makes a memory request as shown in Equation 4.4. An example instantiation of this circuit is shown in Figure 4.3, for context c (thus, for a real design, one needs to instantiate one copy of this circuit per possible context). In this example, the ρ-VEX core has only two lane-groups: more of the blocks in this circuit should be instantiated for each additional lane-group.

$$\forall_{(c \in CTX)}[caccess(c) = \sum_{l \in LG} ((ctxlane(l) = c) \cdot (rena_l + wena_l))] \tag{4.4}$$

Now that we have a definition to see if a context is doing a memory access, we can look at the arbitration ourselves. We want to grant an access when a lane group requests a stream and the context that holds the stream destination lane group is not doing any accesses. Conversely, we want to block an access if a request is done while the destination context is doing a memory access. When we have lane group n as our stream source, and we know that n is on a context boundary (which we know from Equation 4.3), that means that the stream destination is always in the previous lane group, and thus it is $ctxlane(n) - 1$. This is all combined in Equation 4.5, of which a circuit design is also shown in Figure 4.4 (for a single lane-group).

$$\forall_{(n \in LG)}[sgrant(n) = sreq(n) \cdot \overline{caccess(ctxlane(n) - 1)}] \tag{4.5a}$$

$$\forall_{(n \in LG)}[sblock(n) = sreq(n) \cdot caccess(ctxlane(n) - 1)] \tag{4.5b}$$

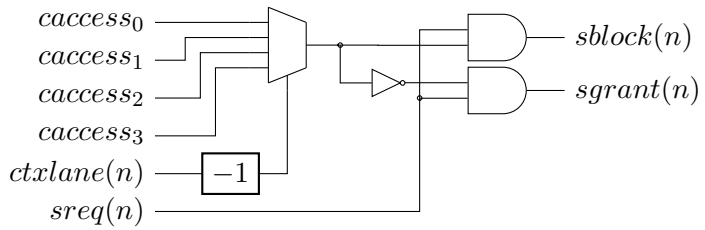


Figure 4.4: The grant and block signals, assuming a ρ-VEX implementation with at most four contexts. This circuit needs to be instantiated for each lane-group.

When a stream is granted, we must stall all the lane groups that are coupled to the destination context for one cycle. When a stream is blocked, we have to stall the lane groups that are coupled to the source context for one cycle instead.

Now we have a definition of when streams are granted and blocked, we can continue with the actual stream arbitration layers. The input layer shifts a read request by one if a stream is granted. Let us define the signals that were offered by the core to the routing network as $input_i(n)$, while the actual inputs of the routing network are called $input_o(n)$ (both n being the corresponding lane group). The original implementation was obviously such that $input_o(n) = input_i(n)$. Likewise, the output from the routing network that was offered back to the core is defined as $output_i(n)$, while $output_o(n)$ is the input of the core. Again, in the original implementation it would hold that $output_o(n) = output_i(n)$.

$sgrant(n)$ implies that lane group n wants to send a stream request to the cache of lane group $n-1$. This means that we need to send the read input of n to $n-1$ (or in other words $n+1$ to n). From the result, we should forward the output of $n-1$ to n, but only if it registers a hit. This is expressed in Equation 4.6. Circuit designs are shown in Figure 4.5 for the input layer and Figure 4.6 for the output layer.

$$\forall_{(n \in LG)}[input_o(n) = \begin{cases} input_i(n+1) & \text{when } sgrant(n+1) \\ input_i(n) & \text{otherwise} \end{cases}] \tag{4.6a}$$

$$\forall_{(n \in LG)}[output_o(n) = \begin{cases} output_i(n-1) & \text{when } sresponse(n-1) \cdot hit(n-1) \\ output_i(n) & \text{otherwise} \end{cases}] \tag{4.6b}$$

However, the result of the stream is available one cycle after it was issued. Not only that, the request is not necessarily issued when the read signal is high. *sgrant* is thus not usable to select the result. For this purpose we pass *sgranted* to the cache blocks, who latch this signal when when they accept a stream request, to make it align with the rest of the response. This signal (which we call *sresponse*) is sent back through the output network and is used instead of *sgrant* in Equation 4.6b.

4.5.2.1 Evaluation

Unfortunately, mid-development it was observed that this implementation had a significant impact on the critical path and prove to affect the maximum frequency at which

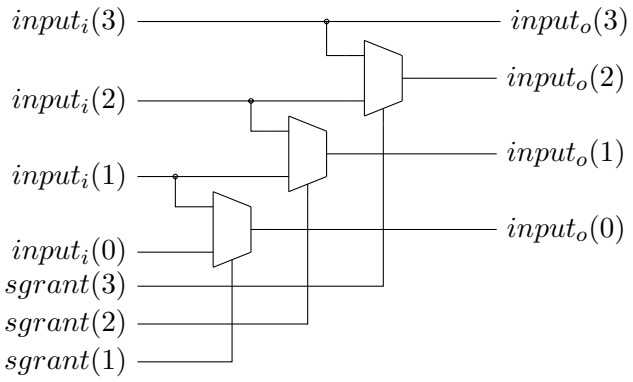


Figure 4.5: The arbitrating input routing layer that shifts granted stream-requests by one, in a ρ-VEX configuration with four lane-groups.

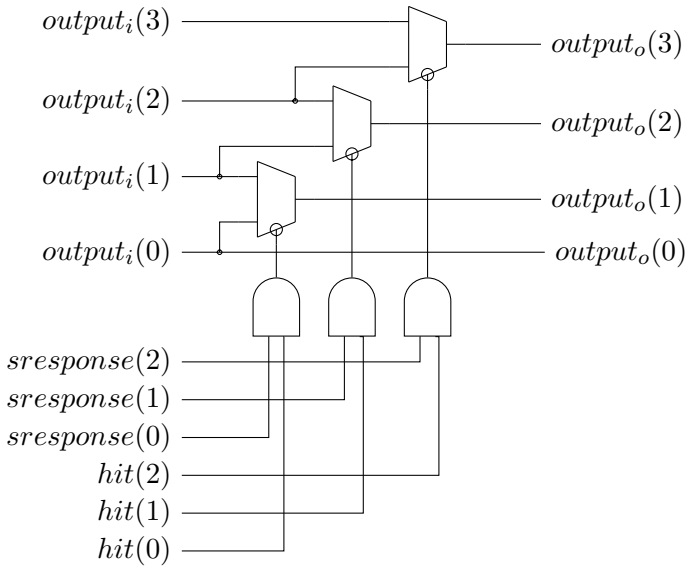


Figure 4.6: The arbitrating output routing layer that shifts granted stream-requests back by one, in a ρ-VEX configuration with four lane-groups (the multiplexer selector inputs are inverted).

the core could operate too much. Also, we are unable to stop or suspend accesses that are in progress externally. This means we block stream accesses when the block is already busy. This might not be what we want, because a stream access is generally much shorter than a normal access (so prioritizing it would be logical). It was thus decided to abandon this design in favor of the next implementation style, where we route additional streaming related signals inside the network and do either dual-port or at cache-block level arbitration next.

4.5.3 Cache Level Arbitration

In the next implementation we do not solve the arbitration before and after the routing network, we delay it to the cache blocks instead. This was done to reduce the additional logic levels added by the previous design, as we were modifying existing logic instead of adding more levels. Also, it would be possible to make streams interrupt existing transfers, which makes sense because they are finished quickly.

To do this, we add additional stream related signals to the routing network. For the input layer, a stream enable, stream address and stream stall signal were added. All of these signals were shifted once, in other words connected going from lane group n to input $n-1$ of the routing network. The first was simply connected to the stream request signal, the second was always connected to the requested address and the stall was always connected to the stall input signal from the lane group. In the output layer, a hit and latched enable were added. The stream enable, stream address, hit and latched enable signals serve an obvious purpose, and the need for a stalling signal is discussed later in Section 4.5.3.3.

Inside the input routing network nodes were added to also forward streaming related signals in the same way: a stream request would be forwarded to both outputs and the error state would be asserted if a stream exists on both inputs at the same time. Note that the normal read/write requests and stream request are thus routed separately, thus we can have both a read/write and stream on the same cache block. This is arbitrated inside the cache blocks themselves. The output network simply broadcasts a hit or stream hit.

$$\forall_{(n \in LG)}[data_o(n) = \begin{cases} data_i(n-1) & \text{when } streamhit(n-1) \\ data_i(n) & \text{otherwise} \end{cases}] \tag{4.7a}$$

$$\forall_{(n \in LG)}[data_o(n) = \begin{cases} data_i(n-1) & \text{when } streamhit(masterlane(n)-1) \\ data_i(n) & \text{otherwise} \end{cases}] \tag{4.7b}$$

Now we add a multiplexer on the data outputs of the cache by using the stream hit response, as shown in Equation 4.7a. The stall output from the cache back to the lane group is extended to also be released when a stream hit is registered. With this design, arbitration can be done by the separate stall signals, as long as we assure that a stream hit can only occur when a stream request is done.

4.5.3.1 Distributing Stall Signals

One fundamental issue is left to be solved, however. In the previous implementation, the routing network already was made aware of the context to lane group mapping and used this to stall all lane groups that are coupled to a lane group that loses arbitration. In this implementation we use the stream hit signal to arbitrate, but this signal only arrives to the lane group that issued the stream request. We need one final mechanism to distribute this hit signal to all other lane groups.

First we calculate for each lane what the coupled lane would be that can issue a stream request (the lowest indexed lane group). This is done again by observing the

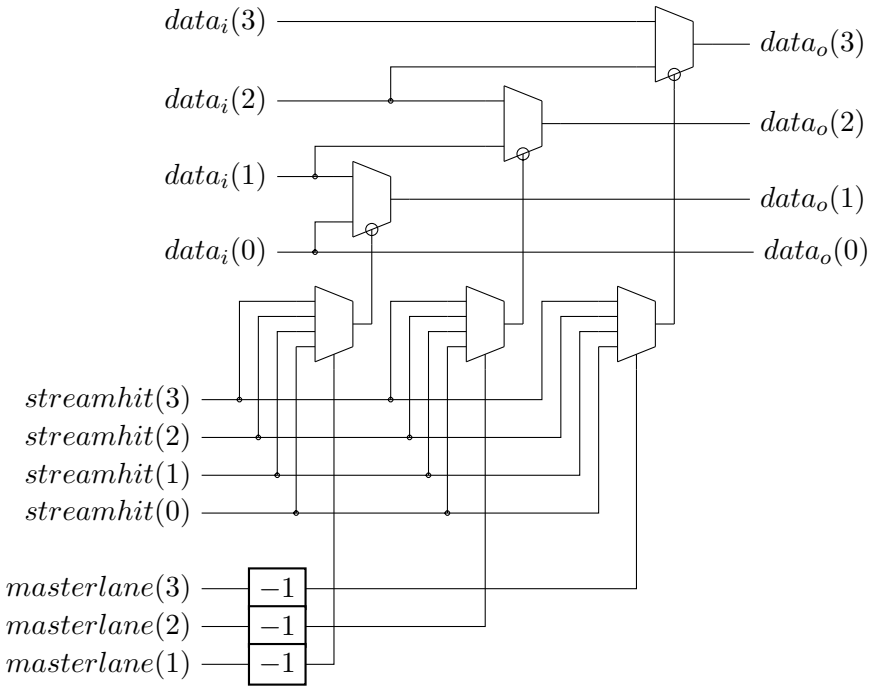


Figure 4.7: The input arbitration logic for the cache blocks for a ρ-VEX design with four lane-groups.

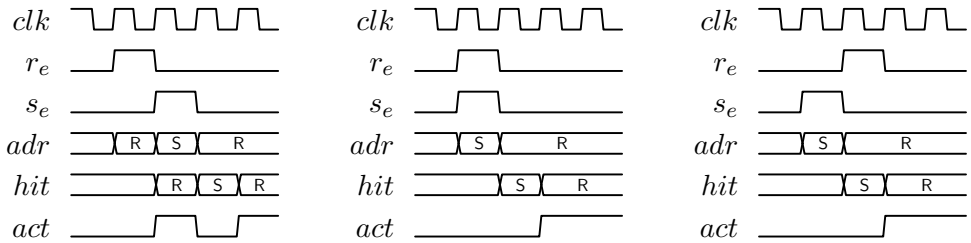
bits inside the decouple vector. This function is defined so that $masterlane(n)$ returns the lane group with the lowest index that is still coupled to lane group n. We ensure that this is not part of the critical path by calculating this when decoupled changes and storing the result in a register. Note that this might give incorrect results when a stream request is done directly the cycle after a reconfiguration. For every lane group, we can now simply observe the stream hit signal of its master lane. This gives us the updated Equation 4.7b. The corresponding circuit design is shown in Figure 4.7 for a ρ-VEX design with four lane-groups.

Now we have a setup for the overall cache system for the cache level arbitration design. In Section 4.5.3.2 we will continue with the actual arbitration process inside the cache blocks, while in Section 4.5.3.5 we will start from this point again but this time avoid arbitration as much as possible by using dual-port memories.

4.5.3.2 Arbitrating Data Cache Blocks

In Section 3.3.5 we already discussed the need for arbitration. The implementation in Section 4.5.2 relied on routing-level arbitration, but that required an additional layer in front and at the end of the routing network, which affected the critical path too much. In this implementation we chose to route the stream request through the network and solve the arbitration at the cache blocks. Since we should be able to determine a streaming hit in a single cycle, it should not be so difficult to arbitrate, in theory.

In practice however, it is complicated because of the current cache design. The



(a) Read before stream (b) In the same cycle (c) Stream before read

Figure 4.8: Stream and read signal timings for three different situations.

requested address is routed to each sub-block inside the cache block directly, while the main cache controller always gets signals delayed by one cycle, as shown in Section 2.7.2.2. The lane groups continue when they register a hit, while the main cache controller does nothing in hit cases. Only when a miss is registered, the main controller takes over. The problem with this is that we cannot implement the streaming arbitration in the main controller.

A streaming operation can be done in a single cycle: the address is issued to all blocks, and a hit and the data is forwarded the next cycle. This same cycle the original address can again be issued to all blocks who will thus resume normal operation one cycle later. Let us quickly summarize all the possible situations in Figure 4.8, where we assume a stream request takes priority over a normal. We distinguish three situations: one where a read is issued one cycle before a stream (Figure 4.8a), one where they are issued simultaneously (Figure 4.8b) and one where the read is issued one cycle after the stream (Figure 4.8). Here we denote read enable and stream enable as r_e and s_e, respectively. The letters R and S denote that the response of the read and stream are on either the address or hit lines, while the act signal denotes that the main controller is acting on a miss (if a miss was observed). The main controller reacts on the hit signal and the address delayed by one cycle, which is synchronized with *hit*.

We can observe two properties from this table:

- The read hit response can at most be delayed by one cycle: only if both are issued at the same time.
- The main controller needs to be paused for one cycle in the situation where a read is issued just before a stream.

Initially, the arbitration was implemented by delaying the stream enable signal one cycle, and using that as a clock enable signal for the state register inside the main cache controller. The original stream enable signal is used to select the correct address on the wire that goes to all blocks. This did however give rise to two more problems: some signals were asserted longer than expected (for example, validate would remain high even though the stream address is placed on the bus for a cycle, validating the streamed address by accident), and other signals were missed (the acknowledge signal sent by the bus to indicate a write or read finished remains high for only one cycle).

This implementation was refined by refining the main controller such that it performed only safe operations and transitions when streaming is enabled; further more any

signal that writes to any of the cache blocks is forced low when stream enable (delayed by one) is high and the acknowledge signal is stored in a register that ensures the main controller catches up.

4.5.3.3 Multiple Stall-sources

Up until now we brushed over the fact that streaming request signals are, just like normal read request signals, only valid when the issuing lane-group is not stalled. This means that the stall signal of the lane group that issues a stream is now also required to be routed to the cache block. We used the same approach as taken for the rest of the inputs, like discussed in Section 2.7.2.3, but of course by using the stream stall signal for the registers and multiplexer.

4.5.3.4 Evaluation

This proved to be our most promising implementation, but it was not without issues. Our test-cases were refined while evaluating, testing and debugging this design. Several difficulties encountered that suffered from a lot of bugs where tackling the multiple streaming sources as described in Section 4.5.3.3, distribution of the stall signals (Section 4.5.3.1) but also which states in the main controller should be stalled or not. This was strongly complicated because of our added hybrid write-back behavior, as there were a lot of additional possible accesses, for example the complexity that eviction could be caused by a write-through or write-back access, both behaving differently.

4.5.3.5 Dual-Port Implementation

The third proposed design relies on using dual-port memories instead. One would say that this is trivial since dual-port memories are a commonly available resource, and our targeted FPGA design has dual-port block RAMs available.

Remember that the ρ-VEX cache consists of several blocks:

- The data block that holds the cache line itself.
- The valid block which has a bit per cache line to indicate if it is valid or not.
- The dirty block (in our write-back design) which is another instance of the valid block, a bit that indicates if the block is dirty or not.
- The tag block which holds a tag per block and determines if the requested address is in the cache or not.

Only the data and tag blocks are traditional memories that can be mapped on a Block RAM (BRAM), unfortunately. The valid and dirty blocks are more complicated, since they require the ability to be reset/cleared entirely in a single cycle. This requires them to be built from sources other than BRAMs.

One more complication is that the both the valid and tag blocks are already used as a dual-port memory. The former is only used for writing (invalidation), while the latter is only used for reading (before invalidating a line, it should first be confirmed

that this line hits). Both setting a line to valid and reading the validity of a line are connected to the address bus just like the rest of the cache, but the invalidation logic (setting a line to invalid but also reading from the second hit port) is connected to an update address line generated by the main cache controller, because of the bus snooping mechanic. Fortunately this is not required in the dirty block, where both the validate (mark as dirty) and invalidate (clear dirty) operate on the current address.

Whether we can solve the first problem is mostly up to the synthesis tool, and if we can design a model that it can successfully synthesize. The second problem requires us again to arbitrate: either we arbitrate streaming hit/valid determination with the normal hit/valid determination, or we arbitrate streaming hit/valid determination with the invalidation logic. The latter seems like the best way to go since we are adding a second port to all blocks anyway, which will be connected to the streaming address. Furthermore, this allows the lane-group that is connected to this cache block (and the entire context linked to it) to continue execution as normal, since its own hit/valid determination logic remains un-arbitrated. The disadvantage is however that invalidations are caused by other sources than this lane-group or context. More specifically, the invalidation interface is connected to the bus in such a way that when a value is written to the bus it is invalidated in all caches automatically.

This implies that streaming has to be arbitrated with writing to the bus. Writing to the bus requires several cycles, and even though the invalidation requires only a single cycle it is still connected directly, thus it will keep invalidate high until the value has been written to the bus. If we modify the invalidation logic in such a way that it will only invalidate one cycle during a bus write, we free up all the other cycles for streaming. They are even guaranteed to be free, since the bus will not be available for another write (generating another invalidation) until the first write is finished. It is important to invalidate in the first cycle of a write, since all other caches must be informed of the invalidation as soon as possible (otherwise they might still use the data before the invalidation is performed).

To summarize, for the dual-port implementation we require the following:

- Expose the second port of the data block.
- Add read capabilities to the second port of the valid block (the invalidate port).
- Rewrite the valid block to correctly synthesize a dual-port memory with the ability to reset the entire memory in a single cycle.
- Change the invalidate logic to only invalidate in the first cycle of a write to the bus.
- Ensure that the stream can be stalled while an invalidation is performed on the same cache block.
- Add a multiplexer to the invalidate and second tag address ports that switches between the invalidate address and the stream address.

4.5.3.6 Evaluation

Unfortunately the synthesis tool would always crash if we tried to infer a second port on each block. This is most likely because of the valid block, since BRAMs should be able to use two ports quite naturally. It could simulate correctly, and seemed to behave as intended here. We were not sure if it is due to a badly or incorrectly written behavioral model, or due to a bug in the synthesis tool, of which the latter seemed a bit more unlikely to us since simulation accepted the model. Eventually, we decided to focus on the cache-level implementation first because of this uncertainty, abandoning this implementation before completing it.

4.5.4 Core to Core Streaming

Up until now we have always considered streaming from lane groups inside contexts, but there is no reason to limit ourselves to lane groups inside the same core. We have up until now always expressed all streaming request related signals in terms of lanes n and $n-1$. As an example, let us consider Equation 4.3 one more time. We defined $sreq(n)$ as a signal indicating that lane group n wants to stream read from $n-1$.

This means that if we want to issue a stream to another core, this stream should be done by lane group 0, and it should be issued to the last lane group of the other core: $|LG|$. Let us give an updated version that clarifies this issue in Equation 4.8a. Here we define new inputs $sreq_i$ as a signal that indicates the request done by lane group 0 is a stream request.

The other core is the one who should determine if this request is a stream request however, and for that it should have access to the read enable and address signals of lane 0. These inputs are defined as $rena_i$ and $raddr_i$. They are checked in conjunction with the stream enable, decouple and write-back region of the last lane group to calculate a stream enable signal response that is routed back to the other core: sen_o, as shown in Equation 4.8b.

$$\forall_{(n \in LG)}[sreq(n) = \begin{cases} sen_{n-1} \cdot dec_{n-1} \cdot rena_n \cdot raddr_n \in WB_{n-1} & \text{when } n > 0 \\ sen_i & \text{otherwise} \end{cases}] \quad (4.8a)$$

$$sen_o = sen_{|LG|} \cdot dec_{|LG|} \cdot rena_i \cdot raddr_i \in WB_{|LG|} \quad (4.8b)$$

Each core should thus also expose the read enable and read address signals of the first lane group as outputs, as shown in Equation 4.9a and Equation 4.9b. Similarly, we need to expose the stream outputs of the final lane group's cache to the other core, as shown in Equation 4.9c, Equation 4.9d and Equation 4.9e. A circuit diagram explaining this in more detail is shown in Figure 4.9 (assuming four lane-groups). Here, we see that lane-group zero of the left core shares it's read request signals to the right core. The right core determines if this is a streaming access to it's third cache or not, and performs a stream request to it's third cache block if this is the case. He routes both the stream request (as stream enable), stream hit and stream data back to the left core, who uses

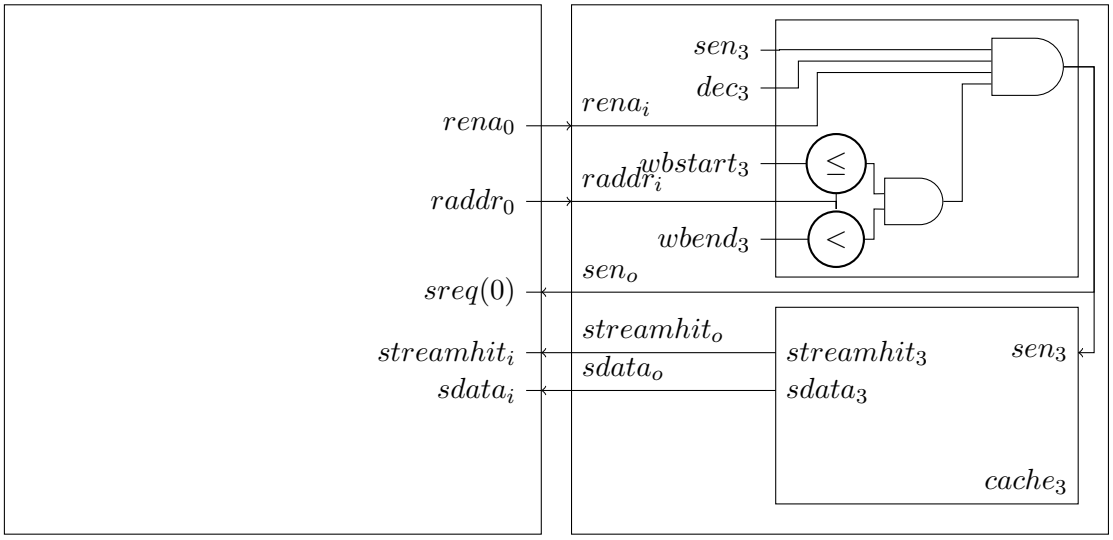


Figure 4.9: Two cores with core-to-core streaming connections, where they have four lane-groups. The left core shares it's read commands to the right core, who then has to calculate whether or not this read command is a stream request (for it's third cache block).

these signals to select the correct data with it's multiplexer.

$$rena_o = rena_0 \tag{4.9a}$$
$$raddr_o = raddr_0 \tag{4.9b}$$
$$streamhit_o = streamhit_{|LG|} \tag{4.9c}$$
$$sresponse_o = sresponse_{|LG|} \tag{4.9d}$$
$$sdata_o = data_{|LG|} \tag{4.9e}$$

The data selection multiplexer is updated as well, and we know that $masterlane(n) = 0$ when the data is being streamed in. This makes the multiplexer behave as in Equation 4.10, where $sdata_i$ and $streamhit_i$ are the core's streaming input data and hit signals, respectively. The circuit for a single lane-group is shown in Figure 4.10.

$$\forall_{(n \in LG)}[data_o(n) = \begin{cases} data_i(n-1) & \text{when } masterlane(n) > 0 \\ & \cdot streamhit(masterlane(n)-1) \\ sdata_i & \text{when } masterlane(n) = 0 \\ & \cdot streamhit_i \\ data_i(n) & \text{otherwise} \end{cases}] \tag{4.10}$$

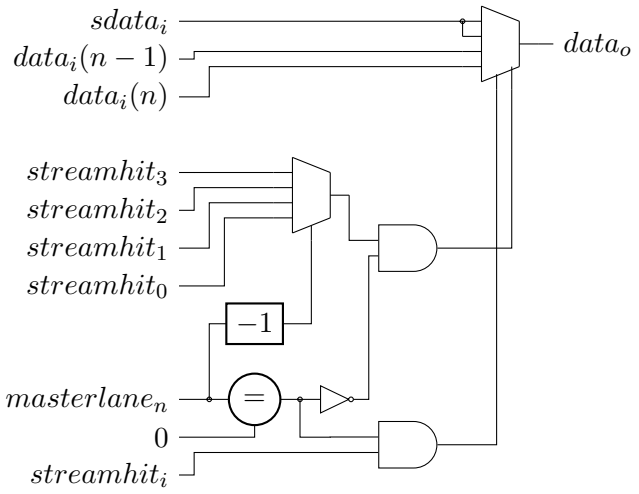


Figure 4.10: The streaming arbitration multiplexer updated with the external stream signals. When $masterlane(n)$ equals to zero, the external stream hit can override, while if it is non-zero the internal stream hit signals can override. This circuit has to be instantiated for each lane-group.

4.6 Consistency Recovery Controller

The last component that was designed and added is the so called consistency recovery controller. This component is activated when write-back is disabled and attempts to restore the memory and caches to a consistent state. For this the three strategies as described in Section 3.4.5 are implemented.

Unfortunately, the address that is routed to the tag, valid, dirty and data blocks is not generated by the main controller, but directly by the core. Also, the main controller should not interfere while consistency recovery is being executed. Therefore, the main controller and the consistency recovery controller have to be arbitrated.

4.6.1 Arbitrating with the Cache Controller

The consistency recovery controller has an override signal to signal to the rest of the cache that it is in control. This signal is used to pause the main controller and to override both the address that is offered to all the blocks but also the outputs to the memory bus with versions generated by this controller.

As the consistency recovery controller should not interrupt the main controller while it is doing an access, the main controller is responsible for actually activating the consistency recovery controller.

When a falling edge of the write-back enable signal is detected by the write-back and stream control register controller, it generates a single cycle signal to the main controller. The main controller has a set-reset latch that keeps it high until the request is granted. It will grant the request only from the idle state. The main controller should be paused the cycle after the request is granted (by the override signal of the consistency recovery

controller), or it shall resume normal operation again.

4.6.2 Consistency Recovery Strategies

Let us recall the three strategies that are defined in Section 3.4.5. These were flush, invalidate and none.

None is trivially implemented: the consistency recovery controller will do nothing and thus also not assert the override signal. This means the main controller will resume normal operation one cycle after granting the request.

The invalidate strategy is also fairly straightforward: it shall simply signal the dirty bit storage to clear all its bits. This can be done in a single cycle, and thus the consistency recovery controller has again no need to override and pause the main controller.

Only in the flush strategy we need to do a bit more work. It has to iterate over all addresses inside the cache, to determine if the line is dirty or not and then flush it back to the main memory. It will finally assert the override signal and clear an internal offset address counter. This offset address counter overrides the normal address internally used in the cache block to fetch the tag, valid and dirty bits. One cycle later, when this information is available, it uses the tag to reconstruct the full address of this cache line, and when valid and dirty, it constructs a complete write command to the main memory bus to write the data. When this is completed (or in the case that the line was not valid or dirty), it returns to the idle state if the offset counter is the last address. Otherwise it increments the offset counter by one and goes back to the tag fetching stage. This means that each address requires at least two cycles to handle, assuming we don't have to actually write it, meaning we require $2 \cdot |c|$ cycles in the best case to check the whole cache.

This controller could perhaps be optimized slightly by means of a pipeline: the full address can be latched allowing us to increment the offset address faster, so we can issue a new tag address to test each cycle (assuming it doesn't have to write).

4.7 Conclusion

In this section we discussed the implementation of both the hybrid write-through and write-back caches and the streaming features. First, the interface to the programmer was specified by defining the used control registers and their contents. For streaming, it was straightforward to insert a bit per context to allow streaming, while for write-back an address and size register was chosen (with several bits in the size register used to enable and to chose the consistency recovery mode).

After defining the interface, a simulation model was developed: the existing ρ-VEX simulator was extended to support these new features via the earlier discussed interface (and is thus forward compatible with the actual implementation). To implement this, the level of accuracy offered inside the simulator with regard to the cache lines was improved, and other parts of the cache code were re-structured as needed. Dirty bits were added, and the concept of eviction was added to allow the cache to function as write-back inside the write-back region. Streaming was then implemented straightforwardly by allowing the memory read function to look in the cache of a different context as well.

The simulator did not support multi-core setups, and thus multi-core streaming was not implemented.

In the hardware, a controller for the newly added control registers was added that converts the contents of these registers to lane-group specific signals that are offered to the cache. It also arbitrated between several contexts, who in theory can request a new streaming configuration at the same time.

The data cache's valid memory block was copied to act as dirty bit instead. The memory controller was extended to behave in a write-back fashion inside the write-back region.

A stream request detection circuit was added, ensuring that it only occurred on context boundaries.

Three different implementations were designed: routing network arbitration, cache level arbitration and a dual-port variant.

In the routing network implementation, arbitration was done before and after the routing network to ensure that streams could be routed as normal read requests. This implementation however had a big critical path penalty, but also suffered from feed-back loops when both the stream source and destination's stalling signals were influencing each other.

Next, the cache level arbitration implementation was made. Here, new signals were added to the routing network to accommodate streaming, allowing the network to route a read and stream read separately. When these signals arrived at the cache blocks, they would be arbitrated with normal accesses. To enable this, a mechanism to distribute the streaming hit signal over all lane groups coupled to the stream source was designed as well. This implementation had a smaller penalty on the critical path, but suffered from long delays since contexts are not always directly ready to read out the result of their memory access. What is even worse, the arbitration still suffers from bugs when conflicting accesses were done.

Finally a dual-port implementation was designed, in which we attempted to avoid the arbitration problem altogether. However, due to the invalidation mechanism, some form of arbitration was still required. This design could unfortunately not be made synthesizeable altogether.

The concept of streaming is next expanded to multi-core implementations, which was relatively simple to do by adding some additional signals.

The last part that was added was the consistency recovery controller, which has the functionality to override the main cache controller in order to write all dirty data back to the main memory when needed.

5 Verification

In the previous chapter, the final hardware implementation was presented. In this chapter we will first describe the experimental setup used. We will then verify the implementation for functional correctness by designing and running several test-cases, but also perform the hardware synthesis and run benchmarking applications to measure speed-up as our analysis.

We start off by describing the different test-setups used in the remainder of this chapter in Section 5.1.

In Section 5.2 we focus on functional verification by describing behavioral test-cases. We start off by writing test-cases in Section 5.2.1 that verify if the simulation model behaves as described in Section 4.2. Next, we define more extensive test-cases aiming the implementation as designed in Section 4.3 to Section 4.6. Firstly we target the write-back region in Section 5.2.2, followed by tests that target the streaming feature in Section 5.2.3. Several of these tests focus on corner cases and behavior that was abstracted away from in the model, like timing issues and arbitration. The last test-case is a larger pipeline model that is explained in Section 5.2.4.

The results of the test cases will be given in Section 5.3 and Section 5.5, for the simulation model and the implementation respectively, where we obtain pass or fail, but also measure performance. The execution of the simulation model serves both as a verification that conceptually both streaming and write-back caches work and as an initial measurement of an upper bound for our performance increase (as arbitration is not abstracted away from). The implementation tests extend behavioral testing with arbitration and conflict testing. Again, a pass/fail verdict is given, and performance measurements are done.

Section 5.4 will discuss synthesis results (resource utilization and maximum clock frequency) in an effort to analyze the cost-effectiveness of the implementations.

Finally, Section 5.7 will summarize all the results obtained in this chapter in order to evaluate the work and answer the research question.

5.1 Experimental Setup

In this section, we will first start with a description of the simulation platform, followed by an overview of all the platforms that were synthesized using the ρ-VEX as a core. The actual synthesis results (resource usage and maximum frequency) will be given later in Section 5.4.

5.1.1 Simulation Platform

The functional simulation model was implemented in sim-rvex. It was compiled and executed on an Intel Core i5-4278U processor, running at 2.60GHz, equipped with 8GB RAM, running the a GNU/Linux distribution (Debian Stretch).

5.1.2 Synthesis Platforms

A processor core by itself is not enough to execute code. We require a platform, in which the core is then placed as Central Processing Unit (CPU). In total, three existing ρ-VEX platforms targeting the *ml605* development board were used, each with a different goal. We will give a short overview of the three different platforms here, and discuss why we synthesized this platform.

- ml605-standalone: this is a relatively simple platform that, as its name implies, is stand-alone: just the core, with an optional cache, a memory controller and a debug Universal Asynchronous Receiver/Transmitter (UART) peripheral is synthesized on the board. This is the most simple design without any additional overhead to test all single-core features. Obviously, the optional cache was enabled in our setup. A single ρ-VEX core with an issue width of eight, and four contexts fitted on this platform comfortably.
- ml605-grlib-bare: in this version, the bare minimum is implemented using grlib[1] IP cores. Again, this design contains the ρ-VEX core (this time the cache is no longer optional), grlib memory controller, interrupt controller grmon JTAG debugging peripheral and again a UART debug peripheral. These components are connected as AMBA High-Performance Bus (AHB) slaves, while the ρ-VEX is a master. The main reason this design was synthesized was because it is the most simple design available that supports more than one ρ-VEX core (as AHB masters), and thus it served as our multi-core streaming test platform. Unfortunately, the board could not hold two eight-issue cores, so they were reduced to four-issue cores with two contexts each.
- ml605-grlib: this is a more complete grlib platform, with many AHB peripherals. This platform was chosen as it had an SVGA video controller, so tests could be performed with a streaming graphics pipeline. It was again synthesized with a single-core setup.

5.2 Testing for Errors

Before we measure anything, we should first test the functional behavior of the implementation. The question we ask in this section is simply: does it work as intended? One distinction has to be made: the simulator only implements functional correctness, and is not entirely accurate, especially when taking cycle-accurate timing into account. For

[1] Grlib is a library of Intellectual Property (IP) cores used in development of System on Chip (SoC) designs, developed by Cobham Gaisler, released under the GNU GPL License[42].

example, the contexts are not truly running at the same time, and conflicting memory accesses and the arbitration cannot be easily modelled. This means we have two levels of testing: first we shall test purely the functional behavior in the simulator, and after that we will also test conflicting memory accesses and such in the actual hardware implementation.

Descriptions of the test-case programs for the former are given in Section 5.2.1, while the latter are described in Section 5.2.2 and Section 5.2.3, for write-back and streaming respectively. Next, the results of these test-cases are listed in Section 5.3.1.

5.2.1 Functional Correctness

The first application (*sim-wb*) was written to test the write-back behavior in the simulator. It tests overlapping write-back regions, which is not recommended under normal operations because of the race conditions in consistency recovery. This program did the following, while working with a single memory location m, and every step used a barrier to synchronize all contexts:

- Start four contexts.
- Each context would in turn print the contents of m, write its core ID to it and print it again. This verifies that the memory works as normal, so it would print the following sequence: $[\upsilon, 0, 0, 1, 1, 2, 2, 3]$ (where υ is an undefined value since the memory is not yet initialized).
- All contexts would activate m as their write-back region, with the flush strategy.
- Each context adds $0x42$ to their context ID and writes that to m.
- Each context prints the value at m in order. Now we can see the effect of write-back memory: if no write-back was activated all contexts would print $0x45$, but with write-back the sequence $[0x42, 0x43, 0x44, 0x45]$ should be printed.
- Now, the contexts in turn disable write-back. Because the first context drops first, his flush will be most likely to be done first, invalidating all other values.
- Again, each context prints the value at m in order to verify if they are indeed $[0x42, 0x42, 0x42, 0x42]$.

The second application (*sim-stream*) tests streaming, again by using a single memory location m and barriers to synchronize steps:

- Start four contexts.
- They all activate write-back with m in the region.
- Each context adds $0x42$ to their context ID and write it to m, and print it. This again should print the sequence $[0x42, 0x43, 0x44, 0x45]$.
- The first context enables streaming for all other contexts.

- All cores again read the values in order. Since the first context doesn't have streaming, it should print $0x42$. If streaming works, we should see the following sequence: $[0x42, 0x42, 0x43, 0x44]$, while if it fails we see $[0x42, 0x43, 0x44, 0x45]$.
- Similarly to the previous test, write-back is disabled and now all cores should see the value $0x42$ at m.

5.2.2 Testing the Write-back Implementation

Now that we observed that the concepts function as intended in the simulator model, the next step is to test the implementation in a similar fashion but also trying to test corner cases in the arbitration. For this purpose, a total of twelve applications were written. We will start with the write-back tests, of which there are four. Each test one aspect of write-back behavior: writing, evicting and two of the consistency recovery strategies (flush and invalidate).

5.2.2.1 Writing

The first, *wb-write*, simply tests if writes to the write-back region work correctly. It tests both a normal loop and a partially unrolled loop, so it covers normal write loops and also attempts to catch fast consecutive writes. This test only uses a single context.

- It sets the write-back region to 1024 words (exactly the default cache size).
- First it does several incrementing sequential writes from 0 to 511.
- After that it loops over this range again, reading to see if any write failed. If this is detected, it writes an error code to the UART.
- Next, it writes a range of 512 to 1023 to the write-back region in a partially unrolled loop: each iteration has four writes (one for each context).
- Similarly, it checks all the values afterwards and prints an error code if failed.
- In the end it will write a success code to the UART, if no error was observed.

5.2.2.2 Evicting

The second test, *wb-evict*, serves to test evictions in the write-back mechanism. To force an eviction, we make use of the fact that the cache behaves as directly mapped when writing, and thus any write at any address modulo the cache size will map to the same cache line, or in other words $x + n * |c|$ will always write to cache block x mod $|c|$. It uses an array to contain $3 * |c|$ integers. In the next test we will speak about relative offsets inside this array. We denote the state of the memory as m, and the state of the cache as c. A write of value v to offset a can be can be stored in the cache or in the memory, and an example where it is written to the memory is noted as $m = [a \mapsto v]$. Since in the cache, any offset with a multiple of $|c|$ will map to the same cache line. To clarify this, we will denote a write of v to $a + n|c|$ to the cache as $c = [a_n \mapsto v]$ (thus a_n for any n will map to the same cache line a).

The test consists of both a short and extensive test. First let us discuss the short test.

- The write-back is set to $2 * |c|$ (with no consistency recovery), so we can test evictions by write-back and non write-back writes.
- A value ($0x42$) is written to an arbitrary variable a inside the write-back region, so $m = [a \mapsto v]$ and $c = [a_0 \mapsto 0x42]$. It is also directly written back to verify that writing actually works, and will print an error code if not.
- Next, we write a new value ($0x1337$) to $a + |c|$, forcing an evict of the previous line. This should make the state $m = [a \mapsto 0x42]$ and $c = [a_1 \mapsto 0x1337]$. We test a read to see if the cache is updated, and print an error code if not.
- Next, we again try to read a. The cache block that would hold a currently holds the value of $a + |c|$, thus this value is evicted and the correct value is fetched. This should lead to the following state: $m = [a \mapsto 0x42, a + |c| \mapsto 0x1337]$ and $c = [a_0 \mapsto 0x42]$. If the read operation did not read $0x42$, an error is printed.
- Again a new value is written to a ($0xCAFE$). Since this line is already in the cache, this simply updates it to hold $m = [a \mapsto 0x42, a + |c| \mapsto 0x1337]$ and $c = [a_0 \mapsto 0xCAFE]$. Since a differs in m and c, the line should be marked dirty.
- Next, test an eviction by a write outside of the write-back region, by writing $0xBABE$ to $a + 2|c|$. This should evict the value at a again to the memory, but also write the value to the memory directly since $a+2|c|$ is not inside the write-back region. The state should now be $m = [a \mapsto 0xCAFE, a + |c| \mapsto 0x1337, a + 2|c| \mapsto 0xBABE]$ and $c = [a_2 \mapsto 0xBABE]$.
- Now, write-back is disabled and the locations a, $a + |c|$ and $a + 2|c|$ are tested. If all values are as they should be, we report a success code.

The extensive test operates on a range defined by a value r such that $r < |c|$.

- Firstly it fills up the entire cache with values ranging from 0 to $|c|$.
- Next, it reads an arbitrary value to confirm writing and reading works.
- Next, it writes more values to a range from $|c|$ to $|c| + r$. This should evict the first r values.
- We then read the value at $|c| + r - 1$ and test if it is correct.
- Lastly, we also check the value at $r - 1$, since this should have been evicted.

5.2.2.3 Flushing

This test (*wb-flush*) uses two contexts and is fairly straightforward. The first context writes several values to its write-back region and then disables it all together, enforcing a flush. The second context will then read to verify that all the data has been flushed.

5.2.2.4 Invalidating

This test (*wb-invalidate*) uses three contexts so that we can read the state of the memory twice.

- The first context writes two known values to locations a and $a+1$, thus the state is $m = c = [a \mapsto 0xCAFE, a+1 \mapsto 0xDEAD]$.
- Next, it enables two additional contexts activates write-back for $2 * |c|$.
- After it will write new values to a and $a+1$, thus the state will now be: $m = [a \mapsto 0xCAFE, a+1 \mapsto 0xDEAD]$ and $c = [a_0 \mapsto 0xBABE, a_0+1 \mapsto 0xBEEF]$. Since both values have different memory and cache values, they are both marked as dirty. This read is tested and an error code will be printed if the reads did not work.
- Next, the first value is evicted by writing a new value to $a + |c|$, making the state: $m = [a \mapsto 0xBABE, a+1 \mapsto 0xDEAD]$ and $c = [a_1 \mapsto 0x1337, a_0+1 \mapsto 0xBEEF]$.
- Write-back is disabled, and thus the dirty bits should have been cleared. a is read, and should not be $0xBABE$, unless the dirty bit was not cleared and it was thus evicted.
- Control is now handed over to the second context, who will test if $a+1$ has correctly been evicted (it should not be equal to $0xDEAD$).
- After this, we test if the dirty bit is still high by writing to $a + 1 + |c|$. If the dirty bit was not cleared, this will cause an evict.
- Next, the third context will perform the same test, the memory should again not be equal to $0xDEAD$.

5.2.3 Testing the Streaming Implementation

Streaming was tested in several different ways, ranging from simple streaming situations to more complex situations where correct arbitration is critical. Several of these tests were also again repeated in a multi-core streaming platform.

5.2.3.1 Simple Streaming

The first test (*str*) is straightforward and executes as follows. The first context writes a value ($0x42$) to location a in its write-back region and passes control over to the next context. Each context stream-reads the value at a, copies this to their own write-back region and increments it by one (and prints it), after which it passes along control to the next. We will thus observe the following sequence: $[0x42, 0x43, 0x44, 0x45]$. A multi-core version of this test is also written (*str-multi*) that works with four contexts spread over two cores.

5.2.3.2 Streaming Large Regions

This test (*str-large*) tests larger streaming blocks by letting the first context write a large block of items, signaling the second context who then reads and sums them all to compute a checksum. If the checksums don't match, an error code is printed.

5.2.3.3 Streaming Conflicts

The second test (*str-conflict*) works with two contexts, who will write and read at the same time to ensure we have conflicts and arbitration. A multi-core version was also written (*str-conflict-multi*) that spread the two contexts over two cores.

- The first context enables the second, enables streaming and write-back.
- Next, the first context will fill half of its test size with values, after which it will wait for a barrier with the second context.
- Once the second context is ready, he will signal the first context and start reading the first half.
- The first context will fill the second half after it receives the signal from the second context. In effect, the second context copies the first half while the first context is writing the second half, thus they are doing a lot of memory operations at the same time.
- After this is completed, the second context will sum all items to compute a checksum and print an error code if the checksum does not match.

5.2.3.4 Streaming Conflicts in Larger Regions

The previous test was also extended to a test where the data-set is split in four parts (*str-conflict-large*). In this test, the second context reads the first part while the first context is busy writing the second block and so on. What is more, this test was performed such that context two copies the data to its own write-back region before summing, but also a test was performed where it was directly summed (as a write to the write-back region takes a single cycle and is less likely to conflict).

5.2.3.5 Streaming while Bypassing

For this test (*str-bypass*), a similar setup was designed to test if streaming would still work while the first context was also performing bypass accesses. To do this, the first context writes a number of items to its write-back region, signals the second context to start and writes to the UART registers. While the first context is writing to the UART registers, the second context stream-reads the values and sums them as a checksum. If the checksum does not match, an error is printed to the UART.

5.2.4 Streaming Pipeline Benchmarks

We have just discussed several test-case applications that were specifically developed to test functionality. Even though their speed-up is measured, these test-cases are however slightly synthetic and fairly small, thus do not really serve as good benchmarks. More realistic applications are required as well, that will provide us with a better clue about the speed-up to expect in the wild. In total, three testing benchmark applications were designed. The first one is left fairly generic so that the result can easily be verified even in the simulator. The latter two are more specific pipeline implementations that operate on visual data, and are intended to run on the actual hardware. We will discuss each in more detail in this section.

5.2.4.1 Streaming Processing Benchmark

The first benchmark application was modeled to represent a generic computational software pipeline. Contexts communicate via queues of command-value pairs, where each context consumes commands from their neighbor's queue and produce commands (while modifying the value), placing them in their own queue. A special stop command is used to terminate the pipeline and application.

The application is designed to work in a multi-core fashion, even without streaming, and thus the queues are implemented using rotating buffers of fixed lengths. A static, fixed length buffer is used to store values. Two counters, a start and stop counter, are used to indicate the first and last values in the queue. When adding a value to the queue, one puts it at the stop index and increments the stop counter (modulo the buffer size). When removing a value from the queue, one takes the value at start and increments the start counter (modulo the buffer size). We can compare start and stop to find out if the queue is empty or full. The advantage of using this data structure is that the consumer only modifies the start counter while the producer only writes to the stop counter, making this thread-safe.

The first context generates a number of commands followed by a stop. The other contexts spin-lock until they input queues are not empty. Then, each context consumes a command and if it is not a stop command increment the value by one. The last context computes the sum of all its input values, and at the stop command will compare it with the expected checksum value.

This application is run in a non-streaming and streaming fashion, where each queue is placed in a different, non-overlapping write-back region, and streaming is enabled. We test streaming reads (where each context stream-reads the start and stop counters and queue entry) but also tests the fact that the stop value is incremented and correctly invalidated by a core that streamed the value (and thus re-fetched by the core that actually "owns" the counter).

5.2.4.2 Image Processing Pipeline

The second benchmarking application resembles an image processing pipeline. An image processing pipeline typically consists of operations that transform an image into another

image, or extract certain features from it. A larger pipeline is constructed by chaining several of these smaller operations after each other.

Often this operation is done via convolution. Convolution is an operation where a block of neighboring pixels is used to generate a new pixel via a convolution matrix, often called a kernel.

In the image processing benchmark, a total of two convolutions is performed: a blurring operation and edge detection convolution. This is a quite common pipeline: edge detection is useful for many applications, but also very noise sensitive which in turn can be reduced by a blurring operation. Note that these kernels are defined for a single color channel. Our input image is in color, so we first convert our image to gray-scale. The two kernels are applied by two different contexts, who stream their entire output buffer. The end result is finally converted to an image format accepted by the frame-buffer (5R6G5B).

Gray-scale Conversion Many gray-scale algorithms are known, producing different effects[43]. Our image consists of three linear color spaces R, G, B. We convert our image to a single gray-scale channel, thus we can consider this value the intensity of the pixel, denoted by I. The naive implementation is by using an average, such as shown in Equation 5.1a. However, this does not do justice to the nature of visual perception very well, since the human eye does not perceive a linear color space. An alternative is Luminance, which uses a weighted average closer to human perception, shown in Equation 5.1b.

Another interesting method for gray-scale conversion is derived from composite color video signal standards used in television, which were expressed in YUV or YCbCr color spaces. The Y (luma) component contained gray-scale information, while the other components added color to it in such a way that old black-and-white televisions would still be able to render the image correctly too[44]. Different standards had slightly different weights for an RGB conversion to luma, for example the ITU-R BT.709 standard for HDTV[45] as shown in Equation 5.1c. To avoid excessive fractional multiplications which are costly in the current set-up, a look-up table is initially constructed for all the used weights (since the R, G and B values are single bytes, these tables only require 255 entries).

$$intensity = \frac{1}{3}(R + G + B) \tag{5.1a}$$

$$luminance = 0.3R + 0.59G + 0.11B \tag{5.1b}$$

$$luma_{709} = 0.2126R + 0.7152G + 0.0722B \tag{5.1c}$$

5.2.4.3 3D Graphics Rendering Pipeline

The third benchmark is a large software pipeline that renders 3D graphics, inspired by the OpenGL graphics pipeline. It uses a queue system similar to the pipeline described in Section 5.2.4.1, to ensure all commands arrive in order. This benchmark is larger and more complicated, for more details we refer to Appendix A.

Section	Testcase	Platform	Pass/Fail
5.2.1	sim-wb	sim-rvex	Pass
5.2.1	sim-stream	sim-rvex	Pass
5.2.2.1	wb-write	standalone	Pass
5.2.2.2	wb-evict	standalone	Pass
5.2.2.3	wb-flush	standalone	Pass
5.2.2.4	wb-invalidate	standalone	Pass
5.2.3.1	str	standalone	Pass
5.2.3.1	str-multi	glib-bare	Pass
5.2.3.2	str-large	standalone	Pass
5.2.3.3	str-conflict	standalone	Pass
5.2.3.3	str-conflict-multi	grlib-bare	Pass
5.2.3.4	str-conflict-large	standalone	Pass
5.2.3.5	str-bypass	standalone	Pass

Table 5.1: Test results from the functional tests cases.

5.3 Simulation Measurements

In this section we discuss results measured with the simulation model implementation as discussed in Section 4.2. Firstly, we give results from the test-cases defined in Section 5.2. Next, a test benchmark application was developed which uses streaming extensively. We measured the amount of cycles that the simulator took to finish the program without and with streaming, and this is used to validate the idea of streaming caches.

5.3.1 Test-Case Results

In this section we summarize the results of the tests defined in Section 5.2.1, Section 5.2.2 and Section 5.2.3. The different target platforms (excluding sim-rvex) are discussed in Section 5.1.2. The results are shown in Table 5.1.

5.3.2 Computational Pipeline Results

Here we provide test results measured in the benchmarking pipeline that was defined in Section 5.2.4.1. We used this test to verify that the concept works in a larger, more realistic scenario, and as a measurement to give a rough idea of what kind of speed-ups we might expect. The simulator does currently not support measuring the amount of cycles stalled due to writes (as the write-buffer is not simulated), so the numbers are not completely accurate.

As the streaming version requires some additional setup, next to the reported total cycle count we counted the amount of cycles spent inside the main loop (by using the cycle count register). These numbers are displayed in Table 5.2. Here we can see roughly factors of 1.18 (total) and 1.25 (without overhead) difference in the total number of cycles.

For a better insight we also listed some additional performance counters in Table 5.3. Here we can see the amount of misses and total accesses (for reads and writes), but also

Not streaming		Streaming	
Reported	In loop	Reported	In loop
277969	214709	235177	170670

Table 5.2: The amount of reported total number of cycles and the amount of cycles measured inside the main loop, for both the streaming and non-streaming implementations. Lower is better.

	Not streaming				Streaming				
	Reads		Writes		Reads		Writes		
	Misses	Total	Misses	Total	Misses	Total	Misses	Total	Bypasses
Ctx 0	561	11834	1235	5541	670	17706	1236	5542	0
Ctx 1	2685	15034	1012	3013	984	16208	1013	3015	4007
Ctx 2	3021	15144	1012	3013	1022	30306	1013	3016	4110
Ctx 3	2041	25666	92	1256	36	3099	93	1260	12579

Table 5.3: The amount of read- and write misses, read- and write accesses and the amount of accesses bypassed by streaming per context.

the added counter that counts accesses bypassed via streaming (times when we had a streaming hit that was a miss in our own cache) per context.

We can observe a few things in this data. The amount of write misses and accesses remains more or less the same. However, the amount of read accesses increase for most contexts. This can simply be due to the fact that the cores have to do additional setup, although the significant drop of read accesses done by context three seems a bit odd.

We can also see the amount of read misses decrease dramatically, where context three even has almost no misses left at all. Keep in mind that context three only consumes packets and never produces any and that there is no context consuming his packets. When adding packets to the queue, contexts have to compare the start and stop pointers to conclude that there is space left in the queue. Everytime when contexts one, two and three consume packets, they modify the start counter of contexts zero, one and two, forcing an invalidation, read miss and re-fetch for them. This might explain why context three has so little read misses.

One more test was performed when using arbitration instead of assuming a dual-port setup. When designing the simulator it was not yet sure if the source or destination context would have priority in arbitration, and at the time it was decided to make the original access hold priority over a stream (which is unfortunately the opposite behavior of the final implementation). As the simulator does not really simulate stalling but merely the cycle count of a stall, we cannot determine the behavior with 100% accuracy. The simulator was once more modified by adding another register that would count every time a stream was performed in the same cycle as the stream destination was accessing its cache.

The results are shown in Table 5.4. We can see that context 3 is hindered the most by arbitration. This seems logical, as this context is not writing that much and is stream reading most of the time. At first these numbers might seem drastic, but let us analyze

	Arbitration losses (cycles)
Ctx 0	0
Ctx 1	435
Ctx 2	197
Ctx 3	2197

Table 5.4: Arbitration loss cycles per context.

what an arbitration loss actually means. We attempt to read at the same time as our stream source is performing a cache access.

While the results seem very promising, we have to mention the fact that this is again a simulation that is not entirely accurate. An arbitration loss has much more effect on the execution than just a single penalty cycle. Not only that, this pipeline is computationally very simple. This means that most effort (cycles) is put in the data transfers. This makes it only suitable as an upper bound situation, where the pipeline is bottlenecked by data transfers.

5.4 Synthesis Results

Merely testing for correctness in simulations is not enough to conclude correct behavior. The design needs to be synthesized to an Field Programmable Gate Array (FPGA) as well, so an actual implementation can be tested for correctness. Of-course, measuring the speed-up is also easier on an implemented design, and synthesis can give us several interesting metrics relating to cost.

5.4.1 Resource Utilization and Frequency

Earlier in Section 5.1.2 we mentioned the three synthesis platforms: ml605-standalone, ml605-grlib-bare and ml605-grlib. On each of these three platforms we synthesized two single-core eight-issue versions: one without write-back or streaming which acts as our baseline, one with write-back only and one with both write-back and streaming. We also synthesized the dual-core four-issue version on the grlib-bare platform.

The area utilization and maximum clock frequency are shown in Table 5.5. In this table we shortened the names of the platforms ml605-standalone, ml605-grlib-bare and ml605-grlib to sa, grb and gr, respectively. The issue lane configuration is appended to the name: 1x8 denotes a single core with an issue-width of eight, while 2x4 denotes two cores with issue-widths of four each. The implementations baseline, write-back only and write-back plus streaming are named BL, WB and WB & S, respectively. Area results are also shown in Figure 5.1, and the percentage of the total used is shown in Table 5.6.

5.4.2 Cost-Effectiveness Analysis

As expected, the amount of BRAMs does not change with the implementations, since the only extra storage (the dirty bit) cannot be mapped to BRAMs due to the flush

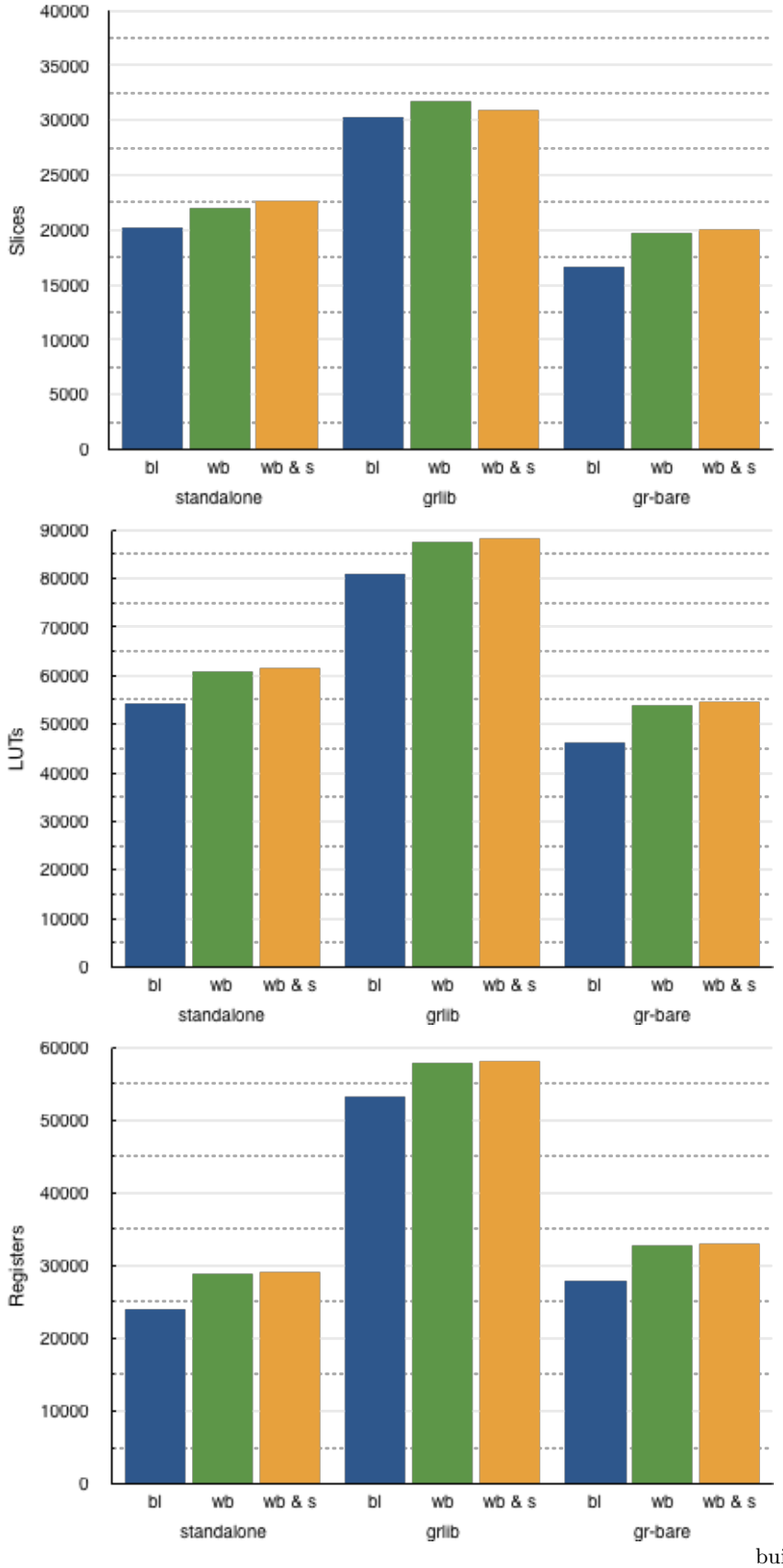


Figure 5.1: Slices, LUTs and register results from synthesis.

Design		Slices	LUTs	Registers	BRAMs	DSPs	Frequency (MHz)
total available		37680	150720	301440	1248	768	
sa-1x8	BL	20290	54325	24082	247	16	37.5
	WB	21937	60837	28979	247	16	37.5
	WB & S	22687	61470	29139	247	16	37.5
gr-1x8	BL	30207	80904	53097	213	16	37.5
	WB	31746	87491	57945	213	16	32.432
	WB & S	30913	88331	58112	213	16	32.432
grb-2x4	BL	16612	46391	27901	114	16	20
	WB	19683	53762	32865	114	16	20
	WB & S	20026	54567	32966	114	16	20

Table 5.5: The resource utilization of each platform when synthesized on the ml605 board.

Design		Slices %	LUTs %	Registers %	BRAMs %	DSPs %
sa-1x8	BL	53.85	36.04	7.99	19.79	2.08
	WB	58.22	40.36	9.61	19.79	2.08
	WB & S	60.21	40.78	9.67	19.79	2.08
gr-1x8	BL	80.17	53.68	17.61	17.07	2.08
	WB	84.25	58.05	19.22	17.07	2.08
	WB & S	82.04	58.61	19.28	17.07	2.08
grb-2x4	BL	44.09	30.78	9.26	9.13	2.08
	WB	52.24	35.67	10.90	9.13	2.08
	WB & S	53.15	36.20	10.94	9.13	2.08

Table 5.6: The relative resource utilization of each platform when synthesized on the ml605 board.

mechanic. This is supported more by the fact that the amount of registers increased significantly: we see an increase in registers of 16.9%, 8.4% and 15.1%, respectively for standalone, grlib and grlib-bare platforms. Likewise, the amount of LUTs increased by 10.7%, 7.5% and 13.7% for the same respective platforms. The relative cost for the grlib platform seems logical, since this platform includes a lot more components outside of the core, so the increase is less significant.

By comparison, the increase in resource usage from the write-back design to include streaming is a lot smaller. Again, this is expected, since mapping memories in logic is relatively expensive, while the streaming feature only adds logic. We can conclude that write-back is relatively more expensive to add than streaming.

The loss of maximum frequency on the grlib platform is significant, and this might affect our overal result. It is interesting that the maximum frequencies of the standalone and grb-2x4 platforms are not affected. Two possible explanations can be given: either these platforms had a different critical path, or there overal larger size of the grlib

platform made it more difficult to route efficiently.

We assumed there to be a large path introduced in core-to-core streaming, which could become the new critical path. This is not observed from the synthesis data, however, since there is no difference in maximum frequency between the baseline and streaming designs on the grb-4x2 platform.

5.5 Implementation Measurements

The same pipeline as used to test the simulator for an upper bound was tested on the hardware implementation (on both standalone and grlib). Unfortunately, it suffered from several bugs. The standalone version would only work with two contexts, if we force the second context to wait until the first is finished and a speed reduction was measured. The grlib implementation measured a speed-up and supported four contexts, but all contexts had to wait until the first finished generating the entire queue.

The cycle counts have been measured inside the main loop, and for completeness the values of the active cycles and stalled cycles per context have been added as well. The test was performed with 32 and 128 packets. For the standalone platform, the results are shown and discussed in Section 5.5.1, while the results for the grlib platform are discussed in Section 5.5.2.

5.5.1 Standalone

We will take a look at the results from the standalone platform. The measurements are shown in Table 5.7 (for 32 packets and 128 packets in Tables 5.7a and 5.7b, respectively). Again, we would like to stress the fact that on this platform, the program only executed correctly when using two contexts, and forcing the second to wait until the first completes. This is a clear indication that arbitration is not yet working perfectly when conflicting accesses are done to the same block, which we are simply avoiding by forcing the wait. What is more, a slowdown is observed in both the in-loop time and overall time. While the latter is expected since we have more set-up, the former is unexpected. We can only conclude that our arbitration protocol is doing more harm than good.

5.5.2 Grlib

Next, we take a look at the grlib results, in Tables 5.8 and 5.9 for the two- and four context implementations respectively. Again, we had to wait for the first context to finish before we could start the other one (or three). We can see a slight increase in cycles for the 32-packet implementations, and a slight decrease of cycles in the 128-packet implementations. The effect seems stronger in the four context implementation. This seems to point at some form of overhead which is compensated for when streaming more often (probably again due to our arbitration), even though we already try to filter out all forms of overhead by using only an in-loop counter.

Test with 32 packages

Context	Not streaming			Streaming		
	In loop	Active	Stalled	In loop	Active	Stalled
	107467			156032		
ctx0		214420	208215		244077	237814
ctx1		231432	217534		284223	279755

(a) Test results for a 32 packet pipeline on the standalone platform.

Test with 32 packages

Context	Not streaming			Streaming		
	In loop	Active	Stalled	In loop	Active	Stalled
	107467			156032		
ctx0		214420	208215		244077	237814
ctx1		231432	217534		284223	279755

(b) Test results for a 32 packet pipeline on the standalone platform.

Table 5.7: Test results for the standalone platform, using two contexts. All numbers presented are in cycles.

Context	Not streaming			Streaming		
	In loop	Active	Stalled	In loop	Active	Stalled
	4032			4143		
ctx0		9443	2872		9773	3166
ctx1		83022	30925		83011	31257

(a) Test results for a 32 packet pipeline.

Context	Not streaming			Streaming		
	In loop	Active	Stalled	In loop	Active	Stalled
	12723			12278		
ctx0		14003	3160		14355	3452
ctx1		83021	28633		83019	28397

(b) Test results for a 128 packet pipeline.

Table 5.8: Test results for the grlib platform, using two contexts. All presented numbers are in cycles.

5.5.2.1 Cache Performance Counters

The grlib platform was also compiled with cache performance counters. We included the data cache measurements from the four-context 128-packet implementation in Table 5.10. The non streaming measurements are shown in Table 5.10a, while the streaming variant is shown in Table 5.10b.

We can clearly see a reduction in the reading miss rate for contexts 1 and 2 (from 5.3% to 0.5% and 2.6% to 0.3% respectively), who benefit quite a lot from streaming. Context 2 seems to do less read accesses overal as well. Unfortunately, context 3 has a slight increase in its read miss rate which (from 39.8% to 43.5%).

Context	Not streaming			Streaming		
	In loop	Active	Stalled	In loop	Active	Stalled
	12478			12586		
ctx0		10798	4027		11599	4799
ctx1		7198	2652		7918	4270
ctx2		10707	2718		11126	4787
ctx3		83017	29265		83019	31475

(a) Test results for a 32 packet pipeline.

Context	Not streaming			Streaming		
	In loop	Active	Stalled	In loop	Active	Stalled
	39501			36936		
ctx0		15934	4315		16562	4914
ctx1		21167	3709		20749	4276
ctx2		33507	3775		32021	4743
ctx3		83019	20153		83016	22244

(b) Test results for a 128 packet pipeline.

Table 5.9: Test results for the grlib platform, using four contexts. All presented numbers are in cycles.

Block	Reads			Writes		
	Misses	Accesses	Miss rate	Misses	Accesses	Miss rate
0	7	520	0.0134	460	719	0.6398
1	137	2584	0.0530	848	1251	0.6779
2	143	5576	0.0256	135	394	0.3426
3	5291	13310	0.3975	50	182	0.2747

(a) Performance counters for the non-streaming implementation.

Block	Reads			Writes		
	Misses	Accesses	Miss rate	Misses	Accesses	Miss rate
0	7	521	0.0134	469	728	0.6442
1	12	2340	0.0051	842	1344	0.6265
2	13	4851	0.0025	137	595	0.2303
3	5476	12597	0.4347	52	483	0.1077

(b) Performance counters for the streaming implementation.

Table 5.10: Data cache performance counters for the grlib platform, using four contexts and 128 packets.

For writing, we see a slight decrease in miss rates for contexts 1, 2 and 3 (from 67.8% to 62.%, 34.3% to 23% and 27.5% to 10.8% respecitvely). Unfortunately, the benefit is lost because of an increase in write accesses in all these three contexts.

Even though we cut our miss rates down, the arbitration is harming access times of other simultaniously performed operations wether they are hits or misses. This can be

the reason why we see speed reductions for some implementations, while the best cases provide only a marginal speed-up.

5.6 Arbitration Losses

The results shown in the previous sections leads us to another question: why does the arbitration harm our accesses so much? The cause for this was investigated further deeply. To answer this question, we need to look back to Section 4.5.3.3. This approach, while correct, has two major problems in combination with the chosen arbitration method (a stream read always interrupts a normal access):

- When the stream source lane-group is not stalled and holds its stream signals high for more than one cycle, they are combinatorially routed to the cache blocks, who will keep the controller stalled for as long as the stream is being issued.
- When the stream source lane-group issues a stream request and this results in both a stream miss and regular miss (on its own cache), then he stalls immediately. The circuit will then keep the original request stable, until this lane-group un-stalls again. This means that it will keep issuing a stream request for this time, which wins arbitration and thus stalls the stream destination cache!

Unfortunately this was discovered at a late stage in this project, so it was no longer feasable to implement a fix for this and run all the experiments once more. We will however give two suggestions to solve this issue:

- Make the dual-port implementation synthesizeable. Again, this avoids most of the arbitration all-together, so it should be a good solution.
- Use a better circuit to ensure the validity of the stream requests. We can buffer the stream address and result immediately after a stream request is received, and ensure that stream requests are always ignored when their address matches the buffered stream address to avoid continuously issuing the same stream request.

5.7 Conclusion

In this chapter we experimentally verified the streaming cache design and attempted to measure its potential gains. We started off by describing the experimental setups used: a software simulation platform and several hardware platforms. We described the functional test-cases used to support development and debugging next for several aspects of both the write-back and streaming implementation, followed by larger realistic test-cases that served as our benchmarks. Next, we provide the results from the functional test-cases, ensuring they all pass. After this, we provide measurements of the benchmark applications (unfortunately only one functioned correctly), ran in the simulator. The following step was a synthesis, where we give resource utilization results and ran the benchmark applications again. We ended with an analysis of the cause of the disappointing results.

Devising a set of test-cases that find all bugs is a difficult, maybe even impossible process. Dijkstra famously said that "Program testing can be used to show the presence of bugs, but never to show their absence"[46]. The observation is clear here: even though our implementation passes our set of 13 test-cases, each designed to test different aspects, two out of three benchmark pipelines will not execute correctly. We also observed interesting promising metrics from the simulation model that were not as impressive on the actual hardware implementation. The one pipeline that did execute correctly showed reductions in the miss rates for the data cache blocks, but this did not translate into a meaningful reduction in the amount of cycles. This lead us to believe that the arbitration implementation is flawed or at least introduces unnecessary stalling, which further investigation in Section 5.6 seems to support.

The measurements from Section 5.5 become even less impressive when combined with the synthesis results from Section 5.4. As frequency is the amount of cycles executed per second, we need to divide the amount of cycles in total by this amount to get the real execution time. For the platform that observed a slight speed-up (grlib) we still lost in maximum synthesizeable frequency. Taking both into account leads to a total execution time of $\frac{12723}{3.75*10^7} = 3.3928 * 10^{-4}s$ on the baseline implementation versus $\frac{12278}{3.2432*10^7} = 3.7858 * 10^{-4}s$ when streaming, thus even on this platform we actually observe a performance loss.

6 Conclusion

In this section we will give the conclusion of this thesis. Firstly, the whole thesis is summarized. Next, the problem statement is discussed once again and we match it to the result of this work in order to answer the research question and assess if we obtained our goals or not. We will list the main contributions of this work. Lastly, we will discuss pointers for future work expanding on this thesis.

6.1 Summary

We started off by laying the foundation of this work in Chapter 2. We now have laid down the foundation on which we can build the rest of this thesis. We now know more about memory hierarchies, the cache and how both affect communication between different cores. An introduction to parallelism and how the ρ-VEX can exploit it in different cases has been given. We also know how to simulate hardware functionality as a validation model beforehand. Our target will be an Field Programmable Gate Array (FPGA) design as they allow for a very cost-effective prototyping work-flow. Lastly, we discussed software pipelines as our target application.

In Chapter 3, a design has been proposed to serve as an alternative to a cache hierarchy for multi-core systems, in order to alleviate penalties observed when doing extensive data sharing between cores. This streaming cache design aims to bypass the cache hierarchy in a simple but effective manner for reading operations.

Two options were considered to ensure a path between the stream source and stream target: a duplicated routing network or by restricting the source context. The latter was chosen as the former would impose a severe penalty on both the critical path and the area usage.

The next decision was to either re-use the existing memory request signals by adding a pre-network arbitration block, or to add additional signals to the routing network for streaming. The former solution would guarantee that each cache block can only get one memory request at all times, allows us to keep the cache blocks and the routing network the same but will affect the critical path, while the latter solution moves the arbitration problem to the cache blocks (unless we can use dual-port cache blocks) which might affect the critical path again, and modifies the routing network extensively. All three solutions will be implemented in the next section.

Stalling is required to be synchronized as well between all contexts that are involved in a stream operation that is blocked (due to arbitration, unless dual-port caches are used), but we also need to distribute any stalling signals due to the arbitration to all contexts that are coupled to the stream source.

The streaming operator is defined to ensure we always prioritize the source context, as we consider it a producer-consumer relationship.

External streaming (that is, streaming to contexts of different cores) is also considered, and the design needs to be done in a scalable fashion. To avoid flooding the memory with write requests and filling up the write-buffer in no time, we need to work around the write-through implementation of the current caches. This means we combine streaming with hybrid write-through and write-back caches, in order to keep things scalable.

For the hybrid caches, the decision is made to allocate a range of memory as write-back instead of marking individual data items as such, in order to stay compatible with the existing tool-chain and instruction set, as both do not need to be aware of the new features. Inside this region, the main cache controller needs to behave as a write-back cache controller instead of write-through. For that we also need to ensure that bus-snooping is disabled and add dirty bits to the cache lines.

To ensure the user can also disable the write-back feature, some form of consistency recovery is required. A controller is proposed with three methods, giving the programmer control over data priorities.

In Chapter 4 we discussed the implementation of both the hybrid write-through and write-back caches and the streaming features. First, the interface to the programmer was specified by defining the used control registers and their contents. For streaming, it was straightforward to insert a bit per context to allow streaming, while for write-back an address and size register was chosen (with several bits in the size register used to enable and to chose the consistency recovery mode).

After defining the interface, a simulation model was developed: the existing ρ-VEX simulator was extended to support these new features via the earlier discussed interface (and is thus forward compatible with the actual implementation). To implement this, the level of accuracy offered inside the simulator with regard to the cache lines was improved, and other parts of the cache code were re-structured as needed. Dirty bits were added, and the concept of eviction was added to allow the cache to function as write-back inside the write-back region. Streaming was then implemented straightforwardly by allowing the memory read function to look in the cache of a different context as well. The simulator did not support multi-core setups, and thus multi-core streaming was not implemented.

In the hardware, a controller for the newly added control registers was added that converts the contents of these registers to lane-group specific signals that are offered to the cache. It also arbitrated between several contexts, who in theory can request a new streaming configuration at the same time.

The data cache's valid memory block was copied to act as dirty bit instead. The memory controller was extended to behave in a write-back fashion inside the write-back region.

A stream request detection circuit was added, ensuring that it only occurred on context boundaries. Arbitration was firstly implemented in a pre-routing network setup to ensure that streams could be routed as normal read requests. This implementation however had a big critical path penalty, but also suffered from feed-back loops and potential when both the stream source and destination's stalling signals were influencing each other.

A second implementation was made where new signals were added to the routing network to accommodate streaming, allowing the network to route a read and stream read

separately. Assuming that dual-port caches are not available, an arbitration mechanism was added at the cache blocks too. Lastly, a mechanism to distribute the streaming hit signal over all lane groups coupled to the stream source is described.

The concept of streaming is expanded to multi-core implementations, which was relatively simple to do.

The last part that was added was the consistency recovery controller, which has the functionality to override the main cache controller in order to write all dirty data back to the main memory when needed.

Lastly, in Chapter 5 we experimentally verified the streaming cache design and attempted to measure its potential gains. We started off by describing the experimental setups used: a software simulation platform and several hardware platforms.

We described the functional test-cases used to support development and debugging next for several aspects of both the write-back and streaming implementation, followed by larger realistic test-cases that served as our benchmarks. A total of 13 test applications and 3 benchmarks were developed.

Next, we provide the results from the functional test-cases, ensuring they all pass. After this, we provide measurements of the benchmark applications (unfortunately only one functioned correctly), ran in the simulator.

The following step was a synthesis, where we give resource utilization results and ran the benchmark applications again. We ended with an analysis of the cause of the disappointing results.

6.2 Main Contributions

This section will revise the original research question, to see how much we can answer and if we met our original goals. Afterwards, we list all contributions made by this thesis, whether they are direct or indirectly related to the work. Initially, a lot of research was done in the field of graphics pipelines. This has led to several contributions that are not directly relevant to the final work. They are however included as they resulted in more knowledge about the nature of software pipelines in the wild.

6.2.1 Reflection

It is time to revise our original research question as posed in Section 1.1.1 and reflect on it. We asked the following question here:

How can streaming applications be efficiently executed on heterogeneous multi-core platforms, for example the ρ-VEX?

In this thesis we described a combination of two concepts in order to achieve this, and implemented a prototype to verify it. A simulation model showed a potential speed-up, but this was not yet supported by the prototype. Even though this prototype's results are disappointing, we believe this work has still led to several interesting insights. Not only did this project result in several graphics applications developed and/or ported to the ρ-VEX platform which can be useful as benchmarks, we believe the idea behind this project is feasible and can lead to speed-ups when the arbitration is done better.

Using a combination of hybrid write-back and streaming caches, we can stream data between cores without almost any code changes. This means the design and method require less porting effort than a set-up as traditional heterogeneous systems with different co-processors, where one usually have to handle the data transfers manually (which is often a bottleneck). We believe the arbitration method can be improved (as discussed in Section 5.6), which is expected to cut access times and thus execution times significantly.

Remember that the idea was proposed in the context of using a heterogeneous horizontal pipeline. Streaming caches can open the way to more speed-ups in a cost-effective manner by matching the computational structure of the target pipeline as perfectly as possible. So even if the speed-up does not completely negate the hit/miss rate loss of spreading work over different contexts, if combined with correct functional units over different lane groups we might gain a cost-effective solution.

The prototype described in Section 4.5 and Section 4.4 contains aspects that are quite ρ-VEX specific, which is not that surprising in an implementation. However, both the concepts as described in Chapter 3 and the simulation model as discussed in Section 4.2 are quite generic, and thus the conclusion that both can lead to a speed-up is not bound to the ρ-VEX platform. It is in fact not even bound to a heterogeneous platform, as inter-core communication can be sped-up on any multi-core platform with separate caches. Only the rationale of doing so for a horizontal software pipeline is bound to heterogeneous designs.

It is important to reflect upon one's work. We feel that beginning with a software simulation model is a good start, as this gave us a way to experiment with the idea early. This helped us shape our ideas quite well.

The difficulty of the actual implementation was however greatly underestimated. In the end, we opted for three designs, of which only two had synthesizeable implementations. Of those two synthesized implementations, only one could successfully run our test-cases, but was still unable to execute all pipelines correctly. A large amount of time was lost in debugging all three implementations.

Not only debugging, but also testing effort was initially underestimated. Initially, only a small amount of test-cases were designed, which assisted in (early) development. Once the implementations passed the initial test set, it was observed that the benchmarks did not run correctly. By investigating further, we found more potential issues and expanded our test-set greatly. Most often, a new test-case would be written that mimicked one aspect of the pipeline, and we would observe a failure on this test-case. Its execution was then thoroughly simulated and investigated, to find the cause and fix it.

An important question that we would like to ask ourselves, is what would have been done differently should we try again. Most likely, we would opt for a new cache design with updated requirements, so we can avoid certain decisions that made streaming caches tricky to implement. Also, we would separate the write-back and streaming features. We believe it might have been the additional complexity of having more kinds of write policies that made the streaming functionality hard to debug and correctly implement. The hybrid write-back policy was suggested for two reasons: we initially assumed (incorrectly) that it would be required for streaming to work, but we also thought that the write-buffer would fill up too quickly if streaming were used extensively. The former assumption later

proved to be incorrect, while a different solution to the latter can be devised. For some streaming scenarios, a deeper write-buffer might prove to be a solution that is cheaper and easier to implement.

6.2.2 Contributions

This research has led to the following contributions, mostly to the ρ-VEX project:

- A general framebuffer driver model was developed for the ρ-VEX platforms which could be used to speed-up development of other projects and is used by all other graphical applications in this project. It supports the grlib and grlib-doom hardware platforms, but also sim-rvex. Different implementations were written for one, two and three byte-per-pixel formats. We even added a version that replaced the backend target from the ρ-VEX framebuffer to an SDL2-backend, to enable cross-platform development on a general purpose platform.

- Ported the first person shooter game DOOM, developed by ID-Software, to the ρ-VEX architecture in order to gain more knowledge about the the ρ-VEX core and tool-chain, the grlib platform, the frame-buffer and SVGA controller, but also to see if it was a suitable target for acceleration.

- Ported TinyGL, a software library that implements a subset of the OpenGL 1.1 fixed function graphics pipeline to gain more knowledge about graphics pipelines, software pipelines in general and to determine if it was a suitable application to optimize.

- Implemented a software graphics pipeline, Runderer, with clearly separated programmable pipeline stages (inspired by OpenGL 2 and up) to serve as a new acceleration target when it was clear that the previous targets were both unsuited.

- Implemented an application that can apply convolution filters over the framebuffer, as a potential speed-up benchmark target.

- Increased the accuracy of sim-rvex's cache simulation.

- Developed a hybrid write-back write-through scheme that can be used to speed-up pure write-through caches in the case where consistency is not required, like the core/context local stack.

- This hybrid write-back write-through scheme was also implemented in sim-rvex.

- Developed a streaming cache scheme that allows multi-context and core processes to share data in a pipeline-suitable fashion, reducing data communication penalties, making vertical software pipeline execution more attractive (which could be interesting in a heterogeneous setup).

6.3 Future Work

In this section we propose several topics that might be suitable for future research. Several of these topics were originally intended to be looked into during this work, but were unfortunately dropped due to time constraints, since debugging the implementation took more time than initially expected.

- At the time of writing, the streaming cache implementation still suffers from some bugs, where an the streaming arbitration can affect other read/writes on the same cache block. These bugs still need to be isolated and fixed. Once this is completed, the other two target benchmarks can be run.
- Arbitration is currently prioritizing streaming too aggressively. Instead of allowing the stream every cycle it is requested (and thus blocking the lane group who owns this cache block), the result should be buffered so that we can let arbitration only win if requested and the requested address is not buffered yet. This should greatly improve the performance and hopefully make streaming actually provide a speed-up.
- We were not able to make the synthesis tool correctly infer dual-port versions of the cache blocks. We did not invest more time in this as this is very tool-specific and we aimed for a generic prototype in this work. There is no fundamental issue here though, as dual-port memory elements are common, and the Block RAM (BRAM)s on the targeted FPGA device family are in also fact dual-port. Given more time, this inference issue could be solved, eliminating the need for most of the arbitration (only the invalidate logic has to be arbitrated with the stream request), potentially improving performance significantly.
- Another instance of the valid bit memory is used to store the dirty bit, as the consistency recovery mechanism required the ability to single-cycle clear all dirty bits. If this mechanism is not required, normal BRAMs could be used instead to store the dirty bit, which might be cheaper in terms of resources.
- The original cache design was of-course not designed with streaming or write-back regions in mind. A re-design of the entire cache with these new features in mind might lead to a more elegant implementation, with a possible performance improvement. For example, the stream hit broadcasting system that is required to synchronize stream stalling might be avoided earlier on in the design phase.
- Streaming was designed to function from the write-back region. At the time that this design decision was made, we made the incorrect assumption that streaming would never work from write-through memory. When this was discovered, the interface was already defined and implemented in the simulator, so we decided to keep it this way for the sake of compatibility. There is no fundamental issue here however, except for the fact that the write-buffer will be filled quite fast in streaming pipelines. If this is not an issue, streaming and write-back can be separated even more by defining separate stream- and write-back regions, that may or may not overlap.

- The write-back region is now defined by its address and size, and manually set by the user. While this is straightforward and relatively simple to set, this can still be made more user-friendly. Generally, a programmer is only concerned with using static memory (most often on the stack) and allocating dynamic memory (usually on the heap), via a standard library function such as *malloc*. A run-time management system could be designed that manages not only the standard heap, but also a separate write-back heap that grows dynamically as needed, and offering a function such as *wbmalloc*.

- The idea of write-back memory can be used to accelerate memory operations on "private" memory, that never needs to be shared across cores, for example the stack of each context. This should be used with caution though, as we are effectively disabling coherency on this memory region. More experiments can be run with this, to see if this is viable.

- The original idea behind mapping a pipeline in a horizontal way was to be able to fine-tune the cores/contexts inside the processor to the profile of each separate pipeline stage. One can add a multiplier only to certain lane groups which correspond to pipeline stages that require one, for example. This idea is still promising and could be looked into.

- Some Runderer-specific variations on this would be to accelerate the rasterization step in hardware, as this is currently making the pipeline very unbalanced, but also to add floating point units to only the early stages in the pipeline, as the fragments outputted by the rasterizer can be expressed in integers. The vertex shader (which requires a floating point unit) contexts can be stripped down severely because there are generally much less vertices as compared to fragments (compare the three points that define a triangle to all the pixels inside the triangle).

Bibliography

[1] M. Zwolinski, *Digital System Design with VHDL*. Pearson Education, 2004.

[2] S. M. Chai, S. Chiricescu, R. Essick, B. Lucas, P. May, K. Moat, J. M. Norris, M. Schuette, and A. Lopez-Lagunas, "Streaming Processors for Next-Generation Mobile Imaging Applications," *IEEE Communications Magazine*, vol. 43, no. 12, pp. 81–89, 2005.

[3] J. Andrews and N. Baker, "Xbox 360 System Architecture," *IEEE micro*, vol. 26, no. 2, pp. 25–37, 2006.

[4] J. L. H. D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, fourth ed., 2009.

[5] J. Montrym and H. Moreton, "The GeForce 6800," *IEEE Micro*, vol. 25, no. 2, pp. 41–51, 2005.

[6] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE micro*, vol. 28, no. 2, pp. 39–55, 2008.

[7] J. v. Straten, "A Dynamically Reconfigurable VLIW Processor and Cache Design with Precise Trap and Debug Support," Master's thesis, Delft University of Technology, The Netherlands, 2016.

[8] P. K. Chatterjee, W. R. Hunter, A. Amerasekera, S. Aur, C. Duvvury, P. E. Nicollian, L. M. Ting, and P. Yang, "Trends for Deep Submicron VLSI and Their Implications for Reliability," in *Reliability Physics Symposium, 1995. 33rd Annual Proceedings., IEEE International*, pp. 1–11, IEEE, 1995.

[9] J. W. McPherson, "Reliability Challenges for 45nm and Beyond," in *Proceedings of the 43rd annual Design Automation Conference*, pp. 176–181, ACM, 2006.

[10] G. Gielen, P. De Wit, E. Maricau, J. Loeckx, J. Martin-Martinez, B. Kaczer, G. Groeseneken, R. Rodríguez, and M. Nafria, "Emerging Yield and Reliability Challenges in Nanometer CMOS Technologies," in *Proceedings of the conference on Design, automation and test in Europe*, pp. 1322–1327, ACM, 2008.

[11] A. Grama, A. Gupta, G. Karypis, and V. Kumar, "Introduction to Parallel Computing," 2003.

[12] J. L. Hennessey and D. A. Patterson, *Computer Architecture: A Quantitive Approach*. Morgan Kaufmann, fourth ed., 2011.

[13] B. R. Rau and J. A. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective," *The journal of Supercomputing*, vol. 7, no. 1-2, pp. 9–50, 1993.

[14] J. Staunstrup and W. Wolf, *Hardware/Software Co-Design: Principles and Practice*. Kluwer Academic Publishers, 1997.

[15] T. v. As, "ρ-VEX: A Reconfigurable and Extensible VLIW Processor," Master's thesis, Delft University of Technology, The Netherlands, 2008.

[16] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, "Portable Performance on Heterogeneous Architectures," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 431–444, ACM, 2013.

[17] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[18] S. Prybylski, M. Horowitz, and J. Hennessy, "Performance Tradeoffs in Cache Design," in *ACM SIGARCH Computer Architecture News*, vol. 16, pp. 290–298, IEEE Computer Society Press, 1988.

[19] A. J. Smith, "Cache Memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[20] A. W. Wilson Jr, "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," in *Proceedings of the 14th annual international symposium on Computer architecture*, pp. 244–252, ACM, 1987.

[21] Intel Corporation, *Intel© and IA-32 Architectures Optimization Reference Manual Manual*, June 2016.

[22] P. Stenstrom, "A Survey of Cache Coherence Schemes for Multiprocessors," *Computer*, vol. 23, no. 6, pp. 12–24, 1990.

[23] J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems (TOCS)*, vol. 4, no. 4, pp. 273–298, 1986.

[24] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory traffic," *ACM SIGARCH Computer Architecture News*, vol. 11, no. 3, pp. 124–131, 1983.

[25] Intel Corporation, *Intel© and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide*, September 2016.

[26] ARM Corporation, *ARM© Cortex©-R7 MPCore Technical Reference Manual*, 2014.

[27] ARM Corporation, *Cortex^{TM}-R5 Technical Reference Manual*, 2011.

[28] ARM Corporation, *ARM© Cortex-A Series Programmer's Guide for ARMv8-A*, 2015.

[29] ARM Corporation, *Cortex-A7 MPCore Technical Reference Manual*, 2012.

[30] "HP Labs : Downloads: VEX." (Visited Oct. 10, 2016).

[31] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[32] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[33] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing," in *ACM SIGARCH Computer Architecture News*, vol. 28, pp. 203–213, ACM, 2000.

[34] J. J. Rodriguez-Andina, M. J. Moure, and M. D. Valdes, "Features, Design Tools, and Application Domains of FPGAs," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 4, pp. 1810–1823, 2007.

[35] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, 1992.

[36] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli, "Architecture of field-programmable gate arrays," *Proceedings of the IEEE*, vol. 81, no. 7, pp. 1013–1029, 1993.

[37] D. A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM*, vol. 28, no. 1, pp. 8–21, 1985.

[38] M. Segal and K. Akeley, "The OpenGLR Graphics System: A Specification Version 1.2.1," 1992.

[39] G. De Micheli and L. Benini, *Networks on Chips: Technology and Tools*. Morgan Kaufmann, 2006.

[40] A. Brandon and S. Wong, "Support For Dynamic Issue Width in VLIW Processors Using Generic Binaries," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 827–832, EDA Consortium, 2013.

[41] Xilinx Inc, *Xilinx Virtex-6 Family Overview*, August 2015.

[42] Cobham Gaisler, *GRLIB IP Library User's Manual*, January 2016.

[43] C. Kanan and G. W. Cottrell, "Color-to-Grayscale: Does the Method Matter in Image Recognition?," *PloS one*, vol. 7, no. 1, p. e29740, 2012.

[44] K. Jack, *Video Demystified: a Handbook for the Digital Engineer*. Elsevier, 2011.

[45] "Parameter Values for the HDTV Standards for Production and International Programme Exchange."

[46] E. W. Dijkstra, "Notes on Structured Programming," 1970.

[47] N. Gharachorloo, S. Gupta, R. F. Sproull, and I. E. Sutherland, "A Characterization of Ten Rasterization Techniques," *ACM SIGGRAPH Computer Graphics*, vol. 23, no. 3, pp. 355–368, 1989.

[48] V. Skala, "Barycentric Coordinates Computation in Homogeneous Coordinates," *Computers & Graphics*, vol. 32, no. 1, pp. 120–127, 2008.

Runderer

A

The project of developing a software rendering pipeline proved to be more straightforward than expected. The project, which we named *runderer*[1] is open-source and has been actively developed on by 2 authors. The codebase is left completely generic and portable, any platform specific code has been left out (and moved to a private repository), to keep it accessable and easy to work with. The only assumption made on the hardware is that it has a 16-bits per pixel framebuffer (in 5R6G5B color format). The interface with this framebuffer was abstracted away from in a minimal framebuffer device driver with a reference implementation using *libSDL2*[2] as a backend.

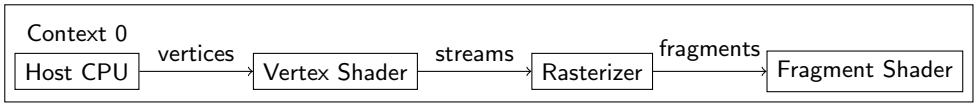
Runderer's pipeline structure is based on the classic OpenGL rendering pipeline (taking the vertex shader, rasterizer and fragment shader stages), with a simple default implementation for each stage to act as the fixed graphics pipeline from OpenGL 1[38]. The basic structure is presented in Figure A.1. The pipeline stages are implemented as function pointers that are placed inside the main runderer structure. These stages are therefore easily replaced with a custom rendering method. Several matrices required to simulate the fixed pipeline are also included in this structure. A very limited number of OpenGL API functions are offered as well to make some simple demo's possible.

Runderer currently supports rendering only triangles. Quad support is also added by splitting each quad into two triangles. As the ρ-VEX has up to four reconfigurable hardware contexts, we can map this pipeline in several different ways. We offer two methods of interfacing between the pipeline stages for different use-cases:

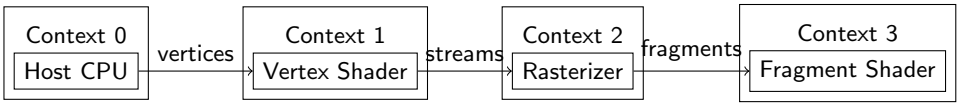
- Default, generic single context version. The draw method simply calls each pipeline stage in order, directly passing along the vertices, streams and fragments locally stored from stage to stage. An example mapping and dataflow is shown in Figure A.1a.

- Pipelined, multi-context ρ-VEX version. This version uses global static FIFO buffers to communicate between the stages. This enables us to run pipeline stages on different contexts. These buffers don't contain raw vertices, streams and fragments, but rather a command structure to ensure that we can perform several higher-level operations in the correct order as well. An example mapping and dataflow is shown in Figure A.1b.

[1] Runderer: https://github.com/fumyuun/runderer
[2] libSDL2: https://www.libsdl.org/



(a) An example configuration of the runderer rendering pipeline in single context mode.



(b) An example configuration of the runderer rendering pipeline in multiple context mode.

Figure A.1: Two different pipeline stage configuration interfaces provided by runderer.

function DRAW_TRIANGLE_ARRAY($vertices$, $count$)
  while $i < count$ **do**
    $stream_1 \leftarrow$ VERTEX_SHADER($vertices_i$)
    $stream_2 \leftarrow$ VERTEX_SHADER($vertices_{i+1}$)
    $stream_3 \leftarrow$ VERTEX_SHADER($vertices_{i+2}$)
    $fragments \leftarrow$ RASTERIZER($stream_1, stream_2, stream_3$)
    for all $fragment \leftarrow fragments$ **do**
      $framebuffer \leftarrow$ FRAGMENT_SHADER($fragment$)
    end for
    $i \leftarrow i + 1$
  end while
end function

Figure A.2: Algorithm for the single core pipeline.

A.1 Pipeline Stages

As shown before, the rendering pipeline is split into several stages, each with characteristic data inputs, outputs and function. In this section we will briefly discuss the function of each stage. After that, we will describe the type of data that is moved between stages and the typical volume. Finally, we will shortly discuss the difference between the single- and multi core setups.

We have the following stages in the pipeline:

- Host CPU: not really a part of the pipeline per se, the host CPU generates colored points in *world coordinates*, called *vertices*. World coordinates are described using homogeneous coordinates, thus they are four-dimensional vectors (with the last component set to 1). The colors are also described using a four-dimensional vector (red, green, blue and alpha components).

- Vertex Shader: the vertex shader operates on each vertex in parallel, converting them from world coordinates into *screen coordinates*, called *streams*, by a series of vector space transformations. Screen coordinates are no longer in a homogeneous coordinate system, thus the fourth dimension of the position is dropped. This stage can also be used to do clipping to the screen or clipping areas, but this is

```
function DRAW_TRIANGLE_ARRAY(vertices, count)
    while i < count do
        vertex_buffer ← vertices_i
        i ← i + 1
    end while
end function
function VERTEX_MAIN
    loop
        if |vertex_buffer| > 0 then
            vertex ← vertex_buffer
            stream_buffer ← VERTEX_SHADER(vertex)
        end if
    end loop
end function
function RASTERIZER_MAIN
    loop
        if |stream_buffer| >= 3 then
            stream_1 ← stream_buffer
            stream_2 ← stream_buffer
            stream_3 ← stream_buffer
            fragment_buffer ← RASTERIZER(stream_1, stream_2, stream_3)
        end if
    end loop
end function
function FRAGMENT_MAIN
    loop
        if |stream_buffer| > 0 then
            fragment ← fragment_buffer
            framebuffer ← FRAGMENT_SHADER(fragment)
        end if
    end loop
end function
```

Figure A.3: Algorithms for the multi core pipeline.

not supported at the time of writing. We can currently consider it as a stage that consumes one vertex and produces exactly one stream, but when clipping is supported it can theoretically generate 0 or more streams (depending on how many points are outside the clipping area).

- Rasterizer: the rasterizer is responsible for filling the area in between these points that make up a triangle. For each pixel inside the triangle, a depth-value and color has to be interpolated from the three defining points. It generates *fragments*, which are points in *device coordinates*, called *fragments*, still augmented with color information. Since we only support triangles for now, the rasterizer always con-

Vertex								Stream/Fragment						
position				color				position			color			
x	y	z	w	r	g	b	a	x	y	z	r	g	b	a

Table A.1: The datatypes communicated between the stages.

sumes three streams and most likely generates a lot of fragments (as each fragment corresponds to a single framebuffer entry). The chosen rasterization algorithm is based on barycentric coordinates, which are explained in Section A.3.

- Fragment Shader: the fragment shader operates on each fragment in parallel, doing the typical final color calculations related to lightning and material properties (both not supported at the time of writing), and performs the framebuffer writes.

A.2 Commands

These data-structures are sufficient when running the single-context mode, as there is not really a pipeline. When expanding to the multi core implementation, one problem arises: the host CPU is now detached from the rendering process and thus has no guarantees over when they are finished. As a direct consequence of this, he can no longer directly clear the screen (to prepare for drawing a new frame) and he does not know when the frame is done (to swap the front- and back buffers).

To solve this problem, a command structure is introduced for each elemental data structure. Each data structure is augmented with a command field: *draw*, *clear* or *swap*. The draw commands are executed as normal. Each stage simply passes both clear and swap commands directly to the next frame, preserving the order. The final context will interpet the commands, clearing the screen, swapping the buffers or executing the normal fragment shader. This solution was straightforward to implement, but will no longer work when there is more than one fragment shader core.

A.3 Rasterization

Rasterization is a computationally expensive procedure, and we discovered that this process was always a big bottleneck in the pipeline. The basic problem of rasterization is: how to draw a triangle (defined by three vertices)? When the triangle is not solid, this is quite easy to solve (as we only need to draw a line between each vertex). However, when the triangle is solid, this is much more computationally expensive. Most rasterization algorithms express a test of how to determine if a given pixel falls within the triangle or not, of which the calculation should be entirely independent of any other pixel. If this is the case, rasterization is intensive but also highly parallelizeable, as we can for each pixel in parallel test if they are in the triangle or not.

A sub-problem of rasterization is the depth- and color value interpolation: given three vertices we not only need to know the x and y coordinates of each point inside the

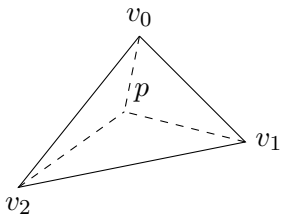
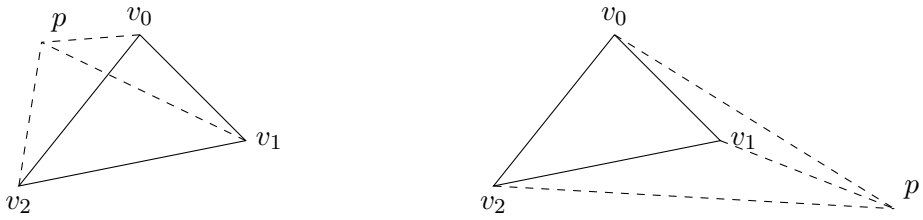


Figure A.4: An example barycentric system where p is expressed by its distances from three vertices, normalized.



(a) An example barycentric system where the distance between p and $v1$ is larger than 1.

(b) An example barycentric system where the distance of p to $v1$ is less than 0.

triangle, but also the corresponding depth value of this pixel (and its color, which is also often interpolated from the colors of the three vertices).

A lot of rasterization techniques require special kinds of framebuffer memories[47], but as said before this implementation is generic. The rasterizer that was implemented is quite complex and based on barycentric coordinates in a homogeneous coordinate system[48]. Barycentric coordinates express a point (or vector) in a two-dimensional space with respect to a given triangle, as a system of linear equations where each dimension of the vector describes the (normalized) distance from one of the vertices of the triangle. An important property is that the sum of all values inside the coordinate is 1.

Once a point is converted to barycentric coordinates, it is trivial to determine whether or not the point is inside the triangle: the new vertex describes a normalized distance from each point, while the sum is always 1. This means that if any of the coordinates is 1, the point lies on the border of the opposite line. More importantly, if one of the coordinates is more than 1, the point is on the other side of that line, and thus outside of the triangle (an example is shown in Figure A.5a). Because all points sum to 1, this means that at least one of the other points has to be negative. Another example is shown in Figure A.5b, where the distance between p and v_0 is negative.

This means that once we have converted a point to barycentric coordinates, the test is simply checking if any of the components is negative. Also, depth interpolation is now trivial, since it is simply the weighted sum of all of the depth values of each of the triangle's vertices, with the distance of the corresponding point as weight. Color interpolation is done in the same way.