Implementation of a Hardware Ray Tracer for digital design education

Jonas Agentoft Eggen

Master of Science in Electronics
Submission date: June 2017
Supervisor: Øystein Gjermundnes, IES

Norwegian University of Science and Technology
Department of Electronic Systems

Assignment text

Implementation of a Hardware Ray Tracer for digital design education

Supervisor : Øystein Gjermundnes (NTNU/ARM)

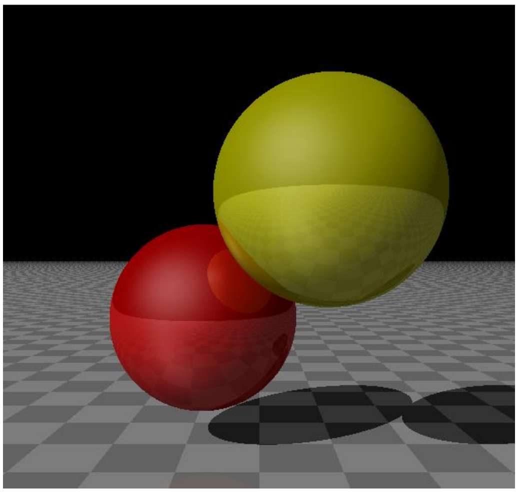


Figure 1. Scene rendered with a model of the ray tracer.

Platform for advanced digital design education

NTNU and ARM are developing a platform for teaching advanced digital design built around a Hardware Ray Tracer. The goal is to carefully documenting all steps in the design process of the Ray Tracer, starting from the *requirement capture*, *architecture exploration and modelling*, *specification writing*, *RTL implementation* and finally *creating a working prototype*.

Ray Tracing in Hardware is an excellent case for exploring and documenting topics such as:

- Best practice design and verification processes of a system of relatively high complexity.
- Modelling on different abstraction levels with the aim to understand system performance, system scalability, memory bandwidth analysis and quantization effects in mathematical computations.
- Sub-System design with common sub blocks such as memory system, processor elements and interconnects.

- Examples of how to efficiently implement a large range of mathematical operations in hardware.
- Understanding how the Hardware Ray Tracer can be integrated into a larger SoC together with CPU, display controller and memory controller.

Project thesis “Design of a Hardware Ray Tracer for digital design education”

In the project thesis “Design of a Hardware Ray Tracer for digital design education” submitted December 2016 by NTNU student Jonas Agentoft Eggen modelling and design exploration of a hardware ray tracer was described. Through extensive design exploration with constant refinement of the model Eggen arrived at a proposal for the architecture, instruction set and microarchitecture for a scalable multicore ray tracer. As a part of this work, Eggen carried out analysis of numeric precision, throughput, and scalability of the multicore system.

Master thesis description

In this master thesis, the goal is to refine the microarchitecture for the ray tracer core proposed in the project thesis, propose an interconnect that scales well with the number of cores, write RTL code, verify the design and synthesize the design targeting a Xilinx Zynq-7000 FPGA. Extract examples from the thesis work that helps demonstrating important aspects of digital design.

Abstract

Digital design is a large and complex field of electronic engineering, and learning digital design requires maturing over time. The learning process can be facilitated by making use of a single learning platform throughout a whole course.

A learning platform built around a hardware ray tracer can be used in illustrating many important aspects of digital design. A unified learning platform allows students to delve into intricate details of digital design while still seeing the bigger picture. Effects of changing parameters at a low level in the ray tracer design can be seen at the top-level straight away. This kind of fast feedback can help keep students motivated through the learning process.

Throughout this thesis, many interesting examples of both assignments and student discussions are presented. These cover topics such as technology dependent optimisations, low power design techniques, verification and means of accelerating the design process. The combination of these examples and the implementation effort in this thesis is a good starting point for a learning platform.

Ray tracing is a parallel problem well suited for processing in a multi-core architecture. Here, a system that can be synthesised with a parameterisable number of processing cores is proposed. Each of the cores interleave processing of rays using fine-grained multithreading. Large parts of the system have been implemented using the SystemVerilog hardware description language. Tools used in exploring the impact of architectural changes have been developed and results from these are discussed. The implementation is verified through simulations and partly using formal methods. Synthesis results for a Xilinx Zynq SoC are presented and discussed.

Simulation and synthesis results indicate that the ray tracer can render a VGA frame at 25 frames per second in a 32-core configuration. This configuration utilises $\sim 77\,\%$ of the LUTs on the target FPGA, leaving room for additional logic on the device.

Sammendrag

Digitaldesign er et stort og komplekst felt innen elektronikk. Det å lære digitaldesign krever modning over tid. Læringsprosessen kan lettes ved å benytte én enkelt læringsplattform gjennom et helt emne.

En læringsplattform bygget rundt en raytracer kan brukes til å illustrere mange viktige aspekter ved digitaldesign. En enhetlig læringsplattform gjør det mulig for studenter å fordype seg i intrikate detaljer innen digitaldesign samtidig som de kan ha et overblikk over hele systemet. Resultater av parameterendringer på et lavt nivå i raytracerens design kan ses på toppnivå med en gang. Slike raske tilbakemeldinger kan hjelpe studenter med å holde motivasjonen oppe gjennom læringsprosessen.

Gjennom hele oppgaven presenteres mange interessante eksempler på både oppgaver og studentdiskusjoner. Disse dekker emner som teknologiavhengige optimaliseringer, energisparingsteknikker, verifisering og metoder for å effektivisere designprosessen. Kombinasjonen av disse eksemplene og arbeidet med implementasjonen i denne oppgaven gir et godt utgangspunkt for en læringsplattform.

Raytracing er et parallelliserbart problem godt egnet for prosessering med en flerkjernearkitektur. Her foreslås et system som kan syntetiseres med et parameteriserbart antall kjerner. Hver kjerne prosesserer stråler (rays) med finkornet (fine-grained) multithreading. Store deler av systemet er blitt implementert i SystemVerilog. Verktøy for å undersøke virkninger av arkitektoniske endringer er utviklet og resultater fra disse diskuteres. Implementasjonen er verifisert gjennom simuleringer og delvis ved hjelp av formelle metoder. Synteseresultater for en Xilinx Zynq SoC presenteres og diskuteres.

Resultater fra simuleringer og syntese indikerer at raytraceren kan generere 25 bilder i sekundet gitt en 32-kjerners konfigurasjon. Denne konfigurasjonen bruker $\sim 77\,\%$ av LUTene på FPGAen. Dette betyr at det er plass til øvrig logikk på enheten.

Preface

This master's thesis has been written during the spring semester of 2017 at the Department of Electronic Systems, Norwegian University of Science and Technology (NTNU). The work has been done in collaboration with ARM. The thesis accounts for the full workload of the final semester at NTNU.

The interesting nature of digital design and computer graphics has been a major motivation in my work with this thesis. These are both areas that have experienced huge growth the last few decades, and will continue to play an important role in the world for many years to come. Furthermore, I have found working with the technical intricacies encountered very rewarding. I hope the teaching platform will trigger some of the same feelings in students once it is finished.

I would like to thank my supervisor Øystein Gjermundnes for his support and continued enthusiasm regarding technical problems of any size as well as results I have presented throughout this last year.

Thanks to Silja for being so nice.

Contents

List of Figures

List of Tables

List of Learning examples

Acronyms

ALU arithmetic logic unit.

APB Advanced Peripheral Bus.

ASIC application-specific integrated circuit.

AXI Advanced eXtensible Interface.

BRAM block RAM.

CPU central processing unit.

DSP digital signal processing.

EDA electronic design automation.

FF flip-flop.

FPGA field-programmable gate array.

FPS frames per second.

FPU floating-point unit.

FSM finite-state machine.

HDL hardware description language.

HDMI high-definition multimedia interface.

IC integrated circuit.

IP intellectual property.

IRQ interrupt request.

ISA instruction set architecture.

LSB least significant bit.

LUT look-up table.

LUTRAM LUT/distributed RAM.

MAC multiply-accumulate.

MADD multiply-add.

NaN not a number.

PC program counter.

PPA power, performance and area.

RAW read after write.

RGB red, green, blue.

RTL register-transfer level.

SoC System on Chip.

STA static timing analysis.

TLM transaction-level modeling.

VFPU vector floating-point unit.

VHDL VHSIC hardware description language.

VHSIC Very High Speed Integrated Circuit.

Chapter 1

Introduction & Motivation

Digital design is a field within electronics that has experienced great growth since the first integrated circuits (ICs) were introduced in the 1960s. Since then, ICs have become ubiquitous, and found in virtually all electronic devices. Competent engineers are vital to keep up the progress in the field. Educating these engineers can be facilitated by making use of one single learning platform throughout a whole course on digital design.

Computer graphics is another field that has come to take a huge part of people's lives. Since the term was first coined in 1960 [The+08], it has found applications in computer aided design, medical imaging, scientific visualisation, video games, special effects in movies and more. Now, with virtual reality on the rise, the demand for more powerful hardware is not showing any sign of decline.

Because of this, designing a learning platform for digital design around a hardware ray tracer is a natural choice. The system complexity is kept at a moderate level, while still demonstrating key concepts of digital design. The learning platform guides students through the phases of digital design, resulting in a system that offers a visual output. A visual output can help motivate students to really understand how the system works, and thus learn digital design. Students that in a few years will be important in defining the technology of tomorrow.

The benefit of a platform of this complexity is that it can be used as an example throughout a whole course. It can be used to demonstrate everything from ‘requirement capture, architecture exploration and modelling, specification writing, RTL implementation [to] finally creating a working prototype.' ([Gje17])

This thesis is a continuation of the project thesis 'Design of a Hardware Ray Tracer for digital design education' [Egg16]. In that thesis, a high level functional model was provided [Gje17]. This model was analysed and broken down into more specialised models. The result of this was a system that was partitioned in a way that had a parameterisable number of processing cores. This allowed for scalable performance, and it was found that achieving a frame rate of 24 frames per second (FPS) at VGA resolution required 74 cores when running at 50 MHz. The project thesis concluded by recommending improvements to the design in order to make implementation feasible.

In this master's thesis, improvements to the architecture proposal from [Egg16]. Models developed during the project thesis have been adapted, and used in both performance analyses and in verifying correctness of the register-transfer level (RTL) implementation. RTL for a large part of the system, including the processing elements and their interconnect, has been implemented using SystemVerilog. The RTL has been synthesised for a field-programmable gate array (FPGA) target. The impact of varying important design parameters has been analysed through simulations and synthesis.

1.1 Main contributions

The main contributions of this thesis are:

- The specification from [Egg16] has been improved and extended.
- Devised a multi-core architecture where cores are fully utilised due to the use of fine-grained multithreading.
- Implemented RTL for large parts of the system and made an assembler for the instruction set architecture (ISA).
- Simulated and synthesised the RTL.
- Verification of the implementation through simulation.
- Analyses of expected system level performance.
- Learning examples and proposal for use of presented material in a course on digital design.

1.2 Thesis outline

Some necessary background theory is given in Chapter 2. It gives an introduction to ray tracing, number representation formats, the digital design process, the targeted development board and interconnect design. Chapter 3 lists the requirements for both the hardware ray tracer and its specification documents. Some of the requirements were addressed during the work on the project thesis, and some are entirely new. Important results from the project thesis are summarised in Chapter 4. Chapter 5 describes the design process of the ray tracer system. Results of design choices are also found in this chapter. This presentation method is chosen as it is in accordance with the design process of a digital system. Anecdotal learning examples are scattered around in this chapter, providing the author's view on various themes. Chapter 6 discusses the viability of this system as a learning platform. An example of how one of the modules in the system can be integrated into a digital design course is also given. Finally, Chapter 7 sums everything up, and lists work that remains before the hardware ray tracer can be used in digital design education.

Chapter 2

Background

This chapter presents some background theory that is necessary for the understanding of Chapters 4 and 5. The ray tracing algorithm is introduced. Basics of the fixed-point and floating-point number representations are presented. A short introduction to a typical design flow is given. The ZedBoard development board and interconnects in digital systems are introduced. Sections 2.1 and 2.2 has been taken directly from [Egg16].

2.1 Ray tracing

Ray tracing is a rendering algorithm that can produce realistic looking images. It achieves this by tracing light through the image plane, and simulates their interaction with the objects in the scene. A scene refers to a description of objects to be rendered, lighting, camera viewpoint amongst other parameters. The ray tracing algorithm can simulate many optical effects like reflection, refraction, scattering and dispersion.

Figure 2.1 illustrates the basic concepts of the ray tracing algorithm. A primary ray is cast from the camera position through a pixel in the image. Intersection tests against the objects in the scene are performed in order to find the closest intersecting object. A shadow ray is traced towards the light source, to determine if the object is in shadow. A local illumination model is then applied for the hit object. In recursive ray tracing, reflection and refraction rays can then be generated and are traced in the same way. The contributions from each term is then summed and the final pixel colour is returned.

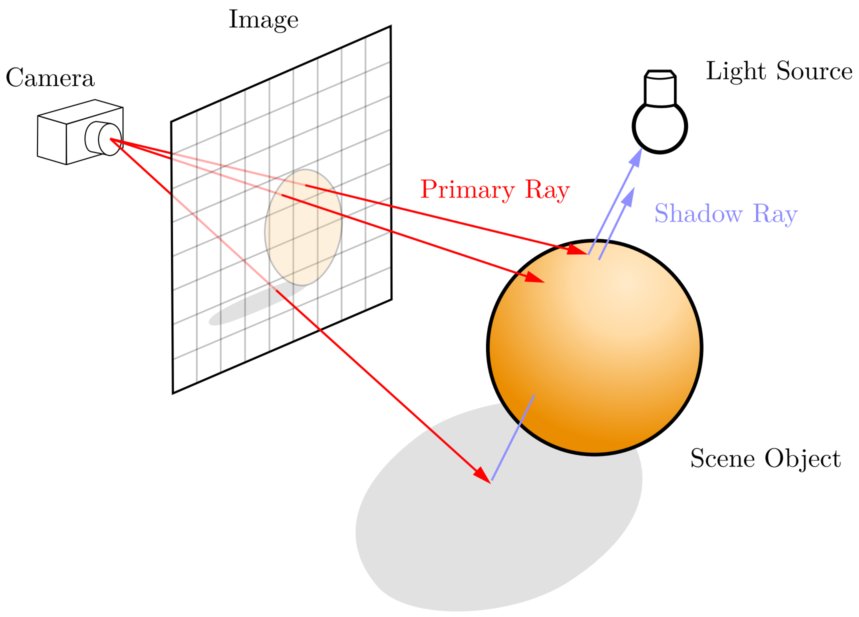


Figure 2.1: Ray tracing visualised [Hen08]

The recursive ray tracing algorithm has been around since 1980 [Whi80]. Since then, a lot of effort has been put into making ray tracing more efficient and feature-rich [Suf07]. The work has mainly focused on improving the runtime of the algorithm in scenes with a large number of objects. A naive implementation of the algorithm will try to intersect any ray with all objects in the scene. This is very inefficient for large scenes, and has been solved by subdividing the scene objects into data structures of objects that are cheap to perform intersection tests on. These data structures allow for a major reduction in the number of needed intersection tests.

2.2 Number representations

There exist many ways to represent numbers in digital systems. The choice of representation has a big impact on the design, performance etc. of the system. Here, only fixed-point and floating-point numbers will be considered due to the popularity of those representations. Also, no literature recommending any other number formats for use in ray tracing have been found. For more in-depth coverage of fixed-point and floating-point numbers, refer to [EL04].

2.2.1 Fixed-point representation

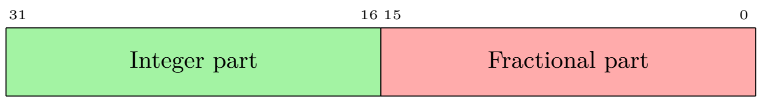


Figure 2.2: Fixed-point format with 16 bit reserved for the integer part, and 16 bit for the fractional part.

Fixed-point numbers represent real numbers using a fixed number of bits before and after the binary point. Figure 2.2 shows a fixed-point format with 16 bit reserved for the integer part, and 16 bit for the fractional part. For signed numbers, one can choose signed number representation freely (i.e. sign-magnitude, two's complement, etc.).

2.2.2 Floating-point representation

Floating-point numbers also represent real numbers. As opposed to fixed-point numbers, the binary point can *float*. This allows for a great dynamic range at

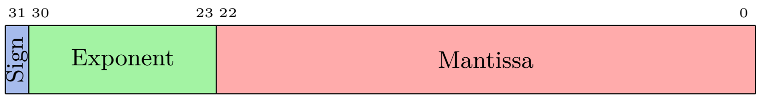


Figure 2.3: IEEE Standard 754 Single-precision floating-point format

the cost of less precision, rounding errors and relatively complex implementation [EL04]. In defining a floating-point system, there are many parameters that must be determined. The IEEE Floating-point Standard 754 [IEEE08] defines the format, rounding modes, special values, operations, gradual underflow and more. The format of a single-precision floating-point number is shown in Figure 2.3.

For normalised values, the number, x, represented by the floating-point format is:

$$x = (-1)^s \times 1.f \times 2^{e-e_0}, \tag{2.1}$$

where s is the sign, f is the mantissa and e is the exponent. e_0 is a bias defined in [IEEE08]. In single-precision numbers $e_0 = 127$. For denormal numbers, zero, infinity and NaN, refer to [EL04].

To minimise the complexity of the implementation, one can choose to not implement some features of the standard. Gradual underflow, an expensive treatment of very small numbers, can be avoided by *flushing-to-zero*. Special values like *infinity* and *not a number (NaN)* can also be avoided [VB08]. Choosing a single rounding mode and not implementing exceptions will also keep the cost of implementation low. These optimisations will not only benefit the implementation cost, but the area of the implementation should also be smaller.

For details regarding implementation of floating-point operators, refer to [IEEE08; EL04; Mul+10].

2.3 Design

Here, some important aspects of digital design that are covered in this thesis are introduced.

2.3.1 Design process

In designing a digital system, a typical design process often starts with requirement capture. In this phase, requirements for the system are gathered. The results of this are both high-level product requirements as well as low-level engineering requirements. Once the requirements are set, an iterative phase of architecture exploration starts. Here, high-level analyses of different possible system architectures are performed. The most promising architectures are then analysed further by e.g. modelling in high-level programming languages. The results from this phase is the high-level models, block diagrams and specifications.

Using the specification from the architecture exploration, RTL code is written. The functional correctness of this code is verified using the methods that will be discussed in Section 2.4. In order to measure the power, performance and area (PPA) of the design, synthesis and simulation of the RTL is performed in this phase.

2.3.2 Clock gating

During the design process, low power design techniques can be applied. A common and easily implemented technique is clock gating. This lowers dynamic power consumption of the design by avoiding excess toggling.

2.3.3 False paths

In static timing analysis (STA), a false path is a timing path that never will be exercised in the final design. Should a false path be one of the critical paths in the system, this can potentially lead to routing congestion and reduced performance.

Figure 2.4 can be used to illustrate what a false path is. As the multiplexers in the system are both controlled by the same signal, sel, the path from B to Y is a false path.

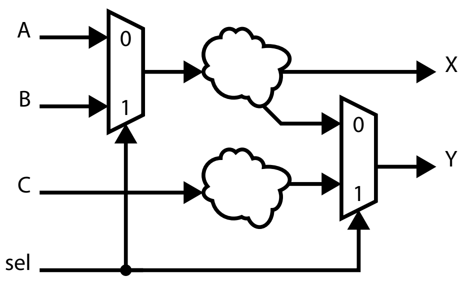


Figure 2.4: Example used in illustrating a false path

2.4 Verification

Functional verification of the design is an important part of the design process. Following a verification plan, various aspects of the design are exercised. Different techniques are often used in verifying different parts of the design. Simulation based verification uses logic simulation with predefined or randomised inputs to exercise the design. Formal verification is a method where stated properties of the design are mathematically proven to be correct.

For simulation, there exist several metrics that say something about the extent to which the design has been exercised. Code coverage says something about how well various parts of the *code* has been exercised by the testbench, and is supported by most logic simulators. While 100 % code coverage means that all the code in the design has been executed, it does not necessarily imply that all functionality has been fully tested. In verifying this, functional coverage can be used. Functional coverage tells us how well the *functionality* has been exercised. SystemVerilog, and thus most electronic design automation (EDA) tools, implement support for functional coverage.

2.5 ZedBoard

The ZedBoard is a development board featuring a Xilinx Zynq System on Chip (SoC) [Avn14]. In addition to this, it is fitted with useful peripherals like 512 MB DDR3 memory, an HDMI transmitter and more.

The SoC on the board is an XC7Z020CLG484-1. This is divided into a *processing system* and a *programmable logic* section [Xil16]. The processing system holds a dual core ARM CPU running at up to 667 MHz. The CPU has a vector floating-point unit (VFPU) for accelerating vector operations. The programmable logic is a regular field-programmable gate array (FPGA). A high-bandwidth AXI interconnect between the sections allows for tight coupling of the two.

2.6 Interconnect

In designing or choosing an interconnect, the main goals are often performance and scalability. There are often trade-offs between latency and throughput of the interconnect.

There are many parameters to a bus protocol. Width of the address, data and control lines play an important role in the performance of the interconnect. Additionally, transfer modes, topologies and arbitration schemes must be designed.

2.6.1 AXI

Advanced eXtensible Interface (AXI) is a high-performance bus standard by ARM. All of Xilinx's intellectual property (IP)-blocks have AXI interfaces. It is often used as the main interconnect in high performance SoCs.

2.6.2 APB

'The APB bus standard defines a bus that is optimized for reduced interface complexity and low power consumption' ([PD10]). Advanced Peripheral Bus (APB) is mainly used for modules with low bandwidth requirements, e.g. status/control registers for peripherals.

2.6.3 Ready/valid handshaking

Ready/valid handshaking is a commonly used handshaking method in digital systems. It is for instance used in the AXI standard [ARM13]. In the AXI standard, there are many rules for how the handshaking process should go down, but these are not universal for any ready/valid interface. The most important feature of this handshaking method is that data is transferred if the slave is *ready* and the master has *valid* data at a clock edge. This event is referred to as a transaction, whereas the term *accept* refers to both ready and valid being asserted. An example transaction is illustrated in Figure 2.5.

Most blocks in this thesis use ready/valid handshaking for their interfaces.

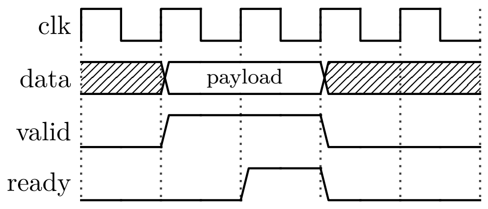


Figure 2.5: Ready/valid handshaking example

Chapter 3

Requirements

Tables 3.2 and 3.3 list the requirements for the specification document and the ray tracer itself. Most of the requirements are taken directly from the project thesis. REQ_DELIV_001 and REQ_DELIV_002 are the only new requirements in this thesis. The tables indicate whether the requirements have been addressed or not and in which thesis they are addressed. A legend for the symbols used is shown in Table 3.1

Table 3.1: Legend for requirement tables

Symbol	**Description**
□	Requirement addressed during project thesis.
■	Requirement addressed during project thesis. No additional work done during master to address the requirement.
○	Requirement addressed during master's thesis.
●	Requirement addressed during master's thesis. Improvements over project thesis work.

Table 3.2: Requirements for the specification document

Requirement ID	**Description**	**Project**	**Master**
REQ_ARCH_001	The specification must list all the requirements for the Hardware Ray Tracer.	□	○ Ch. 3
REQ_ARCH_002	All datastructures for primitives, colors, rays and materials must be specified.	■	
REQ_ARCH_003	The specification must explain what Ray Tracing is.	□	○ Sec. 2.1
REQ_BSPEC_001	The specification must describe the functionality of the engine.	■	
REQ_BSPEC_002	Microarchitecture must be specified. Use block diagrams and other figures.	□	○ Ch. 5
REQ_BSPEC_003	Interfaces must be described.		● Sec. 5.1
REQ_BSPEC_004	The specification must contain an analysis of performance, quantization effects, and scalability. This analysis may involve additional modelling effort. Data from this analysis must prove that the proposed microarchitecture is likely to meet the functional and performance requirements.	□	○ Sec. 5.8
REQ_ISA_001	A suitable Instruction Set Architecture (ISA) must be specified.	□	○ Sec. 5.4
REQ_LEARN_001	One or more submodules with different complexity that are suitable as examples of various challenges and problems in digital design must be identified.	■	
REQ_LEARN_002	One or more problems regarding the development of the Ray Tracer that has been solved by modelling must be presented.	□	○ Ch. 5
REQ_LEARN_003	Propose two solutions for the design of one of the submodules. Explain why one is better than the other.	■	

Table 3.2: Requirements for the specification document (continued)

Requirement ID	Description	Project	Master
REQ_LEARN_004	Identify or deliberately design sub-optimal performance in the system. Create an assignment/case-study out of this.		
REQ_LEARN_005	Create learning examples for interesting problems encountered during the work with the thesis.		● Ch. 5

Table 3.3: Requirements for the Hardware Ray Tracer

Requirement ID	Description	Project	Master
REQ_FUNC_001	The Ray Tracer must be able to execute the ray tracer algorithm given by the python model.	□	○ Ch. 5
REQ_FUNC_002	The Hardware Ray Tracer must be programmable. It must be easy to change the ray tracer program that is running on the Hardware.	□	○ Ch. 5
REQ_FUNC_003	The Ray Tracer must implement the ISA.	□	○ Ch. 5
REQ_FUNC_004	The Ray Tracer must support primitives such as spheres and planes as defined in the architecture specification.	■	
REQ_FUNC_005	The Ray Tracer must support simple animation of frames.	■	
REQ_PERF_001	The Ray Tracer performance must have a scalable performance from 10 frames per second up to 60 frames per second for scenes with one plane and one sphere rendered with VGA resolution. (This could e.g. be achieved through a system with a parameterizable number of cores).	□	○ Sec. 5.8

Table 3.3: Requirements for the Hardware Ray Tracer (continued)

Requirement ID	**Description**	**Project**	**Master**
REQ_PERF_002	The utilization of all functional units should be as high as possible and preferably above 50 % while the ray tracing algorithm is running. This ensures a high performance to area ratio and an efficient use of available resources.		
REQ_TECH_001	The target technology is high end Xilinx FPGAs. The design may contain technology specific optimizations.	☐	○ Ch. 5
REQ_DELIV_001	Write SystemVerilog RTL code for the processing core.		● Sec. 5.6
REQ_DELIV_002	Propose an interconnect that scales well with the number of cores.		● Sec. 5.1

Chapter 4

Project thesis summary

This chapter will give a short summary of the project thesis [Egg16]. A lot of the content in this chapter is taken directly from the project thesis. However, all figures are updated for consistency with the rest of the thesis. The architecture specification, where data structures, message formats and control/status registers are specified, can be found in Appendix A. For more details, refer to [Egg16].

The project thesis takes the reader through the process of designing a hardware ray tracer for educational purposes. Ray tracing is a parallel problem, and is well suited for implementation in hardware. This specific ray tracer is not very feature-rich, but still demonstrates important aspects of digital design. An initial Python model was provided by [Gje17] as a starting point for further development. The model was simplified by removing certain features such as refraction and support for rendering other objects than spheres. This reduces the complexity of the system while keeping the educational value.

In order to model different aspects of the system, two additional models were developed. A transaction-level modeling (TLM) model was created to see how well the partitioned system worked. To verify that the ISA could execute the algorithm, an ISA-simulator along with an assembled version of the algorithm was developed. The models and tools developed during the project thesis have been used extensively throughout the work on both the project and master's thesis.

4.1 Top-level system block diagram

During the work with the project thesis, the foundation for this master's thesis was laid. An important result is the top-level system block diagram shown in Figure 4.1. This diagram shows the system that the ray tracer will be a part of. Blocks that are contained within the green area will be implemented on the FPGA. Blocks partly in the green area may need to be implemented using additional hardware.[1]

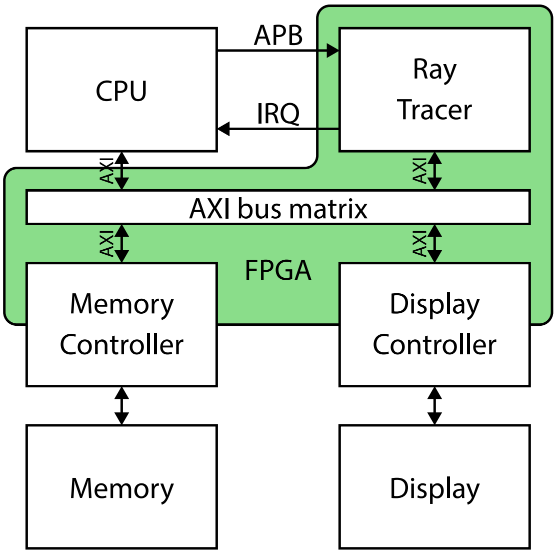


Figure 4.1: Top-level system block diagram

The central processing unit (CPU) generates scenes to be rendered by the ray tracer and saves them in memory. The CPU then commands the ray tracer to render the scene. As the ray tracer renders a scene, the pixels are buffered in memory. Once finished rendering, the display controller will fetch the frame from memory and output it to a display. The CPU can control and read the status of the ray tracer over APB (see commands/statuses in Appendix A.3).

[1]This is clarified in Section 5.1.1.

4.2 Ray tracer block diagram

Figure 4.2 shows the system-level block diagram designed in [Egg16]. All blocks from Figure 4.1 are still shown, the only difference being that the internals of the ray tracer are displayed in the dark cyan area. The partitioning is a result of inspecting the high-level model. Initial rays are generated by the ray manager, and assigned to the cores that perform the actual ray tracing. During ray tracing, the cores read scene data from the object buffers as needed (see Appendix A.1 for data structures). The ray tracer has a parameterisable number of cores in order to have a scalable performance.

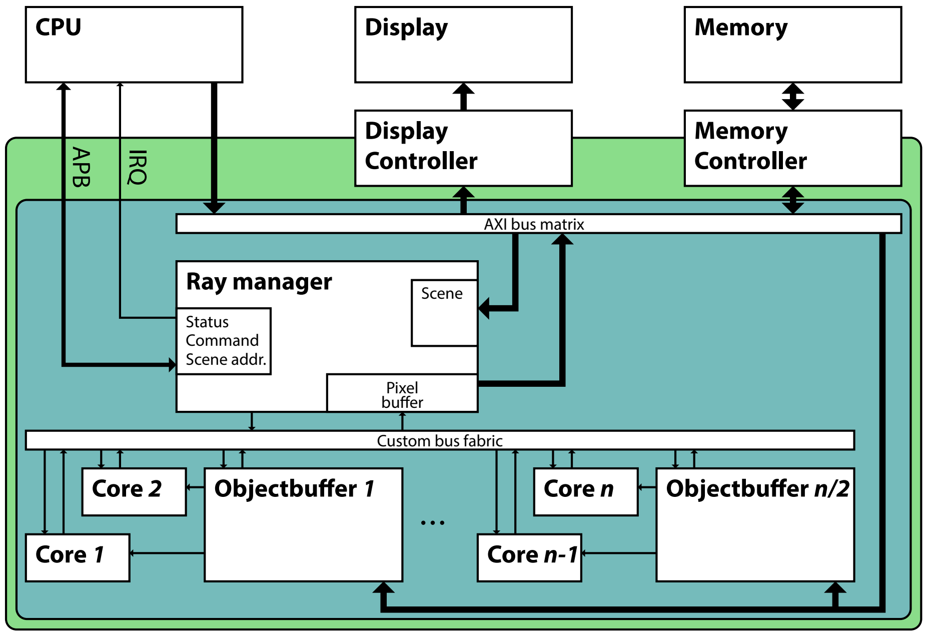


Figure 4.2: Ray tracer block diagram as of [Egg16]. Blocks contained in the green area will be implemented on the FPGA. Blocks contained in the dark cyan area represent the ray tracer.

As discussed, the ray tracer starts rendering once the CPU commands it to start. The CPU does that by telling the ray manager where the scene is located in memory and issuing a *run*-command. Using the scene data, the ray manager will command the object buffers to fetch all objects in memory. When the object buffers are filled, the ray manager will start issuing rays to the cores in the system using the messages defined in Appendix A.2. Once cores finish tracing a ray, they send the resulting pixel back to the ray manager. The ray manager buffers up a few pixels before sending them to the frame buffer located in memory. After all rays have been traced, the ray manager sends an interrupt request (IRQ) to the CPU. The CPU will then tell the display controller to display the frame before starting the process over for the next frame.

4.2.1 Ray datapath

By inspecting the high-level model and taking the partitioning described in Section 4.2 into account, the instruction set shown in Table 4.1 was designed. This was shown to be sufficient to execute the ray tracing algorithm fairly efficiently. Loading of fixed-point data from the object buffers were done in software. Square roots and inverse square roots as well as exponentiations were approximated in software as well. The instruction set is later altered in Section 5.4.1.

Table 4.1: Instruction set from [Egg16]

Name	**Assembly**	**Operation**
Add	`add  $rd, $ra, $rb`	R[rd] = R[ra] + R[rb]
Subtract	`sub  $rd, $ra, $rb`	R[rd] = R[ra] − R[rb]
Shift left logical	`sll  $rd, $ra, x`	R[rd] = R[ra] << x
Shift right logical	`srl  $rd, $ra, x`	R[rd] = R[ra] >> x
Load upper immediate	`lui  $rd, x`	R[rd] = {x, 16'b0}
Or immediate	`ori  $rd, $ra, x`	R[rd] = R[ra] | {16'b0, x}
Subtract immediate	`subi $rd, $ra, x`	R[rd] = R[ra] − x
Multiply immediate	`muli $rd, $ra, x`	R[rd] = R[ra] ∗ x
Load word	`lw   $rd, x[$ra]`	R[rd] = M[R[ra] + x]
Load byte	`lb   $rd, x[$ra]`	R[rd] = {24'b0, M[R[ra] + x]}
Floating-point add	`fadd $rd, $ra, $rb`	$R[rd] = R[ra] \overset{FP}{+} R[rb]$
Floating-point sub	`fsub $rd, $ra, $rb`	$R[rd] = R[ra] \overset{FP}{-} R[rb]$
Floating-point mul	`fmul $rd, $ra, $rb`	$R[rd] = R[ra] \overset{FP}{*} R[rb]$
Branch if equal	`beq  $rd, $ra, x`	PC = x if R[rd] == R[ra]
Branch if positive	`bpos $rd, x`	PC = x if R[rd] >= 0
Branch if negative	`bneg $rd, x`	PC = x if R[rd] < 0

The instruction set has been shown to be executable using the datapath shown in Figure 4.3. An ISA-simulator modelling this datapath was created in [Egg16].

An assembled version of the ray tracing algorithm was successfully run using this ISA-simulator.

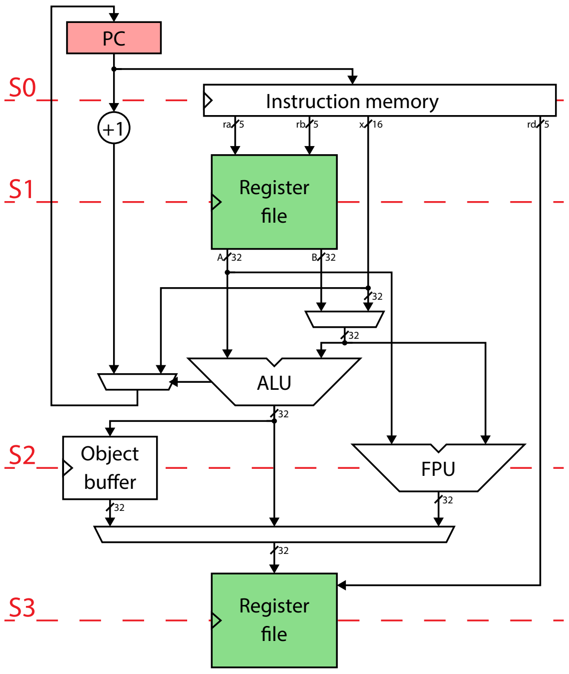


Figure 4.3: Core datapath from [Egg16]. The dotted red lines represent pipeline registers for all signals that pass through them. The two register file blocks represent reading and writing to and from the same physical register file.

4.3 Project thesis results

Using the mentioned models, data used to predict system performance has been extracted. By analysing this data, it was found that achieving a frame rate of 24 FPS at VGA resolution requires 74 cores when running at 50 MHz. Implementing this many cores on an FPGA will present problems in terms of area and design of interconnect. Figure 4.4 shows an example of a scene rendered using the ISA-simulator.

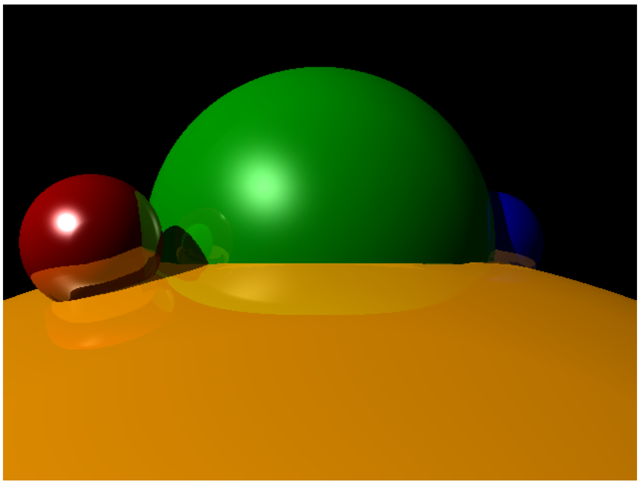


Figure 4.4: Scene rendered using ISA-simulator

4.4 Future work

A proposal for future work was described at the end of the project thesis. The most important points from this list were:

- System-level design. Examine the ZedBoard and Zynq, and determine how system-level blocks maps to the resources on these.
- Analyses and design of thread interleaving scheme.
- Interfaces and interconnect between ray cores and ray manager.
- Interfaces and interconnect between object buffers and ray cores.
- Interfaces and interconnect between object buffers and ray manager.
- Improvements to the ISA.
- Deciding upon final instruction formats.
- Deciding whether to implement the ray manager as a dedicated hardware unit or using a general-purpose processor.
- Implementing a fixed-point to floating-point instruction.
- Implementing the system in RTL.
- Synthesising the RTL for the target FPGA.

Chapter 5

Design process

In this chapter, large parts of the ray tracer will be designed, implemented and tested. This work builds upon what was presented in Chapter 4. System-level design will be discussed in Section 5.1. Here, implementation on the Zynq SoC, as well as refinement of ray tracer blocks will be discussed. Sections 5.2 to 5.4 will go through the design of the thread interleaving scheme, communication protocols and the ISA. Design and modelling of floating-point operators will be covered in Section 5.5. Sections 5.6 to 5.8 cover implementation, verification, synthesis and performance analysis.

The ray tracing algorithm is very parallelisable. In this system, one initial ray is cast per pixel, and these rays are traced by independent threads. Due to this, the terms *thread*, *ray* and *pixel* will be used interchangeably, depending on what suits the situation best.

Rendering of VGA frames at 24 FPS should be assumed where nothing else is explicitly stated. The system is designed around the ZedBoard [Avn14], that features a Xilinx Zynq SoC [Xil16]. This thesis will mainly focus on digital design, and thus the programmable logic part of the SoC.

5.1 System-level design

In [Egg16], the communication between the ray manager and the cores were not fully specified. Figure 4.2 showed a custom bus matrix as well as some wiring between the ray manager and the cores. By itself, this is not enough to enable communication, as the cores themselves has no way of receiving or sending any data. Addressing this issue, this section will focus on interfaces and interconnects.

5.1.1 Ray tracer

In Figure 5.1, the first step towards a working interface between the ray manager and cores is shown. This figure abstracts away the ray cores and their interconnect, as the dual core array will be discussed in Section 5.1.2. Ray requests are generated by the ray generator and sent over a ready/valid interface to the dual core array. As results are calculated, they are sent back to the pixel handler over another ready/valid interface. These interfaces will be discussed in Section 5.3. In this thesis, only the dual core array and its submodules will be implemented, while performance of the ray manager will be analysed.

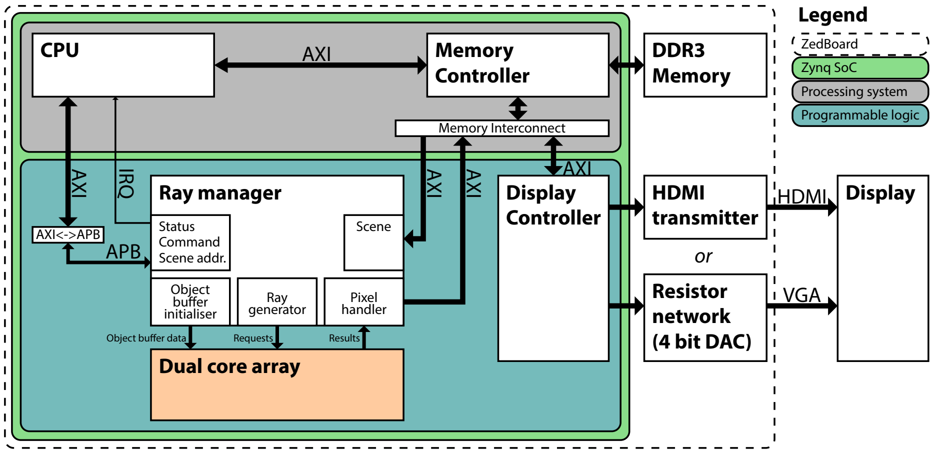


Figure 5.1: Ray tracer block diagram.

As seen from the figure, many components have been moved around and some are added when compared to Figure 4.2. As opposed to that block diagram, this maps directly to resources available on the ZedBoard and Zynq [Avn14; Xil16]. Some other changes within the programmable logic section have also been performed. In [Egg16], each object buffer had an AXI interface in order to fetch data from memory. It was pointed out that this was far from optimal, as each object buffer will fetch the same data. To make this more efficient, propagation of the data was mentioned as a better alternative. This has now been implemented into the design using the object buffer initialiser. This module initialises all object buffers in the dual core array. The object buffer initialisation will be discussed in Section 5.1.5.

As mentioned in [Egg16], the ray manager could be implemented in the CPU, potentially accelerating development. This is possible as the ray manager is a relatively low-throughput module. Performance analyses considering this will be performed in Section 5.8.2.

5.1.2 Dual core array

As mentioned, the dual core array is used to abstract away the ray cores and their interconnect. Figure 5.2 shows the dual core array. It consists of a chain of feeder elements that is the interconnect feeding requests to the cores, as well as a chain of drain elements that routes results out of the array. The dual cores are also chained together, allowing for initialisation of object buffers (see Section 5.1.5).

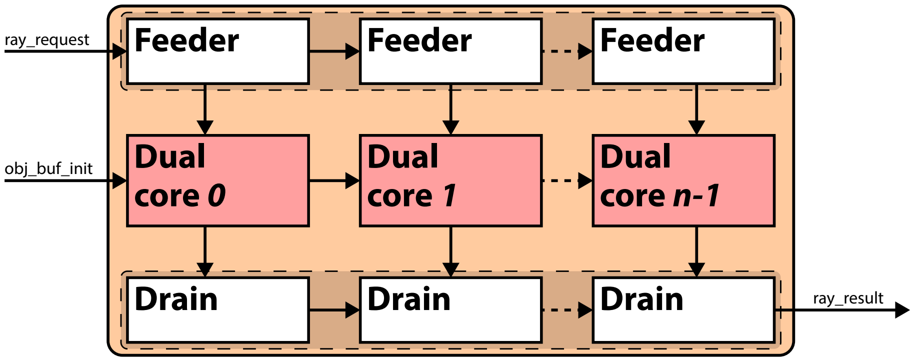


Figure 5.2: Dual core array. Each dual core holds two ray cores, one instruction memory and one object buffer.

Internally, the feeder elements register incoming requests, and passes them on to either a dual core or the next feeder element the next clock cycle. This given that the dual core or the next feeder element is ready to accept a new request. In case both are ready, the dual core has the highest priority. The way this is designed, requests reaching the end of the feeder chain will be stuck waiting for dual core $n-1$. This could have been overcome by looping requests back to the start of the chain. However, as shown by Figure 5.18 in Section 5.8.1, system performance is not greatly affected by this.

The drain elements work in the same way, accepting results from either a dual core or the previous drain in the chain. The result is registered and propagated to the next drain element the next clock cycle. Also here, dual cores are prioritised. Dual cores are given the highest priority as this guarantees that the dual core can start processing a new ray. The dual cores themselves will be further discussed in Section 5.1.3.

Registering the requests and results in the feeder and drain elements helps the synthesis tool avoid routing congestion on the FPGA. The elements break timing in the forward direction (i.e. the valid and payload signals). In Section 5.7.4, experiments with also breaking timing in the reverse direction (i.e. the ready signal) are conducted. The widths of the request and result buses are equal to the width of the messages transmitted over them. This helps in simplifying the design process. The message formats are shown in Sections 5.3.2 and 5.3.3. Internally in the dual core array, all interfaces use ready/valid handshaking.

Learning example 5.1: Avoiding routing congestion

By inserting register slices on the request and result buses, the fan-out of the ray generator is reduced[a]. This lowers the burden of place and route for the synthesis tool by avoiding routing congestion on the FPGA. Had register slices not been used, and purely combinatorial arbitration between the cores applied, the ray generator would have to drive each of the ray cores directly. For this to meet timing, the synthesis tool would have to place all ray cores physically close to the ray generator. In configurations with many cores, this is not physically possible, and would result in the synthesis tool giving up, giving a low operating frequency for the system.

As mentioned, only the ready and payload signals are registered, while a combinatorial ready-path through all register slices are still present. It is assumed that this might become a problem, and register slices that break timing in the reverse direction have been designed. Experiments with these are performed in Section 5.7.4.

[a]Also the fan-in of the pixel handler is reduced. The same arguments as for the fan-out of the ray generator are valid for this.

5.1.3 Dual core

Making dual cores a separate entity was a natural choice. In [Egg16] it was pointed out that block RAM (BRAM) on Xilinx FPGAs are dual ported [Xil13], and to fully utilise this, each instruction memory and object buffer should be shared by two ray cores. As shown in Figure 5.3, a purely combinatorial 'arbitration' between the two cores is used. The core labelled *Ray core 0* has the highest priority for both requests and results. The ray core will be discussed in Section 5.3. The instruction memories and object buffers will be discussed in Sections 5.1.4 and 5.1.5, respectively.

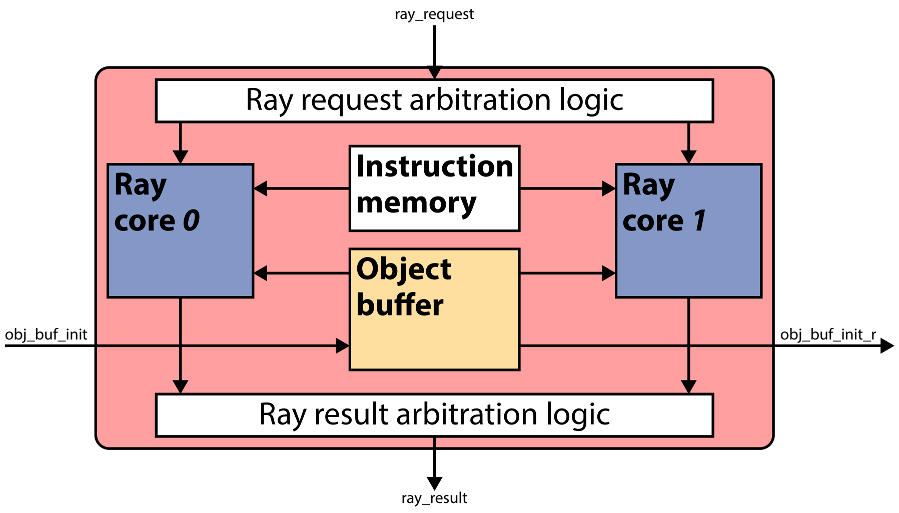


Figure 5.3: Dual core

Learning example 5.2: Performance density

Performance density is a measure of the performance delivered per unit area. Keeping this high is a goal for most digital systems. Here performance density is increased by sharing the same physical instruction memory and object buffer between two ray cores.

5.1.4 Instruction memory

The instruction memory has a simple interface. It performs a synchronous read of the instruction memory every clock cycle. In order for it to map to dual ported BRAM, it has two independent read ports. As instructions are read every clock cycle, no read enable is needed. Figure 5.4 shows an example timing diagram for the instruction memory.

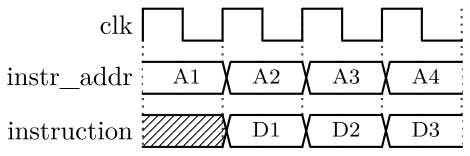


Figure 5.4: Instruction memory interface example waveform

5.1.5 Object buffer

The object buffers play the same role in this system as caches hold in more complex systems: keeping frequently used data readily available to the processing elements. The only difference is that the data held by the object buffers is constant for a whole frame. This is because the simple scenes this system is designed to render are small enough to be stored in their entirety in BRAM close to the ray cores. The object buffer memory layout is shown in Figure A.5.

The read interfaces are simple. Inputs are the object address and a read enable, while the read data is available at the output the following clock cycle. Just like the instruction memory, two cores share the same object buffer in order to increase performance density. The read enable signal is used to increase energy efficiency (see learning example 5.3). In Figures 5.5 and 5.6, signals c0_obj_addr and c1_obj_addr holds the addresses from ray cores 0 and 1, respectively. c0_obj_-read_en and c1_obj_read_en are the read enable signals, while c0_obj_data and c1_obj_data holds the read data.

As seen from Figures 5.2 and 5.3, all object buffers are chained together. This chain is used in propagating initialisation data to all object buffers in the system. The object buffer initialiser (from Figure 5.1) pushes data to the first object buffer in the chain, that stores the data and propagates it onward to the next object buffer and so on. The data on in_init_data is stored to the address in_init_addr when the signal in_init_load is high. These signals are all pipelined through to the next object buffer in the chain using the signals out_init_data, out_init_addr and out_init_load. By chaining and pipelining object buffers, routing congestion is avoided (remember learning example 5.1).

This is all implemented using one BRAM, one multiplexer, a register and a flip-flop (FF). The BRAM is set to write-first synchronisation [Xil13], eliminating the need for registering that as well. As seen in Figure 5.5, port A of the BRAM is shared between the initialisation interface and core 0's read interface. This means that reading and writing using these at the same time is not possible. This is not a problem, as the system architecture specifies that object buffer initialisation is never to occur at the same time as ray tracing.

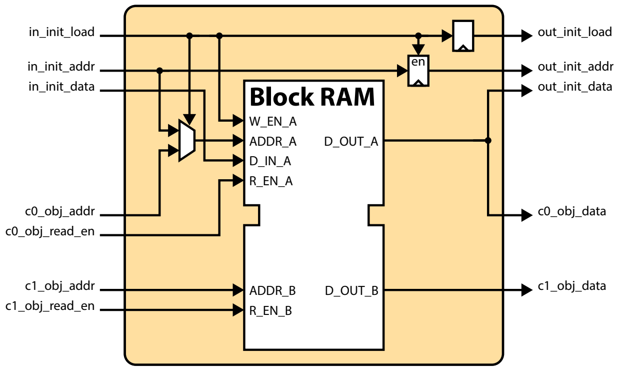


Figure 5.5: Block diagram of object buffer

Learning example 5.3: Improving energy efficiency

In digital circuits, dynamic power is consumed when signals toggle (remember Section 2.3.2). Due to this, unnecessary toggling of signals should be avoided. In this section, a read enable was added to the object buffer, keeping the output of the BRAM unchanged when no read is needed. We even have a clock enable on the register for the address, further improving energy efficiency (see Figure 5.5).

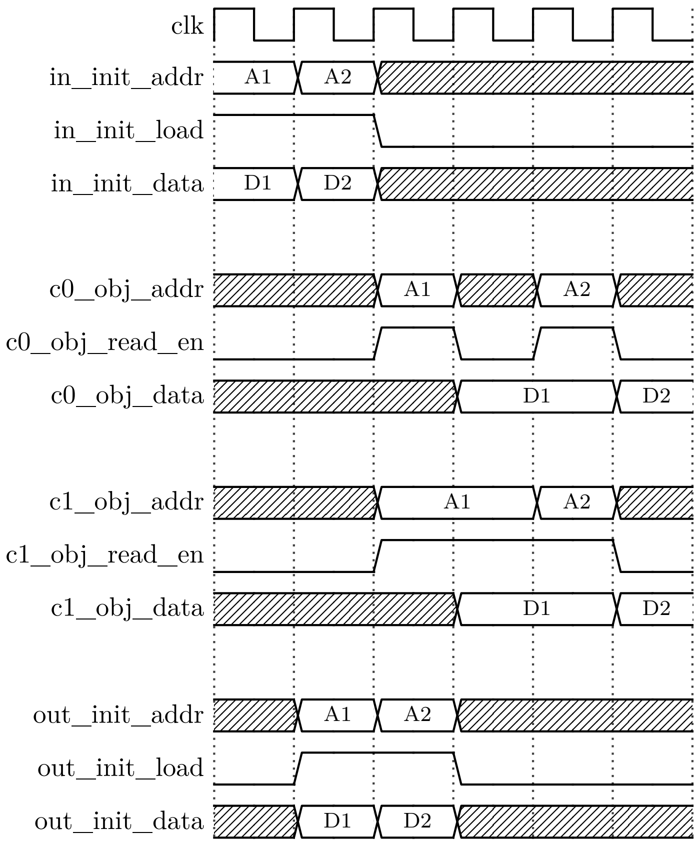


Figure 5.6: Object buffer interfaces example waveform

5.2 Thread interleaving

In the ray datapath presented in [Egg16] (Figure 4.3), both data hazards and control hazards pose a problem. [Egg16] proposed interleaving of 4 threads in a round-robin fashion to overcome the data hazard, while the control hazard was not discussed. However, the control hazard is also eliminated by interleaving in the way that was proposed. This multithreading technique is commonly referred to as *interleaved multithreading* [LGH94], a variant of *fine-grained multithreading* [HP12].

As previously mentioned, all threads are independent. This is what enables thread interleaving to be effective in eliminating the hazards. Both hazards will be discussed, starting with the data hazard.

The data hazard in this architecture is a read after write (RAW) data hazard [HP12; PH14]. A RAW hazard occurs when an instruction tries to access a result of a previous instruction, where the result is not yet calculated or available in the register file. Utilising thread interleaving, this situation is avoided by interleaving processing of independent threads. By interleaving a sufficient number of threads, it can be guaranteed that the result of an instruction is stored in the register file by the time the next instruction in the same thread fetches its operands. This does however require storing multiple program counters (PCs) and register file segments.

In the datapath shown in Figure 4.3, operands are read in stage S1 and stored in S3. This means that interleaving of two threads would be enough to eliminate the RAW hazard. However, [Egg16] specified that the register file was to be implemented in BRAM. To map directly to BRAM, data written is not available for reading before the next clock cycle. Xilinx refers to this mode of read/write synchronisation as read-first [Xil13]. This means that one will have to interleave at least three threads to avoid the data hazard. This has been rounded up to four, allowing for a 2 bit thread id to keep track of the thread executing in each pipeline stage. This id is used in selecting the thread PC and register file segment. The effects of using write-first synchronisation or even LUTRAM is discussed in learning example 5.5.

The control hazard is also a result of the architecture shown in Figure 4.3. The PC is read in S0, and updated in S2. This means that interleaving of two threads is enough to eliminate also this hazard.

Table 5.1 illustrates how different threads are being executed in the different pipeline stages. The PC of each thread is also shown, demonstrating how it is updated in S2 (but first visible in S3).

Table 5.1: Illustration of thread interleaving. #N indicates the Nth clock cycle. S corresponds to the stages in Figures 4.3 and 5.12. T is for thread and PC refers to the different program counters.

	#0	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**
S0	T0	T1	T2	T3	T0	T1	T2	T3
S1		T0	T1	T2	T3	T0	T1	T2
S2			T0	T1	T2	T3	T0	T1
S3				T0	T1	T2	T3	T0
PC0	0	0	0	1	1	1	1	2
PC1	0	0	0	0	1	1	1	1
PC2	0	0	0	0	0	1	1	1
PC3	0	0	0	0	0	0	1	1

Learning example 5.4: Dependency removal

Thread interleaving is one method that can be applied to remove the effects of pipeline dependencies. What other methods are there, and could they be used in this system? How would using these methods impact the performance and complexity of the system?

Alternatives to thread interleaving include code scheduling, pipeline stalling and operand forwarding. By using operand forwarding, only the data hazard is resolved. The control hazard could then have been solved by e.g. branch prediction or a branch delay slot [PH14]. Thread interleaving was chosen as it completely eliminates the need for stalling, giving higher and predictable performance.

In considering other methods, the performance (density) impact of these would have to be evaluated. Analyses using pen and paper as well as simulations would be the way to go.

Learning example 5.5: Register file implementation

There are many different properties to consider when implementing the register file. In [Egg16], a BRAM mapped register file with read-first synchronisation was specified. Here, a short comparison of the different alternatives is presented. Where adding pipeline stages is discussed, these are to be placed between the read and write of the register file (i.e. between

S1 and S3 in Figure 4.3).

- Keeping the implementation from [Egg16] is naturally the first possibility.
- Keeping the register file as it is, but introducing another pipeline stage could improve performance of the system.
- Changing the synchronisation mode to write-first. This would require some additional logic in the register file. Using this synchronisation mode gives two alternatives:
  - Add two additional pipeline stages, potentially increasing f_{max} significantly.
  - Go down to only interleaving two threads.
- Implementing the register file using LUTRAM and using asynchronous read. This gives the same alternatives as BRAM with write-first synchronisation gave.

The effects of implementing these alternatives would have to be analysed and simulated in order to conclude which is 'best'. Performance and area are important keywords in this analysis.

5.3 Ray core

The ray core is essentially a wrapper for the ray datapath, providing ready/valid interfaces for incoming ray requests and outgoing results. It manages and keeps track of each of the four threads in the ray datapath. A block diagram showing the ray core is shown in Figure 5.7.

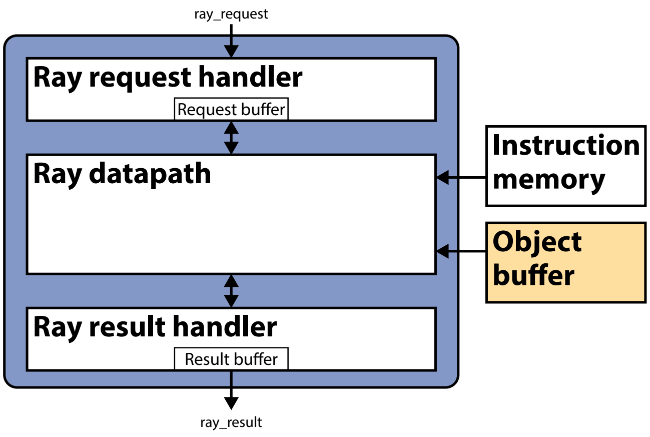


Figure 5.7: Ray core block diagram

The ray request handler accepts ray requests into the request buffer and assigns them to idle threads. The ray datapath then starts tracing the ray from the request. Once finished tracing, the ray datapath loads the resulting pixel colour into the result buffer, and the ray result handler sends the colour out on the result interface. The following sections will describe these interfaces in more depth.

5.3.1 Commands and statuses

Communication between the ray core and the ray datapath is facilitated by the use of command and status registers for each thread in the system. The legal commands and statuses are shown in Tables 5.2 and 5.3, respectively.

Table 5.2: Commands

Command	**Encoding**	**Description**
NONE	00	The core should keep doing what it is doing
START	01	The core should start processing a ray
RESBUF_GNT	10	The core is granted access to the result buffer

Table 5.3: Statuses

Status	**Encoding**	**Description**
IDLE	00	The core is idle
BUSY	01	The core is tracing a ray
RESBUF_REQ	10	The core requests access to the result buffer

The commands are issued by the request and result handlers, while statuses are reported by the ray datapath. The state of each thread is made up of a command, a status and the coordinates of the pixel being rendered by that thread. This is all held in a table used for tracking the state of the individual threads. An example of this is shown in Table 5.4.

Table 5.4: Table for thread state tracking. The table holds example data.

Thread	**x**	**y**	**Command**	**Status**
0	12	20	NONE	BUSY
1	12	28	RESBUF_GNT	RESBUF_REQ
2	12	30	NONE	RESBUF_REQ
3	12	19	START	IDLE

Learning example 5.6: Assertions

Not all combinations of commands and statuses are legal. For example, RESBUF_GNT should only be given to exactly one thread at a time, and only if the thread actually requested it. Another example is that a BUSY thread should never be given a START command. Assertions for these and other properties can been written and used in the system testbenches. Some of these properties can be proven using formal methods.

5.3.2 Ray request handler

The ray request handler accepts ray requests on a ready/valid interface. The ray request format (see Figure 5.8) was designed in [Egg16], and holds the pixel coordinates of the ray (`x`, `y`) as well as the ray direction as a vector of three single-precision floating-point numbers.

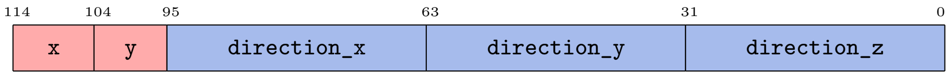


Figure 5.8: Ray request format

Once a ray request is accepted, it is stored in a buffer. The request handler assigns the ray to a thread by copying the pixel coordinates into the thread's row in the thread state tracking table and setting the command to START. The thread in ray datapath will then start executing, and starts off by executing the POP instruction three times, reading in each of the ray direction components. Once this is done, the thread sets its status to BUSY, and the request interface is ready to accept new requests again.

5.3.3 Ray result handler

The ray result handler is responsible for sending pixel colours from the ray datapath out on the result interface. This is also a ready/valid interface. The ray result format is shown in Figure 5.9, and was also discussed in [Egg16]. Here, the pixel colour is represented by an RGB triplet of 8 bit unsigned integers. The result handler stores the result in a result buffer until it is accepted on the result interface.

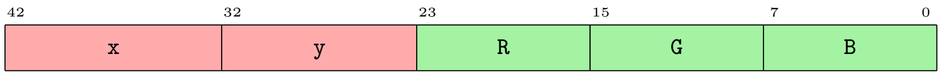


Figure 5.9: Ray result format

When a thread has successfully traced a ray, it will request access to the result buffer by setting its status to RESBUF_REQ. Should the ray result handler be idle, it will grant access by setting the thread command to RESBUF_GNT. At the same time, it will initialise the result buffer with the pixel coordinates saved in the thread state table. Now, the thread will execute the PUSH instruction thrice, to store each RGB component in the result buffer. The thread is now freed, and is ready to accept new requests. The result handler is ready to accept new results once the buffered result has been sent out on the result interface.

5.4 Ray datapath

The ISA and ray datapath designed in [Egg16] has some shortcomings. Most importantly, the ray datapath had no means of communicating with the rest of the system. Furthermore, the proposed way of eliminating pipeline dependencies was not integrated in the architecture. In this section, both shortcomings are addressed based upon the discussions in Sections 5.2 and 5.3.

The instruction set has been extended and altered compared to Table 4.1, resulting in increased system performance. The new instruction set and ray datapath is shown in Table 5.5 and Figure 5.12.

5.4.1 Instruction set

The instruction set presented in [Egg16] has been extended to enable communication with the request and result handlers of the ray core. Some other alterations enabling more compact and thus faster executing assembly code have been made. Table 5.5 shows the current instruction set. The final assembly code is shown in Appendix B.

Table 5.5: Instruction set

Name	**Assembly**	**Operation**	**Format**	**Opcode**
Add	`add $rd, $ra, $rb`	R[rd] = R[ra] + R[rb]	R	00000
Subtract	`sub $rd, $ra, $rb`	R[rd] = R[ra] − R[rb]	R	00001
Shift right logical	`srl $rd, $ra`	R[rd] = R[ra] >> 1	R	00010
Load upper immediate	`lui $rd, x`	R[rd] = {x, 16'b0}	I	00011
Add immediate	`addi $rd, $ra, x`	R[rd] = R[ra] + x	I	00100
Subtract immediate	`subi $rd, $ra, x`	R[rd] = R[ra] − x	I	00101
Multiply immediate	`muli $rd, $ra, x`	R[rd] = R[ra][15:0] ∗ x	I	00110
Load word	`lw $rd, x[$ra]`	R[rd] = M[R[ra] + x]	I	00111
Load byte	`lb $rd, x[$ra]`	R[rd] = {24'b0, M[R[ra] + x]}	I	01000
Load FX1.7	`lfx $rd, x[$ra]`	R[rd] = fx2fp(M[R[ra] + x])	I	01001
Floating-point add	`fadd $rd, $ra, $rb`	R[rd] = R[ra] $\overset{\text{FP}}{+}$ R[rb]	R	01010
Floating-point sub	`fsub $rd, $ra, $rb`	R[rd] = R[ra] $\overset{\text{FP}}{-}$ R[rb]	R	01011
Floating-point mul	`fmul $rd, $ra, $rb`	R[rd] = R[ra] $\overset{\text{FP}}{*}$ R[rb]	R	01100
Branch if equal	`beq $ra, $rb, x`	PC = x if R[ra] == R[rb]	B	01101
Branch if positive	`bpos $ra, x`	PC = x if R[ra] >= 0	B	01110
Branch if negative	`bneg $ra, x`	PC = x if R[ra] < 0	B	01111
Branch if command	`bceq y, x`	PC = x if y == command	B	10000
Set status	`stat x`	status = x	I	10001
Pop request buffer	`pop $rd`	R[rd] = reqbuf; reqbuf «= 32	R	10010
Push result buffer	`push $ra`	resbuf = {resbuf[..], fp2uint(R[ra])}	I	10011
Floating-point add imm	`faddi $rd, $ra, x`	R[rd] = R[ra] $\overset{\text{FP}}{+}$ {x, 16'b0}	I	10100
Floating-point mul imm	`fmuli $rd, $ra, x`	R[rd] = R[ra] $\overset{\text{FP}}{*}$ {x, 16'b0}	I	10101
Add upper immediate	`addui $rd, $ra, x`	R[rd] = R[ra] + {x, 16'b0}	I	10110

The differences between the new instruction set and the one designed in [Egg16] are summarised in Table 5.6.

Table 5.6: Alterations of instruction set

Mnemonic	**Status**	**Comment**
SRL	Modified	*Shift right logical* is only used in approximations, and always used to shift by 1 bit. It has therefore been replaced by a constant right shift of 1 bit. In Table 4.1, it was a parameterisable shift by `x`.
SLL	Removed	*Shift left logical* was only used for converting fixed-point numbers to floating-point numbers. As a new instruction (LFX) that combines loading and converting a fixed-point number to floating-point has been introduced, SLL has been removed.
ORI	Removed	*Or immediate* was used for loading constants and setting bits. The same functionality can easily be achieved using ADDI.
ADDI	New	*Add immediate* is used for loading constants. This replaces ORI, as it is more versatile. Additionally, the required required is already in place.
MULI	Modified	*Multiply immediate* is now a 16×16 bit multiplier instead of 32×16 bit. This was changed as a result of inspecting the assembly code, and finding that the numbers to be multiplied would never require a 32×16 bit multiplier. This change allows the multiplication to be implemented using only one DSP element instead of two for the wider multiplier.
LFX	New	This instruction is used to load FX1.7 numbers from the object buffer and convert them to floating-point. The implementation of this is discussed in Section 5.4.2.
BCEQ	New	*Branch if command* is added to allow a thread to act upon commands from the request and result handlers. These commands were shown in Table 5.2.
STAT	New	This instruction sets the status of the currently executing thread. The legal statuses were shown in Table 5.3.

Table 5.6: Alterations of instruction set (continued)

Mnemonic	**Status**	**Comment**
POP	New	*Pop request buffer* is used to read in the ray direction of a request. This instruction must be executed once per component of the ray direction, i.e. three times in total. The 32 LSBs are shifted out of the request buffer and read by the core. This was further discussed in Section 5.3.
PUSH	New	*Push result buffer* converts the floating-point operand into an 8 bit unsigned integer and pushes it to the result buffer. The conversion between floating-point and the unsigned integer is discussed in Section 5.4.3.
FADDI	New	*Add floating-point immediate* does not need any new hardware, as it uses the same muxing logic as LUI. It saves one instruction when performing the inverse square root approximation.
FMULI	New	*Multiply floating-point immediate* does not need any new hardware, as it uses the same muxing logic as LUI. This saves another instruction when performing the inverse square root approximation.
ADDUI	New	*Add upper immediate* does not need any new hardware, as it uses the same muxing logic as LUI. This saves an instruction when performing the square root approximation.

As seen in Table 5.6, many new instructions have been added. The instructions BCEQ, STAT, POP and PUSH were added first. These are the instructions that allow communication between the datapath and the ray core. After adding these, the instruction set contained 20 instructions, and the opcode width had to be increased from 4 to 5 bit. Having increased the opcode width, adding even more instructions were not seen as a problem. LFX, FADDI, FMULI and ADDUI were then added. These instructions are used in optimising the assembly code for speed and size.

In [Egg16], four different instruction formats were considered (R, I, X and Y). With new instructions, X and Y have been swapped for the new instruction format B. The instruction formats are shown in Figure 5.11, and Table 5.5 shows what instruction format each instruction uses. In the B-format, a new field (y) has been added. This field is used by the BCEQ instruction.

The ISA-simulator has been updated to reflect the discussed changes. Statistics generated by the simulator has been used to analyse the effects of the modifications to the ISA. Figure 5.10 shows a comparison of the total number of instructions executed in order to render the scene in Figure 4.4 given different revisions of the code and ISA. As seen from this, the addition of communication gave an increase of 2.22 % in the number of instructions. Adding communication to the ray datapath was a necessity, and could not have been avoided. Due to this, one can argue that the optimisations of the code and ISA has given a decrease of 9.82 % in the number of instructions.

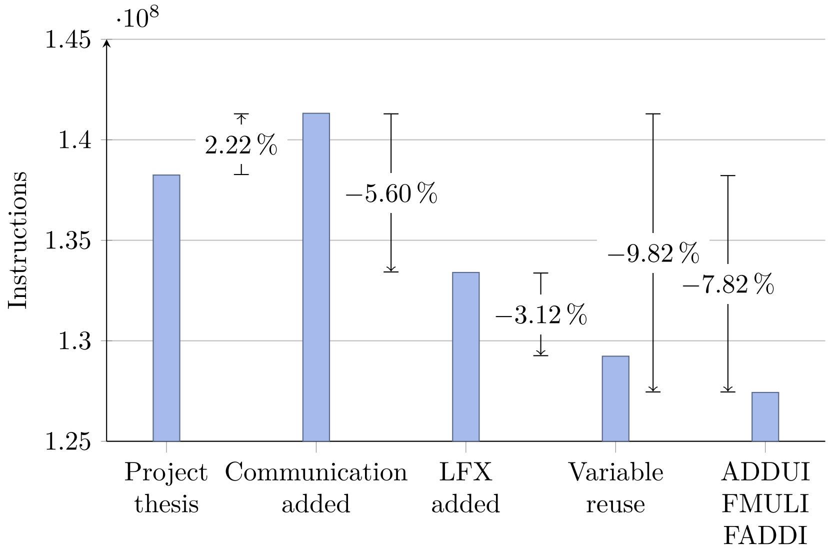


Figure 5.10: Number of instructions used in rendering a VGA frame given different revisions of the code and instruction set. Annotations show the relative change from one particular revision to another. The data is extracted from the ISA-simulator, and does not model congestion nor the ray core. Numbers used in creating this figure can be found in Table B.1.

By reusing variables, redundant calculations of the same values were removed. E.g. memory address calculations were performed way more often than necessary. The redundant calculations allowed for register reuse and better readability [Egg16]. Inspecting the assembly program after LFX was added, it was found that one register was never used. Due to this, a commonly used base memory address was assigned to this register. This change gave a significant reduction in the number of instructions by 3.12 % at the cost of lower code readability.

In addition to reducing the execution time of the assembly program, the optimisations have made the assembly program shorter. The project thesis revision of the assembly program, was 275 instructions long, while the optimised version is 229 instructions long. This has allowed halving the address space for the instruction memory from 9 bit to 8 bit. This is reflected in the PC width and the `x` field of the B-format (see Figure 5.11).

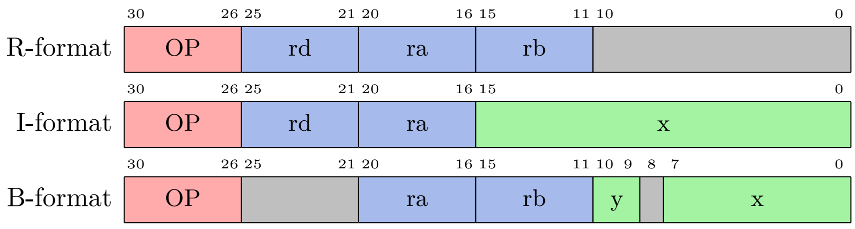


Figure 5.11: Instruction formats

5.4.2 Fixed-point to floating-point conversion

The material descriptor shown in Appendix A.1 makes heavy use of fixed-point numbers. The FX1.7 fixed-point format has 1 bit reserved for the integer part and 7 bit for the fractional part. Loading these from memory and converting them to floating-point was previously done using the software routine shown in Listing 5.1. Now, this sequence can be replaced by the new instruction `lfx $18, 12[$23]`. As shown in Figure 5.10, introducing LFX reduces the total execution time by $\sim 5.60\,\%$.

Listing 5.1: Excerpt of assembly code showing the fixed-point to floating-point conversion routine from [Egg16].

```
lb   $18, 12[$23]      ; load k_refl as fx1.7
ori  $18, $18, 0x8000 ; put in 2.0
sll  $18, $18, 15      ; shift it up to the right position
lui  $28, 0x4000       ; load 2.0
fsub $18, $18, $28     ; sub loaded 2.0 to get k_refl as FP32
```

Implementing this instruction in hardware is done by appending a small block to the output of the object buffer. It was found that the bits from the FX1.7 format more or less map directly into the format of a single-precision floating-point number. The small differences should be handled well by the synthesis tool. Due to this, a case statement mapping all 256 different FX1.7 numbers to their corresponding floating-point values was made.

Standalone synthesis of this module requires 12 look-up tables (LUTs). When instantiating this module in the ray datapath, no results for the module are reported by the synthesis tool. This is probably due to the module being absorbed into surrounding logic, reducing the area to virtually 0.

Learning example 5.7: Hardware/software codesign

'[T]he first release of a product may contain a sizable software component (for time to market and flexibility reasons) while later releases may implement part of this software in hardware for performance and/or cost reasons' ([MG97]).

This is exactly what has been demonstrated in this section.

5.4.3 Floating-point to integer conversion

The PUSH instruction requires floating-point numbers to be converted to 8 bit unsigned integers. Here, 0.0 should map to the integer 0, and 1.0 should map to 255. Numbers higher than 1.0 should saturate to 255. I.e. the operation to be performed is $x = \min\{\text{round}(y \times 255), 255\}$, where y is the floating-point number and x the resulting integer.

Investigating the floating-point number representation, it was found that by adding 1.0 to y, bits 15 to 22 of the result is given by $\text{round}(y \times 256)$. When y is greater than 1.0, this can be detected by checking the exponent of the addition result.

Implementing this requires very little logic if using the floating-point unit (FPU) to do perform the addition. The extra logic needs to check the exponent of the addition result, and explicitly set x to 255 if an overflow occurred.

The method described here scales the resulting colours in the rendered image by 256 instead of 255. This error is considered to be negligible in comparison with the errors introduced by using the approximations to the square root and inverse square root operations discussed in [Egg16].

5.4.4 Further optimisations

Analysis of data from the ISA-simulator has shown that the SUBI instruction was executed 5 611 174 times when rendering a frame using the final version of the ISA. By inspecting the assembly code, it can be seen that the SUBI instruction is always followed by a branch instruction. This means that the SUBI could be replaced by one or several *decrement and branch* instructions, which would reduce the total number of instructions by $\sim 4.40\,\%$. This has not been implemented in the current ISA.

Learning example 5.8: Decrement and branch

The decrement and branch instruction is a common instruction in many ISAs. Implementation of this could make for an interesting student discussion and/or assignment.

5.4.5 Block diagram

Figure 5.12 shows the ray datapath that has been designed to enable execution of the ISA with interleaving of threads. Comparing this to the datapath from [Egg16] (Figure 4.3), it is clear that some changes has taken place. Thread interleaving has been incorporated using a 2 bit thread id in indexing the PC array and register file. The thread id is also used as an index when reading commands and updating statuses (see Table 5.4).

A fixed-point to floating-point conversion unit, FX2FP, has been appended to the object buffer, enabling implementation of the LFX instruction. Reading the request buffer and writing to the result buffer is handled by the two new blocks with the same names. Between the FPU and result buffer, a block named FP2UINT is inserted. This handles saturation and bit selection of the FPU result according to the method described in Section 5.4.3.

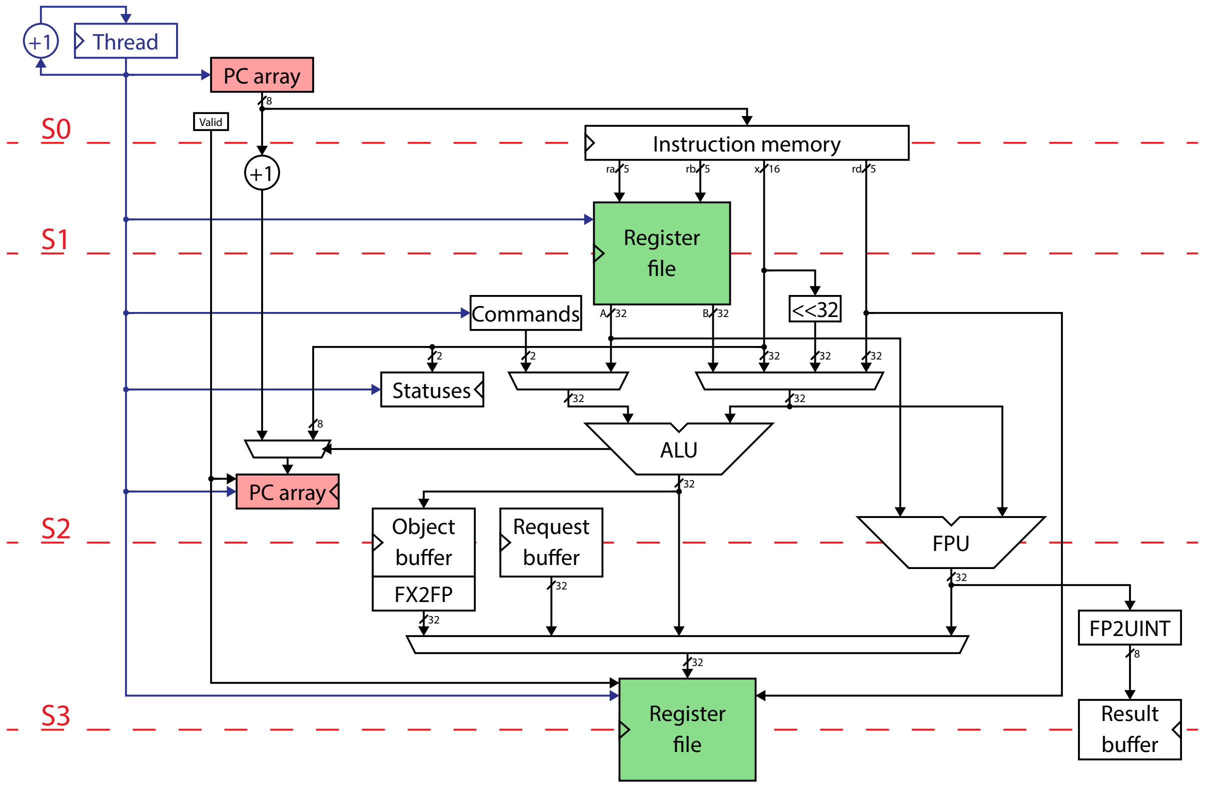


Figure 5.12: Ray datapath. The dotted red lines represent pipeline registers for all signals that pass through them. The two register file blocks represent reading and writing to and from the same physical register file. The same applies for the PC array.

5.5 Floating-point unit

In this section, design of the floating-point operators in the instruction set will be discussed. The instruction set makes use of floating-point addition, subtraction and multiplication. Addition and subtraction is implemented as one combined unit, while multiplication is implemented on its own.

In order to do this, the IEEE floating-point standard [IEEE08] and [EL04; Mul+10; Fie16; Bru09] were studied. Operators that are fully compliant to the IEEE standard may not be necessary for this application, and restricting the feature set of the implementations will be discussed. Python models of the operators has been developed and tested in the ISA-simulator.

5.5.1 Required features

Proper handling of denormal numbers is generally referred to as expensive both in terms of area, complexity and time [Mul+10]. Because of this, omitting support for denormal numbers should be considered for this application. Handling of the special values *inf* and *NaN* is also costly in the same way as denormals. Proper rounding of results can be both complex and costly in terms of area and execution time as well.

In considering omitting support for denormal numbers, the ISA-simulator was used. The simulator was extended to report how often denormal numbers occurred as operands or results of the floating-point operators. This revealed that denormal numbers rarely appeared. This motivated an attempt to see what happened if denormal numbers were treated as zero. Denormal inputs were treated as zero and denormal results explicitly set to zero[1]. Rendering a full scene after these alterations yielded the exact same rendered image as before. Due to this discovery, implementing support for denormal numbers was deemed unnecessary.

Neither *NaN* nor *inf* are used by the program in any way, and would result in undefined behaviour. Because of this, support for neither will be implemented.

Using native Python floating-point operators, it is not possible to change to another rounding mode than the default (round to nearest, ties to even). Due to this, the effects of disabling rounding could not be modelled at this stage, and will rather be considered in Section 5.5.2.

[1]This is referred to as flushing-to-zero [Mul+10].

5.5.2 Modelling

Models of the floating-point operators have been developed in Python. Modelling the arithmetic operators in Python is advantageous. First off, the model can easily be verified by comparing results with native operators. This was done for both random inputs and ranges of numbers. Additionally, the floating-point operators could easily replace the native Python operators in the ISA-simulator. An example frame was rendered using the simulator, and produced exactly the same results as the native operators. Seeing that the models produced correct results for both random inputs and inputs from the ISA-simulator, the models were presumed to be functionally correct.

Having plugged the operator models into the ISA-simulator, the effects of disabling rounding[2] was investigated. Rendering a frame with rounding disabled resulted in some anomalies in the rendered frame compared to that of Figure 4.4. For a few pixels in the frame, each RGB component has been shown to differ by ± 1. The visual quality of the rendered image is not affected by this. The performance impact of disabling rounding is discussed in Section 5.7.1.

Learning example 5.9: Sign of add/sub result

Determining the sign of a multiplication product is simple, you just have to XOR the operand signs.

Determining the sign of a result from the add/sub operation is more complex. The sign depends upon the signs of the operands, the operation being performed and the relative magnitude of the operands; it is a function of 7 bit. Deriving the truth table for this is a cumbersome and mind twisting task. I made a small script that by the help of random multiplicands generated this truth table based upon the sign generated by the FPU in my computer.

[2]Disabling rounding is essentially the same as the rounding mode *round toward 0*.

5.6 Implementation

RTL for the dual core array (Section 5.1.2) and all its submodules has been implemented. The RTL implementation has been found to successfully run the assembly program from [Egg16] with the alterations discussed in Section 5.4.1 (shown in Appendix B). Simulating the dual core array has been found to produce the exact same results as the ISA-simulator did.

5.6.1 RTL

RTL for the dual core array and all its submodules has been implemented using SystemVerilog. In doing this, some of Xilinx's coding guidelines have not been followed, mainly to make the RTL more technology independent and synthesisable for e.g. application-specific integrated circuits (ASICs). This is further discussed in learning example 5.10. To infer BRAMs and DSP elements, inference patterns described in [Xil13] were followed.

Learning example 5.10: Technology independence

Xilinx has many coding guidelines for their FPGAs that conflict with those of ASICs [Xil13]. Some of these are reproduced here:

- 'To initialize the content of a Register at circuit power-up, specify a default value for the signal modeling it.'
- 'Avoid operational set/reset logic whenever possible. There may be other, less expensive, ways to achieve the desired effect, such as taking advantage of the circuit global reset by defining an initial contents.'
- 'Always describe the clock enable, set, and reset control inputs of Flip-Flop primitives as active-High. If they are described as active-Low, the resulting inverter logic will penalize circuit performance.'

These guidelines are not followed in this thesis, mainly in order to make the RTL more technology independent, and synthesisable for e.g. ASICs. As stated by the guidelines, this might lead to lower than optimal performance.

Implementing the modules in RTL was a relatively quick process, as the interfaces and functionality of all modules had been specified prior to implementation. Further accelerating development, the FPU was initially implemented using a behavioural model. This helped take focus away from the details of the floating-point units during verification of the system-level functionality.

The behavioural model of the floating-point unit was implemented using SystemVerilog's shortreal operators. After verifying the system-level functionality, synthesisable RTL for the FPU was implemented. Also this was an uncomplicated process, as the Python models developed in Section 5.5 was at a low level. Pipeline stages were not part of the Python model, and were positioned in what was assumed to be the middle of the floating-point modules.

Learning example 5.11: Behavioural models

Behavioural models in the RTL can be really helpful when it comes to getting test and verification started early. It helps take focus away from the details of submodules, while still allowing verification of system-level behaviour.

5.6.2 Assembler

The assembly program developed in [Egg16] and Section 5.4.1 has to be translated into machine code. In order to do this, an assembler has been developed. The assembler created here is a two-pass assembler. During the first pass over the assembly program it builds a table of all labels found in the program. The second pass assembles each instruction using the table from the first pass.

The machine code is built according to Table 5.5 and Figure 5.11. To stay up to date with the RTL code, the assembler reads out the opcode mappings from the Verilog code. The assembled machine code is saved to a file that is used to initialise the instruction memories.

Learning example 5.12: Single source of constants

The opcode mappings are written down in a type package common for the whole ray tracer. The assembler analyses the Verilog type package in order to always use the current opcode mappings. This is a technique that can help avoid unnecessary debugging due to inconsistencies between different versions of code.

5.6.3 Verification strategy

Each of the submodules have been individually verified. Starting from modules within the ray datapath and all the way up to the dual core array. Some modules have been verified using testbenches and some using formal verification. All RTL simulations has been performed using QuestaSim 10.5c by *Mentor, a Siemens Business*[3], while formal verification is performed by OneSpin 360 - Version 2016_12(74) by *OneSpin Solutions GmbH*. Where whole frames have been rendered, the scene seen in Figure 4.4 has been used. Assertions were written for critical assumptions in the code.

Ray datapath

The first RTL module that was implemented and tested was the ray datapath. To test this module, some of the ray core functionality had to be mimicked by the testbench. Additionally, behavioural models of an object buffer and an instruction memory was developed for this. All instructions were tested using simple and small assembly programs created especially for this purpose. Some implementation errors were identified and corrected using this method.

Testing the ray datapath more exhaustively, the full ray tracer program was run for a single pixel using a single thread. In order to do this, the ISA-simulator was altered so that it could dump the contents of its object buffer. This data was used in initialising the behavioural model of the object buffer. A ray request along with its corresponding ray result was also extracted from the simulator. The testbench mimicked the functionality of the ray request handler in order to feed the request in to the ray datapath. Simulating this returned the exact same colour as the simulator did for that pixel.

As mentioned, this only executed using one thread, while the three other threads were idle. Extending this testbench to test all threads were seen as unnecessary, as this will be easier to test using the testbench for the ray core.

Ray core

The ray core was tested with more than one request, allowing processing by all threads in the datapath. This was enabled through further extending the ISA-simulator. The simulator was now used to dump all requests and results for a full frame. This was fed in through the request interface of the ray core and

[3]Formerly Mentor Graphics, Inc

results were captured at the result interface. A tool used for comparing the results of the simulator and the RTL model was made. This showed that the results from the RTL model were identical to the ones of the simulator. The behavioural models of the object buffer and instruction memory were used here as well.

Dual core

The dual core was easily verified using essentially the same testbench as for the ray core, as it implements the same interface for requests and results. The instruction memory was removed from the testbench, as the dual core implements its own instruction memory. The dual core also implements an object buffer. In order to fill this, the testbench had to mimic an object buffer initialiser.

Dual core array

Before attempting to test the full dual core array, the feed and drain elements of the dual core array were verified using formal verification tools. See learning example 5.13 for more about this. The dual core array was also found to render the whole scene identically to the ISA-simulator.

Learning example 5.13: Formal verification

For the feeder and drain elements, no testbenches were made. These modules were verified exclusively using formal tools. This made verification both quick and simple.

During verification, a bug was uncovered by OneSpin. Using the provided counterexample[a], the problem was easily identified and fixed.

For formal tools to be effective, properties that cover all possible design behaviour must be written. Helping users verify this, formal verification tools provide *completeness checkers*. These tools are used in verifying that a property suite covers all possible design behaviour. As this property suite has not been checked for completeness, this would make for an interesting student assignment.

[a]A waveform showing an example of a property not holding.

Floating-point units

Verification of the floating-point operator implementations was performed by testbenches applying both random numbers as well as ranges of numbers. After running several hundred million input vectors without error, the behavioural code in the FPU was replaced with these synthesisable implementations. The testbench for the dual core array was then run and gave the same results as before.

Keep in mind that the floating-point units have not been exhaustively verified. By testing 600 million input vectors, only 0.000 000 003 % of the possible inputs have been exercised. However, as the frame was rendered correctly, and proper verification of floating-point units is beyond the scope of this thesis, further verification effort was deemed to be unnecessary.

Code coverage

A coverage report for the dual core array and all its submodules is attached in Appendix E. This has been generated as a result of a simulation rendering the frame in Figure 4.4. 100 % coverage is attained for most coverage types. This indicates that this testbench exercises the design well.

Statement coverage reaches 100 % for all modules but the fixed-point to floating-point converter (fx2fp). This was anticipated prior to enabling coverage, as the scene does not contain all possible FX1.7 numbers. However, this module being generated by a script, is presumed to be functionally correct. This module has low coverage for all coverage types, and this same argument holds for these types.

The reasons for other coverage types not reaching 100 % have been investigated. Low coverage of finite-state machine (FSM) transitions are caused by reset not being asserted while the FSMs are in all possible states. A signal used in ensuring a known state at power up is causing low coverage of some expressions. Finally, the testbench is always ready to accept results, causing low coverage of 'FEC Condition Terms'. Tests of toggling this signal at random occasions have been conducted and found not to produce any problems. This was not done during this simulation, as the results were to be used in performance analysis as well.

For all other coverage types where 100 % coverage is not reached, this is caused by signals used for performance logging. A signal counting the number of active threads (see Section 5.8) is an example of this.

Learning example 5.14: Limitations of code coverage

While code coverage is a great tool in verifying that all design code has been executed, it has its limitations. Firstly, it does not say anything about the functionality of the design. This implies that e.g. missing features will not be identified.

While coverage of states can be checked, the combination of all legal states is not. Using state machines as an example, one can verify that all states and transitions between them are covered. However, one cannot check that the combination of two FSMs cover all legal combinations of states.

These are some of the limitations associated with code coverage. Most limitations are overcome by the use of functional coverage.

5.7 Synthesis

Synthesis results for of various modules and configurations have been collected. In this section, some of these are presented and discussed. For more synthesis results, see Appendix C.

All synthesis results are obtained with the ZedBoard in mind. Xilinx's Vivado 2016.4 has been used in synthesising for the XC7Z020CLG484-1 device with 'Vivado Synthesis Defaults (2016)' and 'Vivado Implementation Defaults (2016)'. Where nothing else is specified, the target frequency has been set to 100 MHz.

Please note that all synthesis results are obtained with the *frame id* being part of the message formats. The purpose of the frame id was to keep utilisation of the ray cores high at all times, but has been removed as its effect was found to be minimal. Further information about the purpose of the frame id is given in Section 5.8.1. It is assumed that synthesis results would not change significantly by removing the frame id. Removal should slightly lower the usage of FFs and LUTs, while it is assumed that the operating frequency would remain unchanged.

5.7.1 Floating-point units

During design and implementation of the floating-point units, some aspects were left for exploration during synthesis. The performance impact of disabling rounding has to be explored. How good the pipeline stage placement performed in Section 5.6.1 is must also be analysed.

Synthesising the dual core with rounding disabled has shown to give an increase in operating frequency of 6.78 % compared to leaving rounding enabled. At the same time, the LUT count went down by 2.15 %. As discussed in Section 5.5.2 the visual quality of a rendered frame is not affected by disabling rounding. Due to this, rounding will be disabled for all further results and discussions.

Synthesis results indicate that the placement of the pipeline registers is close to optimal. I.e. the path delays in and out of the pipeline registers are equally large. Another advantage of the placement is that one of the pipeline registers has been absorbed by a DSP element. This helps improve timing while lowering the overall resource usage.

Resource sharing between the arithmetic logic unit (ALU) and FPU has been considered. It is possible to share the significand multiplier of the floating-point multiplier with the ALU. The significand adder of the floating-point adder could also be shared with the ALU. Experiments with resource sharing have shown that it has a negative impact on the clock frequency. This is due to the extra logic needed to share the resources, and the resources being in the critical path of the design. Also, as seen in Table 5.7, the usage of DSP elements is quite low, and trying to use fewer at the cost of performance is not necessary.

Learning example 5.15: Resource sharing

Resource sharing can be used in order to reduce circuit area and thus maximise performance density. As discussed in this section, a multiplier can be shared by the ALU and FPU. In order to do this, a multiplexer has to be used in selecting either the operands from the ALU or the FPU. This multiplexing adds some area, while also adding to the delay path.

By employing resource sharing, energy efficiency can either go up or down [CB95; RJD98]. Static power consumption will generally go down as the area is decreased. Dynamic power consumption on the other hand, can go either up or down. If a lot more toggling occurs due to the resource sharing, the dynamic and thus total power consumption can increase. The cause of this excess toggling would be low correlation between the operands of the ALU and FPU.

5.7.2 False path elimination

As seen in Figure 5.12, the ALU and FPU operands are selected using multiplexers. When synthesised, a path from the command registers through the ALU's adder to the ALU result pipeline register was reported as a critical path in the design. The command register is only used for the BCEQ instruction, where the output of the adder is not relevant. This indicates that this path is a false path, and that the synthesis tool was unable to recognise this. To circumvent this problem, the multiplexers were instead internalised into the RTL of the ALU and FPU.

Learning example 5.16: False path elimination

In this section, the code that generated a false path was explicitly removed. Another way around the problem would be to use synthesis directives, informing the synthesis tool that the path is indeed a false path. Here, changing the RTL was chosen, as it keeps the code more portable.

5.7.3 Synthesis of dual core array

Different configurations of the dual core array have been synthesised. Figure 5.13 shows how f_{max} goes down as more cores are added to the dual core array. At 20 dual cores, the synthesis tool is no longer able to meet the timing requirements. As seen in Table 5.7, $\sim 96\,\%$ of the LUTs are used when synthesising 20 dual cores. This contention is likely to be the cause of the slight drop in f_{max}. This is also the reason why synthesis of more than 20 dual cores is not possible. The peak in f_{max} for 4 dual cores is deemed to be due to randomisation by the synthesis tool. As seen in Appendix C, peak f_{max} at higher target frequencies is achieved for one dual core.

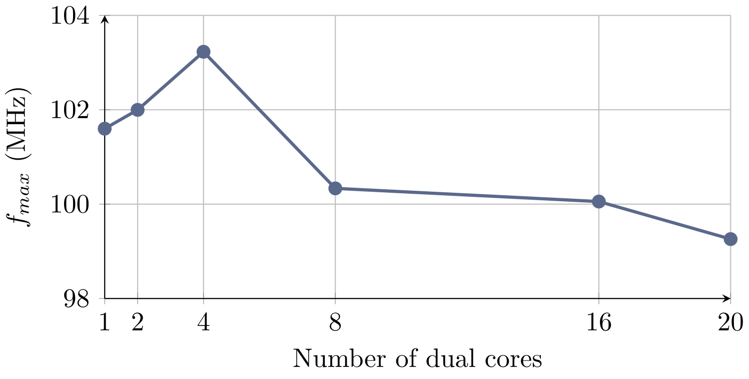


Figure 5.13: Maximum clock frequency for different configurations of the dual core array.

Table 5.7: Resource usage of synthesised dual core array given different number of dual cores (N). The last row shows the total number of available resources of each type.

N	LUT	LUTRAM	FF	BRAM	DSP
1	2574	388	1052	2	6
2	5165	776	2108	4	12
4	10 320	1552	4212	8	24
8	20 548	3104	8428	16	48
16	41 002	6208	16 984	32	96
20	51 098	7760	21 232	40	120
Available	53 200	17 400	106 400	140	220

As seen from Table 5.7, the resource usage scales almost linearly with the number of dual cores. The usage of LUTRAMs, BRAMs and DSPs scales perfectly linearly. Linear regression of the number of LUTs and FFs gives $R^2 \approx 0.999\,99$. The slight increase in usage of FFs per dual core is explained by increased effort to meet timing requirements by the synthesis tool. Despite of this, timing was not met for 20 dual cores.

Analysing the timing report for the case with 20 dual cores, it was found that the critical paths in the design was through the FPUs. Another significant path was the combinatorial ready-path through the feeder and drain elements in the dual core array. Attempts to increase the maximum clock frequency will be discussed in Section 5.7.4.

From Table 5.7, one can see that when synthesising 20 dual cores, $\sim 96\,\%$ of the LUTs on the FPGA are in use. In order to ensure that there is enough space for the system-level blocks discussed in Section 5.1, further analyses will be based upon a configuration with 16 dual cores, occupying $\sim 77\,\%$ of the LUTs on the device.

Synthesising the dual core array with 16 dual cores at different target frequencies gives a peak clock frequency of 102.78 MHz, given $f_{target} = 142.86\,\text{MHz}$. At the same time, the usage of LUTs are $2.31\,\%$ higher than that of $f_{target} = 100.00\,\text{MHz}$ yielding $f_{max} = 100.05\,\text{MHz}$. A plot of the resulting operating frequencies is shown in Figure 5.14. This implies that performance density drops slightly when synthesised at 142.86 MHz, and that synthesis at 100 MHz is a good trade-off.

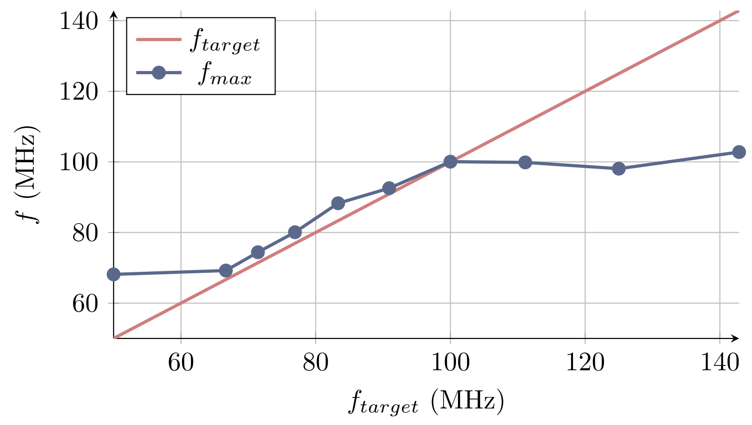


Figure 5.14: Maximum clock frequency for different target frequencies. The array of dual cores was synthesised with 16 dual cores.

5.7.4 Increasing clock frequency

In an attempt to increase ray tracer performance, experiments with increasing operating frequency has been carried out. Inspecting the timing reports, it is found that the paths with the worst negative slack are in the floating-point adders. Manual retiming of the adders was not attempted, as the paths seemed to be reasonably balanced.

Although the paths appeared to be balanced, automatic retiming by the synthesis tool was turned on. Synthesis gave the exact same results as those presented in Figure 5.14. Inspecting the synthesis log, it seemed as though the tool was not able to perform retiming for the critical paths it identified. Furthermore, the paths that after synthesis ended up having the worst negative slack (floating-point adders), were not identified in the retiming step. This indicates that retiming of the FPU would have to be done manually, but with little gain in operating frequency.

In Section 5.2, it is pointed out that by interleaving four threads, one can add another pipeline stage in the ray datapath. Introducing additional pipeline stages in the FPU seems to be the only way to increase the operating frequency. Looking at Figure 5.12, this should not be very complicated; a pipeline stage between S1 and S2 should be relatively easy to fit in. The ALU has no problems with timing, it is left unchanged. Figure 5.15 shows the placement of the new pipeline stage. Here, S2 is the new stage and S3 is the old S2 stage.

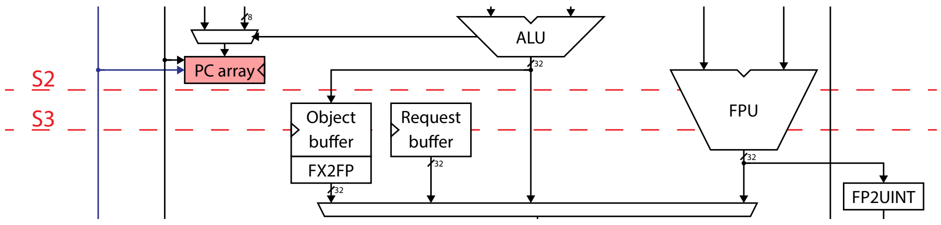


Figure 5.15: Illustration of new pipeline stage placement

In the floating-point multiplier, an additional pipeline register was inserted after the significand multiplier, fully utilising the DSP's internal registers. In the floating-point adder, the timing report was used in order to move the pipeline stages to their optimal position. After synthesising this for a range of target frequencies, it was concluded that the floating-point adder was no longer in the critical path.

As mentioned in learning example 5.1, the combinatorial ready-paths through the chain of feeder and drain elements in the dual core array are quite substantial for configurations with many cores. After insertion of the new pipeline stage in the core datapath, the ready-paths were the new critical paths in the design. These paths were broken by inserting register slices designed to break the ready-path in the middle of the chains.

Synthesising this gave an increase in f_{max} by $\sim 25\,\%$ compared to the results presented earlier. At this point, further increasing the operating frequency requires more effort, and will have to be done at a later stage. Due to limited time, the architecture discussed in this section has not been fully verified. Additionally, no detailed simulation or synthesis results have been collected. Because of this, this architecture will not be used in further discussions.

5.7.5 Summary

The dual core array seems to scale quite well. As seen in Figure 5.13, f_{max} goes down as more cores are added. This seems to be due to contention on the FPGA rather than any real scalability problems of the design. At 16 dual cores, the combinatorial ready-paths through the feeder and drain chains are beginning to become significant. They are not the critical paths, but could very well be the cause of contention. It has been shown that these paths can be broken by inserting reverse register slices in the middle of the chains.

In Section 5.7.4, it was found that introducing another pipeline stage in the ray datapath and FPU gives a performance increase of $\sim 25\,\%$. Even higher performance could be attainable by exploring the other architectural changes outlined in learning example 5.5.

5.8 Performance and scalability

In this section, the performance of the system will be analysed. Important design parameters will be varied to see how well the system scales. In all analyses, only VGA frames are considered. Figure 5.16 illustrates how many instructions that are executed in generating each pixel of the scene in Figure 4.4. This figure will be used in discussions throughout this section.

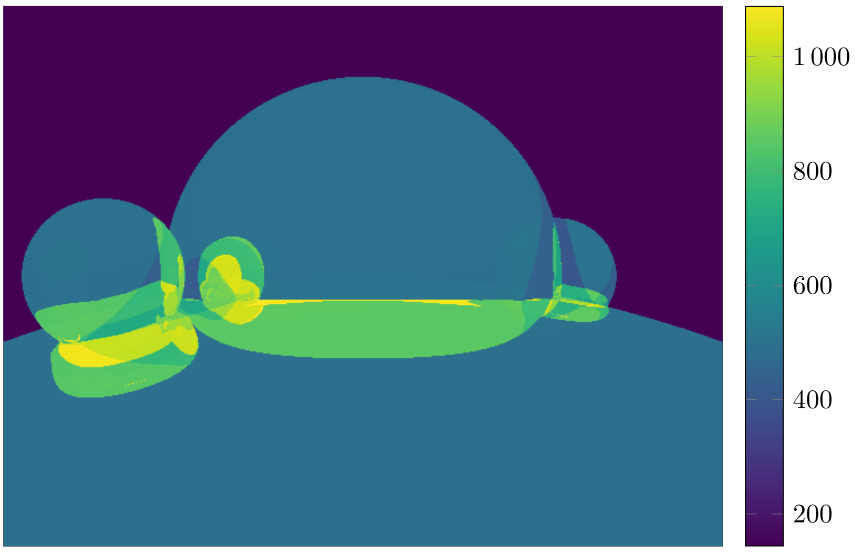


Figure 5.16: Pseudocoloured frame showing how many instructions that are executed in generating each pixel of the scene in Figure 4.4. The data is extracted from the ISA-simulator.

The ray manager is an important component in the system. While it has not been discussed to a great extent in this thesis, its performance is important to analyse. The pseudocode in Algorithm 1 will be used for this. On lines 3 and 4 the scene is read in and the coordinate system is set up. The loop on line 6 loops over all pixels in the image frame, calculating the initial ray direction of each pixel. The initial ray direction is then handed over to the dual core array where a ray core will calculate the colour of that pixel.

Algorithm 1: Ray manager pseudocode

```
1  while True do
2      Wait for request from CPU
3      Read scene information
4      Set up coordinate system
5      Initialise object buffers
6      for all pixels do
7          ray = calculate ray direction
8          pixel colour = raytrace(ray)
9      end
10     Send IRQ to CPU
11 end
```

5.8.1 Thread activity over time

Here, two bottlenecks that were anticipated early in the work with this thesis will be discussed. Firstly, it was assumed that the time spent doing calculations associated with lines 3 to 5 of Algorithm 1 would be a bottleneck in the system. The other anticipated bottleneck was ramping up and down of the number of active threads. Ramping up refers to the number of clock cycles from the first request is issued until full utilisation of all cores is reached. Conversely, ramping down is the number of clock cycles from the last time all threads were active until the last result is sent out of the system.

In order to mask the effect of these bottlenecks, a *frame id* was added. This was added to all requests and responses, as well as stored in the thread state table (Table 5.4). The idea was that every other frame would be assigned frame ids 0 and 1, allowing rendering frames back-to-back. To fully support this, the object buffer and ray tracer architectures would have to be altered as well. Due to the reasons presented below, support for the thread id was never fully implemented.

To analyse how the number of active threads ramp up and down, the RTL has been instrumented with debug signals that report the number of active threads at any time. This is logged by the dual core array testbench and saved to a file after simulation. Analysing this data has helped get insight into the performance bottlenecks of the system.

As seen from Figure 5.17, the ramping is barely noticeable for a VGA frame rendered using 16 dual cores and 2 cycles between each request (this last part will be further discussed in Section 5.8.2).

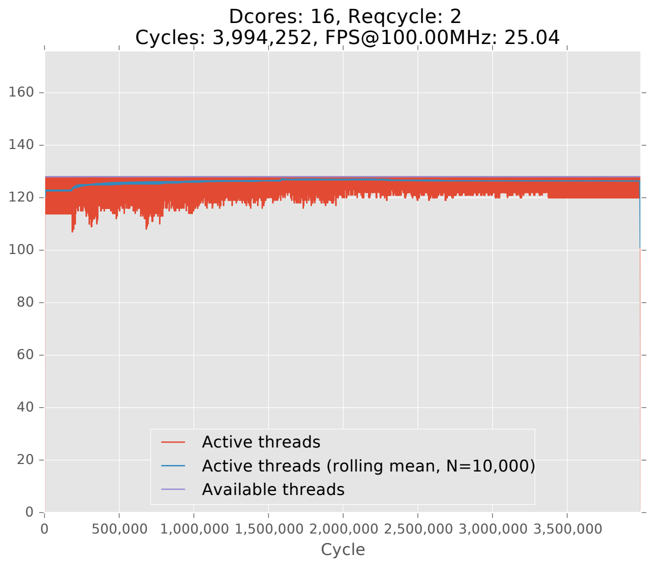


Figure 5.17: Screenshot of a tool used in visualising the number of active threads over time. Here, data for 16 dual cores with 2 cycles between each request is shown.

By zooming in on the plot, it was found that all 128 threads were active after only 360 clock cycles! As seen in Figure 5.18, ramp down is a bit worse, with 3400 cycles from the last time 128 threads were active until the result was accepted by the testbench. Adding these together, this means that the ramping accounts for $\frac{3400+360}{3\,994\,252} \approx 0.094\,\%$ of the time spent ray tracing.

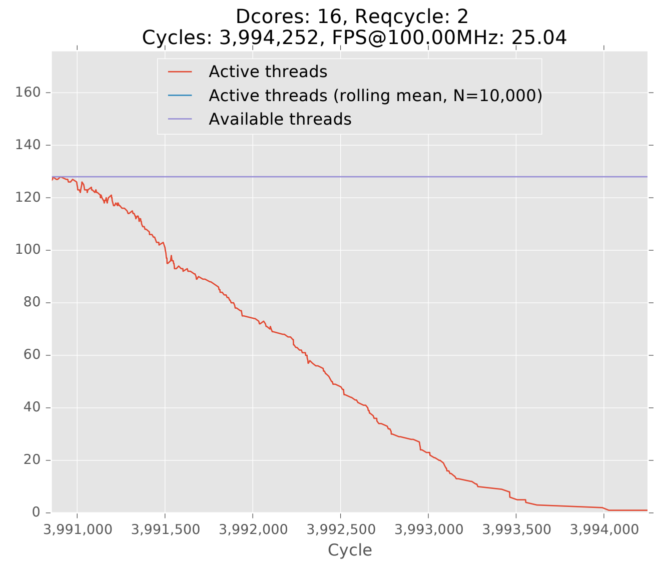


Figure 5.18: Number of active threads over time. Here, data for 16 dual cores with 2 cycles between each request is shown. Only the 3400 last cycles are shown. Rolling mean for this area does not exist when using a centred window of 10 000 cycles.

By looking at lines 3 to 5 of Algorithm 1, one can come up with a rough estimate of the number of clock cycles the ray manager will need to execute them. Greatly overestimating this to 50 000 allows the CPU to spend a few cycles from receiving an IRQ before issuing a new scene to the ray manager. Rendering frames back-to-back, each new frame will be generated every 3 994 252 clock cycles (from Figure 5.17). Adding 50 000 clock cycles to this corresponds to an increase in rendering time by less than 1.3 % compared to back-to-back rendering.

All in all, ramping up and down of the number of requests account for less than 0.1 % of the time spent by the dual core array. Overestimating the time spent by the ray manager and CPU, a performance improvement of less than 1.3 % would be achieved by rendering frames back-to-back. This is negligible, and it is assumed that adding the frame id will negatively impact performance density. Due to all of this, it has been decided that the frame id should not be implemented after all. With proper analysis of this prior to implementation, implementation of the frame id would never have been attempted.

Learning example 5.17: Proper analysis

As seen in this section, proper analysis of a problem should be done before trying to come up with countermeasures. Making an example out of this could help students learn from my mistakes. In a class discussion, the lecturer can present the problem with the number of active threads ramping up and down. The students could then be given some time to discuss the problem in pairs, letting them try to come up with solutions. After this, the data presented in this section could be revealed to students, letting them see that the problem might not have as big of an impact as they (and I) first thought.

5.8.2 Effects of varying ray manager performance

To properly model the performance of the ray tracer, the performance of the ray manager has to be taken into account. As the ray manager has not yet been implemented, an educated guess will have to be made. Here, the pseudocode for the ray manager will be analysed, and its performance estimated. Recall that, as discussed in Section 5.1.1, implementation of the ray manager can be done either in software on the Zynq's CPU or as a specialised hardware unit.

The loop over all pixels in Algorithm 1 has been expanded into Algorithm 2. This shows the actual calculations performed in calculating the initial ray directions, and will be used to analyse how often the ray manager can issue a new request. The calculation of R_p might seem costly in terms of execution time. However, there are a lot of constants involved, and a lot of the code can be moved outside the inner loop. Doing this, R_p can be calculated using only three additions (or one vector addition) in the inner loop. Calculating R_d requires normalising $R_p - C_p$. As C_p is a constant, this subtraction can be moved outside the inner loop. All in all, the calculations needed in the inner loop are one vector addition and one vector normalisation. It is presumed that the calculations that has been moved out of the inner loop will be negligible when amortised over the inner loop.

Considering this, a pipelined hardware unit should be able to issue one request every clock cycle. Anticipating that the hardware unit does not need to be this powerful, the system performance given one request every second clock cycle is

Algorithm 2: Pseudocode of loop in ray manager

```
  Input: Frame size $w \times h$
  Input: Midpoint of screen $P_s$
  Input: Camera position $C_p$
  Input: uvn frame vectors $\mathbf{u}$ and $\mathbf{v}$
  Input: Frame constants $A$, $B$, $C$ and $D$
1 for $j = 0 \to h - 1$ do
2     for $i = 0 \to w - 1$ do
3         $R_p = P_s + (\mathbf{u}((i + 0.5) \times A - B)) + (\mathbf{v}((-j - 0.5) \times C - D))$
4         $\mathbf{R_d} = \text{normalise}(R_p - C_p)$
5         Issue $\mathbf{R_d}$ to dual core array
6     end
7 end
```

analysed instead. This is because running on 100 MHz only requires a new pixel to be generated every 13.6 clock cycles on average (given an FPS of 24).

As the Zynq's CPU has a VFPU [Xil16], it is assumed that it will be able to push out a new request every 5 to 10 clock cycles. If running the CPU at full speed (667 MHz), it will run at 6.67 times the speed of the programmable logic, resulting in a throughput similar to that of a specialised hardware unit. If it is not necessary to run the CPU at full speed, power savings can be achieved by running it at e.g. 100 MHz. Due to this, analyses of the effects of issuing requests every 2, 5 and 10 clock cycles will be performed. From now on, 'reqcycles' refers to the number of clock cycles between each issued request.

In the dual core array testbench, the impact of this on the system performance has been modelled using these numbers. Figure 5.19 shows the relative performance delivered by each individual ray core in the array for different number of dual cores and reqcycles. As the ray manager performance is lowered (i.e. reqcycles goes up), the overall system performance goes down. This indicates that the ray manager is becoming a bottleneck in the system. This performance drop can be explained by looking at Figure 5.20.

As seen in Figure 5.20, the number of active threads is way lower than the number of available threads in the system for a large part of the time. This figure shows data for the case of 32 dual cores with 10 cycles between each request, corresponding to the lower right point in Figure 5.19. This shows that in this configuration, having 32 dual cores is not giving any significant performance improvement, as full utilisation is only achieved for a short period of time (around the 2 000 000 clock cycle tick mark). To get a linear performance increase by

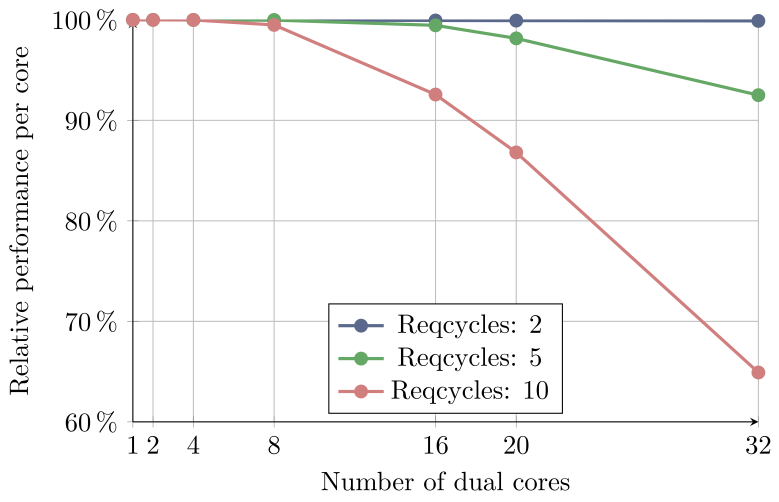


Figure 5.19: Relative performance per core given different ‘reqcycles’. For the numbers used in generating this figure, refer to Appendix C.

adding cores, it is essential that the ray manager can issue requests at the same rate as they are processed.

Explaining the shape of the plot of the number of active threads, one can look at Figure 5.16 and Algorithm 2. Requests are issued in a row-major order, meaning that the section from 0 to 400 000 clock cycles correspond to the darkest area of Figure 5.16. After that, the number of active threads are ramped up because we are covering more and more of first sphere. All other features of Figure 5.16 can be explained in the same way.

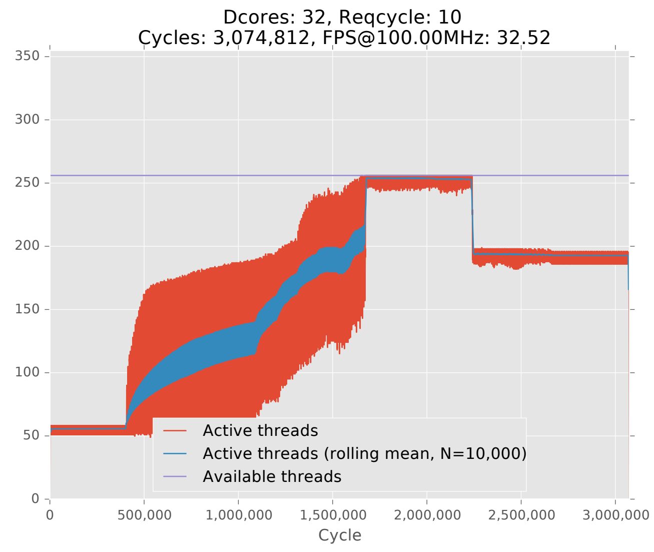


Figure 5.20: Number of active threads over time. Here, data for 32 dual cores with 10 cycles between each request is shown.

5.8.3 System performance

The performance of the system can be estimated using the simulation and synthesis results presented in Figures 5.13 and 5.19. Table 5.8 shows both the estimated FPS and the FPS per dual core.

Table 5.8: FPS per dual core given different number of dual cores (N) and varying reqcycles. The target frequency for the synthesis tool was 100 MHz.

	Reqcycles: 2		**Reqcycles: 5**		**Reqcycles: 10**	
N	**FPS**	**FPS/N**	**FPS**	**FPS/N**	**FPS**	**FPS/N**
1	1.59	1.59	1.59	1.59	1.59	1.59
2	3.19	1.60	3.19	1.60	3.19	1.60
4	6.46	1.62	6.46	1.62	6.46	1.62
8	12.56	1.57	12.56	1.57	12.50	1.56
16	25.05	1.57	24.93	1.56	23.20	1.45
20	31.06	1.55	30.51	1.53	26.98	1.35

The diminishing returns that can be observed are a result of both f_{max} and thread utilisation dropping as more cores are added. The highest FPS per dual core is observed for the configuration with 4 dual cores. This is exclusively due to the peak in f_{max} seen in Figure 5.13. For the case of reqcycles 2 and 5, the performance scales well as cores are added. The small decline in performance per core is mainly due to the clock frequency declining. When new requests are generated every 10th clock cycle, performance per core drops significantly as cores are added. This is due to low utilisation of the available threads, as discussed in Section 5.8.2.

Note that the performance is strongly dependent upon the scene composition. Figure 5.16 shows that the number of instructions executed in tracing rays for the different pixels in the frame vary a lot. The average number of clock cycles used in rendering that specific frame was 415. If one were to zoom in at the yellow areas, making them cover the whole frame, each pixel in the frame would require 1088 cycles. This translates into a drop in FPS by almost two thirds. Conversely, zooming in on parts of the scene without any objects would make the FPS go up. This will be limited by the ray manager performance, as it may become a bottleneck (as seen in Section 5.8.2).

The FPGA is just large enough to allow rendering this particular scene in 24 FPS. For slightly more complex scenes, the FPS will drop below 24. Should the alternative ray datapath discussed in Section 5.7.4 be used, one can assume that the FPS would go up by $\sim 25\,\%$.

Chapter 6

Learning platform viability

The hardware ray tracer can be used in illustrating many key concepts of digital design. This thesis has taken the reader through the iterative design process, starting where the project thesis ‘Design of a Hardware Ray Tracer for digital design education’ [Egg16] left off. During the work on this thesis, some parts designed in the project thesis have been refined in order to optimise performance. These parts of the thesis can be used in illustrating the iterative nature of digital design, where refinements occur continuously.

In Section 5.6, important discussions considering implementation of digital systems were presented. RTL coding, inference patterns for FPGAs, technology independent design and more was covered. Additionally, an assembler for the ISA was implemented. The system was verified using both simulations and formal methods. These are all topics well suited for student assignments. Students can e.g. be instructed to implement a small part of the system and verify that it works according to a set of given requirements.

Performance and scalability analyses are important parts of digital design, and thorough coverage of this was given in Section 5.8. Assignments where students analyse performance and propose performance improving design changes could be made. Here, students should be urged to make back-of-the-envelope calculations and block diagrams.

All in all, this shows that the discussions in Chapter 5 cover large parts of the design process. This thesis has presented results of design choices as the choices have been made. These results have been analysed, and further decisions based upon them. This kind of fast feedback is the essence of the iterative design process. Together with the anecdotal learning examples, the work presented in this thesis can be adopted as assignments, project work, student discussions, lectures and more. While the system has been implemented using SystemVerilog, adapting the material for a course on VHDL should not require much effort.

6.1 Register file example

An example of a module that can be examined during a course is the register file. High level requirements can be broken down to engineering requirements in a class discussion. Important themes to cover include the choice of implementation using BRAM or LUTRAM. In this discussion, students should consider the effects of adding multiply-accumulate (MAC) and/or multiply-add (MADD) instructions.[1] Low power design techniques like clock gating should also be considered as part of the discussions.

Later, students can implement their proposed architecture in RTL. To do this, they should study inference patterns in [Xil13]. Depending on the course focus, different verification methods can be used. Students could be provided with a simple testbench, and told to examine the code coverage the testbench gives. Finding that it does not give 100 % coverage, they will have to extend the testbench with more test cases. This example can easily be extended to functional coverage and even formal verification.

[1] This will require three read ports, and thus three BRAMs if implemented using BRAM.

Chapter 7

Conclusion

In this master's thesis, large parts of a ray tracing system have been implemented in RTL. The system is capable of rendering a simple VGA frame at ~ 25 FPS in a 32-core configuration. This configuration occupies $\sim 77\,\%$ of the LUTs on the target FPGA.

To optimise the performance of the system, the specification from the project thesis [Egg16] has been improved. ISA extensions have resulted in significant performance increases. In addition, an updated system-level block diagram, interconnect design, thread interleaving method and message protocols have been specified. Floating-point operators for addition and multiplication have been modelled and implemented.

RTL for the dual core array has been implemented. An assembler for the ISA was developed and used in generating machine code for the ray datapath. Simulations have shown that the implementation generates the exact same results as the ISA-simulator built in [Egg16]. It has been demonstrated that the RTL is synthesisable for the target FPGA. The RTL has been verified using both simulation and formal methods.

The discussions in this thesis can be adapted for a course on digital design. The learning examples scattered around in the thesis can be used as starting points for interesting student discussions and assignments.

Most of the system-level requirements have been met. The requirements that have not been addressed are part of the proposed future work.

7.1 Future work

While simulation and synthesis of the dual core array have shown promising results, there is still room for architectural improvements. Suggested improvements:

- Verify the architecture with 25 % performance increase (see Section 5.7.4) and use this. Also check the resulting performance density for all cases listed in learning example 5.5.
- Consider adding either *MAC* and/or *MADD* to the ISA. This will require more logic in the ray cores, but will also increase the performance per core. Performance density should be considered here.
- Further optimisations of the ISA. E.g. the suggested *decrement and branch* instruction.
- Adding a hardware unit performing $\sqrt{x}$ and $\frac{1}{\sqrt{x}}$. This will remove the effects of the software approximations introduced in [Egg16]. As such a unit is likely to have a rather high latency, implementation will require alterations to the multithreading scheduler. Implemented cleverly, overall system performance might increase. If this unit becomes large, sharing it between several ray cores might be needed to keep performance density high.
- Proper verification of floating-point operators.
- Compare floating-point performance and area of this thesis' implementations to other implementations. E.g. 'Xilinx Floating-Point Operator IP', Altera's hardened floating-point units [Vis16] and the open source implementations available from OpenCores.
- REQ_PERF_002 states that 'The utilisation of all functional units should be as high as possible and preferably above 50 % while the ray tracing algorithm is running'. While the ISA has been designed for high utilisation of its functional units, no analysis of this has been performed.

To get a working system with an FPGA rendering frames in real-time, some work must be done at the system-level:

- Design of a ray manager with all that encompasses:
  - It is a relatively low-throughput module, and several implementations should be considered. It could be implemented as a specialised hardware unit, in the ARM CPU on the Zynq SoC or using a soft microprocessor like Altera's Nios or Xilinx's MicroBlaze.

  - Determining how writing to the frame buffer should be implemented. This could be done by 1 AXI transaction per pixel, or one could buffer up several pixels before sending them as a chunk. Sending one by one is not efficient in terms of energy consumption or bus utilisation. However, complexity and area should be lower than that of the buffering solution.

- Implementing the system on the ZedBoard. This includes creating a display controller and interfacing with the processing system (see Figure 5.1).

Bibliography

[ARM13] *AMBA AXI and ACE Protocol Specification.* Issue E. ARM. Feb. 2013.

[Bru09] Erik Brunvand. *Floating Point Circuits.* Lecture slides from CS5830 at The University of Utah. 2009.

[CB95] Anantha P. Chandrakasan and Robert W. Brodersen. *Low Power Digital CMOS Design.* Norwell, MA, USA: Kluwer Academic Publishers, 1995. ISBN: 079239576X.

[Egg16] Jonas Agentoft Eggen. 'Design of a Hardware Ray Tracer for digital design education'. Project Thesis. Dec. 2016.

[EL04] Miloš D. Ercegovac and Tomás Lang. *Digital Arithmetic.* The Morgan Kaufmann Series in Computer Architecture and Design Series. Morgan Kaufmann, 2004. ISBN: 9781558607989.

[Fie16] Edvard Fielding. *Binary Arithmetic Unit Design.* Lecture slides from TFE4141. 2016.

[Gje17] Øystein Gjermundnes. *Personal written and oral communication.* 2017.

[HP12] John Hennessy and David Patterson. *Computer Architecture : A Quantitative Approach.* Morgan Kaufmann, 2012. ISBN: 978-0123838728.

[Hen08] Henrik. *File:Ray trace diagram.svg.* Apr. 2008. URL: `https://en.wikipedia.org/wiki/File:Ray_trace_diagram.svg` (visited on 06/12/2016).

[IEEE08] 'IEEE Standard for Floating-Point Arithmetic'. In: *IEEE Std 754-2008* (Aug. 2008), pp. 1–70. DOI: `10.1109/IEEESTD.2008.4610935`.

[LGH94] James Laudon, Anoop Gupta and Mark Horowitz. 'Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations'. In: *SIGOPS Oper. Syst. Rev.* 28.5 (Nov. 1994), pp. 308–318. ISSN: 0163-5980. DOI: `10.1145/381792.195576`. URL: `http://doi.acm.org/10.1145/381792.195576`.

[MG97] G. De Michell and R. K. Gupta. 'Hardware/software co-design'. In: *Proceedings of the IEEE* 85.3 (Mar. 1997), pp. 349–365. ISSN: 0018-9219. DOI: 10.1109/5.558708.

[Mul+10] Jean-Michel Muller et al. *Handbook of Floating-Point Arithmetic.* ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9. Birkhäuser Boston, 2010.

[PD10] Sudeep Pasricha and Nikil Dutt. *On-Chip Communication Architectures: System on Chip Interconnect (Systems on Silicon).* Morgan Kaufmann, 2010.

[PH14] David Patterson and John Hennessy. *Computer Organization and Design : The Hardware/Software Interface.* Morgan Kaufmann, 2014. ISBN: 978-0124077263.

[Pho75] Bui Tuong Phong. 'Illumination for Computer Generated Pictures'. In: *Commun. ACM* 18.6 (June 1975), pp. 311–317. ISSN: 0001-0782. DOI: 10.1145/360825.360839. URL: http://doi.acm.org/10.1145/360825.360839.

[RJD98] Anand Raghunathan, Niraj K. Jha and Sujit Dey. *High-Level Power Analysis and Optimization.* Norwell, MA, USA: Kluwer Academic Publishers, 1998. ISBN: 0792380738.

[Suf07] Kevin Suffern. *Ray Tracing from the Ground Up.* A K Peters/CRC Press, 2007.

[The+08] Theoharis Theoharis et al. *Graphics and visualization: principles & algorithms.* CRC Press, 2008.

[VB08] James M. Van Verth and Lars M. Bishop. *Essential Mathematics for Games and Interactive Applications, Second Edition: A Programmer's Guide.* 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 9780123742971.

[Vis16] Amulya Vishwanath. *White Paper: Enabling High-Performance Floating-Point Designs.* Tech. rep. Intel, 2016.

[Whi80] Turner Whitted. 'An Improved Illumination Model for Shaded Display'. In: *Commun. ACM* 23.6 (June 1980), pp. 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882. URL: http://doi.acm.org/10.1145/358876.358882.

[Xil13] *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices.* Version 14.5. Xilinx, Inc. Mar. 2013.

[Avn14] *ZedBoard Hardware User's Guide.* Version 2.2. Avnet, Inc. Jan. 2014.

[Xil16] *Zynq-7000 All Programmable SoC Overview.* Version 1.10. Xilinx, Inc. Sept. 2016.

Appendix A

Architecture specification

Here, the architecture specification from [Egg16] is reproduced. Some alterations to the text has been made to make it independent of the whole project thesis document.

A.1 Data structures

Data structures used to hold the scene, objects and materials are introduced. All numbers encoded using the FX1.7 fixed-point format have a legal range of 0.0 to 1.0.

🟦	32 bit floating-point.
🟥	Unsigned integer.
🟩	Fixed-point. FX1.7
⬜	Reserved bits. Set to 0.

Figure A.1: Data structures legend

Figure A.2 shows the scene descriptor data structure. It specifies the position of the light, camera and the point that the camera is looking towards. `cam_-fov` defines the camera's horizontal view angle in degrees. The frame size is given in W10. The last four fields identify the memory location and size of the sphere array and material array. These refer to arrays of Figures A.3 and A.4, respectively.

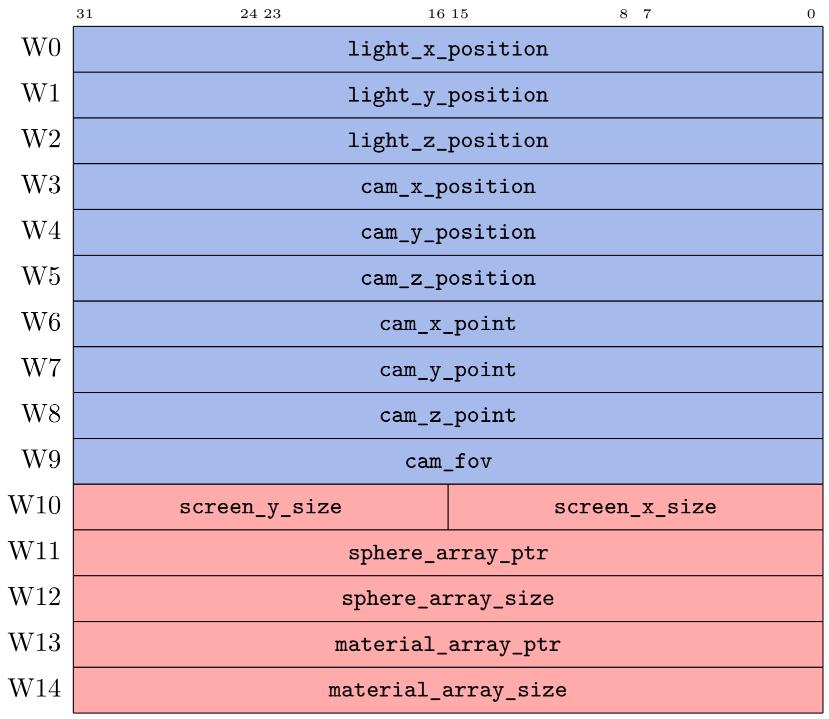


Figure A.2: Scene descriptor

Figure A.3 describes a sphere. The sphere is specified by its position, squared radius and reciprocal radius.

The materials are described by Figure A.4. `ar` is the colour of the ambient reflection, `ad` the diffuse colour and `as` the specular colour. `k_refl` is the reflection coefficient. All of these parameters are used in the Phong reflection model [Pho75].

The object buffers hold copies of some of the data in the scene, all objects and all materials. The memory layout of the object buffer is shown in Figure A.5.

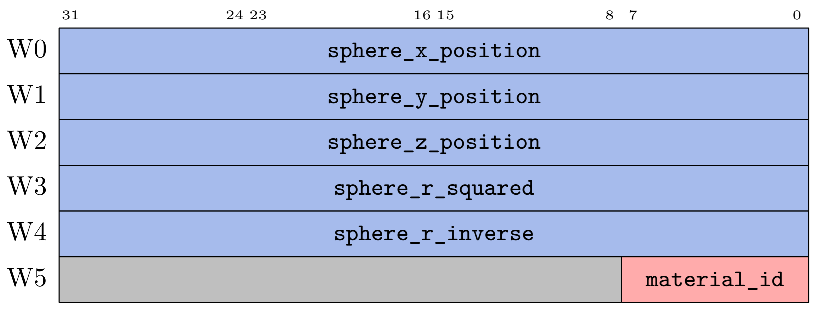


Figure A.3: Sphere descriptor

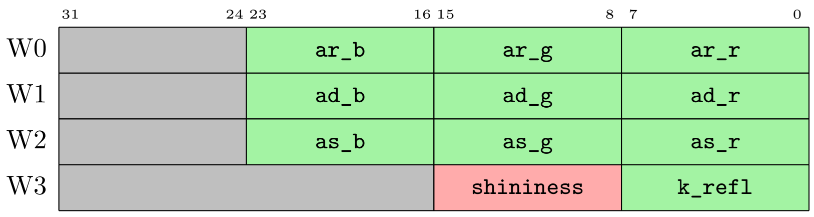


Figure A.4: Material descriptor

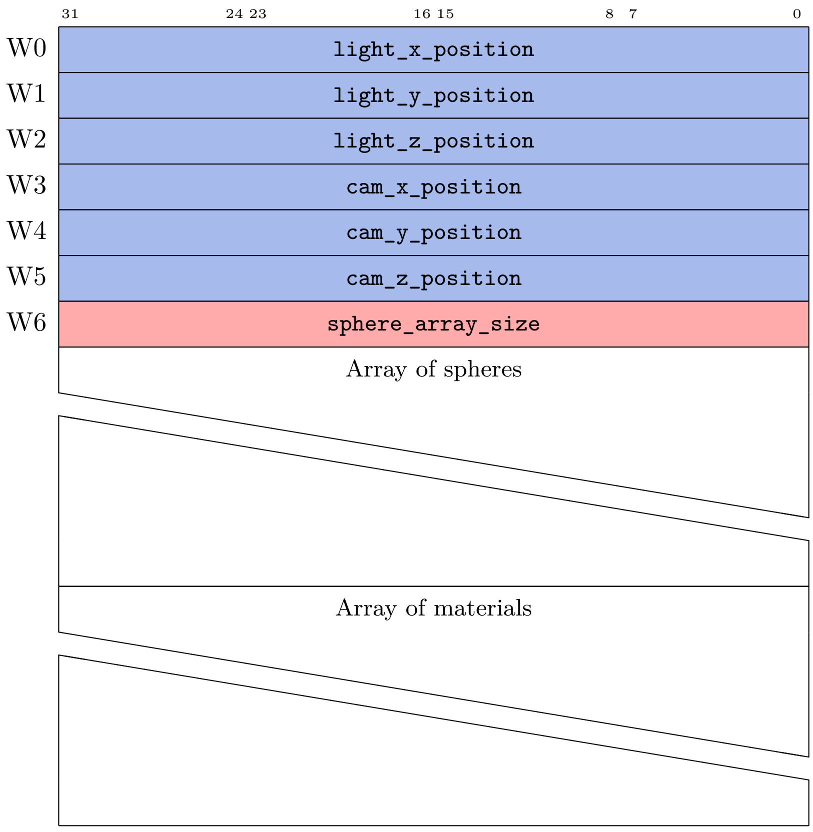


Figure A.5: Object buffer memory layout

A.2 Message formats

Having presented the data structures, it is time to discuss the message formats used in communicating between the ray manager and ray cores. Figure A.6 shows the request sent from the ray manager to a core. This request holds the `x` and `y` coordinate of the pixel being rendered. This works as an ID, and is also included in the response from the core to the ray manager. Given VGA-resolution, 640×480 pixels, `x` needs to be $\lceil \log_2 640 \rceil$ bit $= 10$ bit wide. `y` is $\lceil \log_2 480 \rceil$ bit $= 9$ bit wide. The `direction` is a three-element vector of single-precision floating-point numbers.

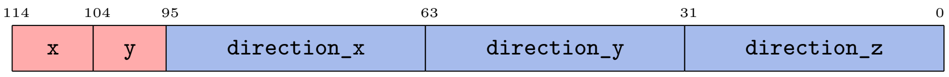


Figure A.6: Ray request

After finishing the ray tracing of the request, the core sends a response containing the calculated colour of the pixel. This is represented as single-precision floating-point numbers, as seen in Figure A.7. In this master's thesis, a different ray response format seen in Figure 5.9 is used in place of this.

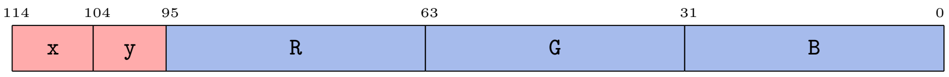


Figure A.7: Ray response

A.3 Ray manager control/status registers

The ray tracer is to be part of a bigger system, and will be controlled by for example a CPU. The ray manager needs to know where the scene is located, and when it should start processing it. A way for the CPU to query the status of the ray tracer is also in place. All registers are accessed over APB.

Figure A.8 shows the layout of the command register, along with the legal values of the `command` field. This register can both be read and written. The names of the commands are self-explanatory. The unused encodings are reserved for future use. The address of this register is `0x00000000`.

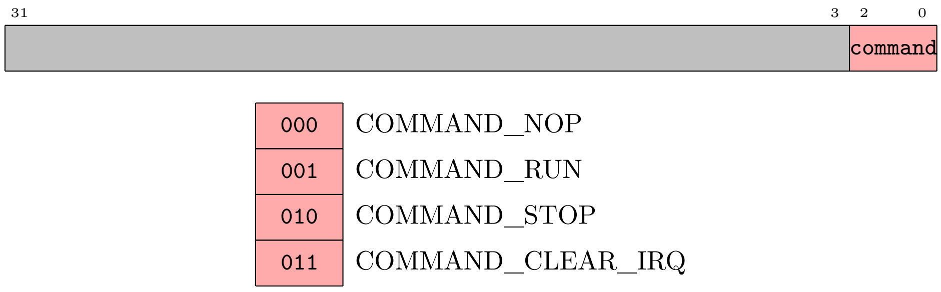


Figure A.8: Command register

Figure A.9 shows the layout of the status register, along with the legal values of the `status` field. This register is read only. The names of the statuses are self-explanatory. The `progress` field indicates the rendering progress as an 8 bit integer. The unused encodings are reserved for future use. The address of this register is `0x00000004`.

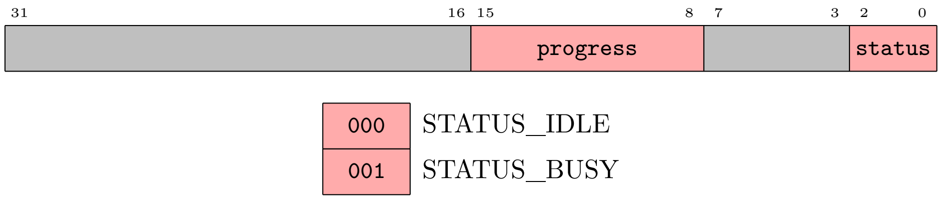


Figure A.9: Status register

Figure A.10 shows the layout of the scene address register. This register can both be read and written to. The `address` field holds the physical address of the scene descriptor. The address of this register is `0x00000008`.

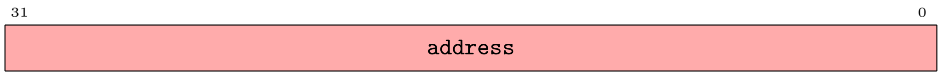


Figure A.10: Scene address register

Appendix B

Assembly code

The numbers used in making Figure 5.10 are shown in Table B.1. Listing B.1 shows the final version of the assembly program.

Table B.1: Number of instructions used in rendering a VGA frame given different revisions of the code and instruction set. The data is extracted from the ISA-simulator, and does not model congestion nor the ray core.

Code/ISA revision	**Instructions**	**Cum. diff**	**% change**
Project thesis	138 245 949		
Communication added	141 317 949	3 072 000	2.22
LFX added	133 398 747	−4 847 202	−3.51
Reuse of variables in code	129 233 505	−9 012 444	−6.52
ADDUI, FMULI and FADDI added	127 429 186	−10 816 763	−7.82

Listing B.1: Assembly program from [Egg16] with alterations discussed in Section 5.4.1.

```
waitreq:
bceq 0, waitreq                  ; wait for command to change
 ↪  from 'none'
pop  $dir_z
pop  $dir_y
pop  $dir_x
stat 1                           ; set status to 'busy'
```

```

lw   $orig_x, 12[$0]            ; Load cam position from object buffer
lw   $orig_y, 16[$0]
lw   $orig_z, 20[$0]
add  $color_r, $0, $0           ; Initialise color registers (FP zero and UINT zero is the same)
add  $color_g, $0, $0
add  $color_b, $0, $0
lui  $k_refl, 0x3f80            ; Upper bits of 1.0 in FP32. Lower bits are zero
lui  $rec_surf, 0xffff          ; Should never have this many objects in scene..

addi $rec_left, $0, 2           ; Hard coded recursion depth! Could be param from scene.

raytraceloop:
addi $isect_surface, $0, 0xff   ; Initial isect_surface
lui  $18, 0x7f00                ; min_t = 1.7014118e38

lw   $23, 24[$0]                ; num_spheres (p)

main_dec_p:
subi $23, $23, 1
beq  $23, $rec_surf, main_dec_p
bneg $23, sphereloopdone

muli $24, $23, 24

lw   $25, 28[$24]               ; center_x
lw   $26, 32[$24]               ; center_y
lw   $27, 36[$24]               ; center_z
lw   $24, 40[$24]               ; radius_squared

fsub $25, $25, $orig_x          ; dP_x
fsub $26, $26, $orig_y          ; dP_y
fsub $27, $27, $orig_z          ; dP_z
fmul $28, $dir_x, $25           ; udP x factor
fmul $29, $dir_y, $26
```

```
fadd $28, $28, $29              ; udP x,y factor
fmul $29, $dir_z, $27
fadd $28, $28, $29              ; udP all factors

fmul $29, $dir_x, $28           ; u_x * udP
fsub $29, $25, $29              ; A_x
fmul $29, $29, $29              ; A.A x factor

fmul $30, $dir_y, $28           ; u_y * udP
fsub $30, $26, $30              ; A_y
fmul $30, $30, $30              ; A.A y factor
fadd $29, $29, $30              ; A.A x,y factor

fmul $30, $dir_z, $28           ; u_z * udP
fsub $30, $27, $30              ; A_z
fmul $30, $30, $30              ; A.A z factor
fadd $29, $29, $30              ; A.A all factors

fsub $24, $24, $29              ; D

bneg $24, main_dec_p            ; if (D < 0): continue

; Used registers from here:
; udP: $28, D: $24, min_t: $18

; start sqrtapprox_a
srl  $24, $24                   ; as_int(D) >> 1
addui $24, $24, 0x1fc0          ; sqrtD : $24 (0x1fc0 =
 ↪  as_int(1.0f) >> 1)
; end sqrtapprox_a

fsub $19, $28, $24              ; t = udP - sqrtD

bpos $19, tchosen
fadd $19, $28, $24              ; (t = udP + sqrtD) if t < 0

tchosen:
bneg $19, main_dec_p            ; if (t < 0): continue

fsub $20, $19, $18              ; t - min_t
```

```
bpos $20, main_dec_p            ; if (t > min_t): continue

add  $18, $19, $0               ; min_t = t
add  $isect_surface, $23, $0    ; isect_surface = p

goto main_dec_p



spherelooopdone:

addi $19, $0, 0xff              ; Initial isect_surface
beq  $isect_surface, $19, welldone

fmul $19, $dir_x, $18           ; dir_x * min_t
fadd $orig_x, $orig_x, $19      ; orig_x += dir_x * min_t
fmul $19, $dir_y, $18           ; dir_y * min_t
fadd $orig_y, $orig_y, $19      ; orig_y += dir_y * min_t
fmul $19, $dir_z, $18           ; dir_z * min_t
fadd $orig_z, $orig_z, $19      ; orig_z += dir_z * min_t

muli $18, $isect_surface, 24
lw   $19, 32[$18]               ; center_y
lw   $20, 36[$18]               ; center_z
lw   $21, 44[$18]               ; radius_inverse
lw   $18, 28[$18]               ; center_x

fsub $isect_normal_x, $orig_x, $18
fmul $isect_normal_x, $isect_normal_x, $21

fsub $isect_normal_y, $orig_y, $19
fmul $isect_normal_y, $isect_normal_y, $21

fsub $isect_normal_z, $orig_z, $20
fmul $isect_normal_z, $isect_normal_z, $21



; calculate local color
lw   $20, 0[$0]             ; Light posx
lw   $21, 4[$0]
lw   $22, 8[$0]
fsub $20, $20, $orig_x       ; shadow ray direction
fsub $21, $21, $orig_y
fsub $22, $22, $orig_z
fmul $23, $20, $20          ; Dot product before norm.
fmul $24, $21, $21
fadd $23, $23, $24
fmul $24, $22, $22
fadd $23, $23, $24
srl  $24, $23               ; Invsqrtapprox_a
lui  $25, 0x5f37            ; magic constant
addi $25, $25, 0x59df       ; magic constant
sub  $24, $25, $24          ; ~Invsqrtapprox_a
fmuli $23, $23, 0xbf00      ; Invsqrtapprox_b ; x * (-0.5f)
fmul $23, $23, $24          ; * y
fmul $23, $23, $24          ; * y
faddi $25, $23, 0x3fc0      ; 1.5f + ((-0.5)*x * y*y)
fmul $23, $24, $25          ; ~Invsqrtapprox_b
fmul $20, $20, $23          ; vector norm.
fmul $21, $21, $23          ; vector norm.
fmul $22, $22, $23          ; vector norm.

lw   $23, 24[$0]            ; num_spheres

muli $23, $23, 24

muli $24, $isect_surface, 24
lw   $24, 48[$24]           ; material_id
muli $24, $24, 16

add  $31, $23, $24          ; material ptr

lfx  $24, 28[$31]            ; AR
lfx  $25, 29[$31]            ; AG
lfx  $23, 30[$31]            ; AB

fmul $24, $24, $k_refl      ; Doing this multiplication here in
↪  order to save a few registers.
fmul $25, $25, $k_refl
fmul $23, $23, $k_refl
fadd $color_r, $color_r, $24
fadd $color_g, $color_g, $25
fadd $color_b, $color_b, $23

lw   $23, 24[$0]             ; num_spheres (p)

dec_p:
subi $23, $23, 1
beq  $23, $isect_surface, dec_p
bneg $23, loopdone

muli $24, $23, 24

lw   $25, 28[$24]            ; center_x
lw   $26, 32[$24]            ; center_y
lw   $27, 36[$24]            ; center_z
lw   $24, 40[$24]            ; radius_squared

fsub $25, $25, $orig_x        ; dP_x
fsub $26, $26, $orig_y        ; dP_y
fsub $27, $27, $orig_z        ; dP_z
fmul $28, $20, $25           ; udP x factor
fmul $29, $21, $26
fadd $28, $28, $29           ; udP x,y factor
fmul $29, $22, $27
fadd $28, $28, $29           ; udP all factors

fmul $29, $20, $28           ; u_x * udP
fsub $29, $25, $29           ; A_x
fmul $29, $29, $29           ; A.A x factor

fmul $30, $21, $28           ; u_y * udP
fsub $30, $26, $30           ; A_y
fmul $30, $30, $30           ; A.A y factor
fadd $29, $29, $30           ; A.A x,y factor

fmul $30, $22, $28           ; u_z * udP
fsub $30, $27, $30           ; A_z
fmul $30, $30, $30           ; A.A z factor
```

```
fadd $29, $29, $30          ; A.A all factors

fsub $24, $24, $29          ; D

bneg $24, dec_p             ; if (D < 0): continue

bpos $28, done              ; if (udP > 0): return local_color

goto dec_p

loopdone:
; Reuseable registers from here: $23 and up!

fmul $24, $20, $isect_normal_x      ; N dot L x factor
fmul $25, $21, $isect_normal_y      ; N dot L y factor
fadd $24, $24, $25
fmul $25, $22, $isect_normal_z      ; N dot L z factor
fadd $29, $24, $25                  ; N dot L

bneg $29, done

lfx  $25, 32[$31]            ; AD
lfx  $26, 33[$31]            ; AD
lfx  $27, 34[$31]            ; AD

fmul $24, $29, $k_refl

fmul $25, $25, $24
fmul $26, $26, $24
fmul $27, $27, $24

fadd $color_r, $color_r, $25
fadd $color_g, $color_g, $26
fadd $color_b, $color_b, $27


fsub $20, $20, $dir_x
fsub $21, $21, $dir_y
fsub $22, $22, $dir_z

```

```
fmul $23, $20, $20          ; Dot product before norm.
fmul $24, $21, $21
fadd $23, $23, $24
fmul $24, $22, $22
fadd $23, $23, $24
srl  $24, $23               ; Invsqrtapprox_a
lui  $25, 0x5f37            ; magic constant
addi $25, $25, 0x59df       ; magic constant
sub  $24, $25, $24          ; ~Invsqrtapprox_a
fmuli $23, $23, 0xbf00      ; Invsqrtapprox_b ; x * (-0.5f)
fmul $23, $23, $24          ; * y
fmul $23, $23, $24          ; * y
faddi $25, $23, 0x3fc0      ; 1.5f + ((-0.5)*x * y*y)
fmul $23, $24, $25          ; ~Invsqrtapprox_b
fmul $20, $20, $23          ; vector norm.
fmul $21, $21, $23          ; vector norm.
fmul $22, $22, $23          ; vector norm.

fmul $24, $20, $isect_normal_x      ; N dot H x factor
fmul $25, $21, $isect_normal_y      ; N dot H y factor
fadd $24, $24, $25
fmul $25, $22, $isect_normal_z      ; N dot H z factor
fadd $29, $24, $25                  ; N dot H

; CODE FOR REPSQUARING
lb   $25, 41[$31]           ; shininess
subi $25, $25, 1
beq  $25, $0, donesquaring

repsquare:
fmul $29, $29, $29
subi $25, $25, 1
bpos $25, repsquare

donesquaring:

lfx  $25, 36[$31]            ; AD as fx1.7
lfx  $26, 37[$31]            ; AD as fx1.7
lfx  $27, 38[$31]            ; AD as fx1.7
```

```
fmul $24, $29, $k_refl

fmul $25, $25, $24
fmul $26, $26, $24
fmul $27, $27, $24

fadd $color_r, $color_r, $25
fadd $color_g, $color_g, $26
fadd $color_b, $color_b, $27


done: ; ~calculate_local_color


lfx  $18, 40[$31]                ; k_refl as fx1.7

fmul $k_refl, $k_refl, $18

; Not doing anything with our scene. .4*.4*.4 > .05
faddi $18, $k_refl, 0xbd4e       ; -0.05029297
bneg $18, welldone               ; if (k_refl < 0.05): break


add  $rec_surf, $isect_surface, $0

fmul $18, $dir_x, $isect_normal_x
fmul $19, $dir_y, $isect_normal_y
fadd $18, $18, $19
fmul $19, $dir_z, $isect_normal_z
fadd $18, $18, $19               ; Ri_dot_N

fmuli $18, $18, 0x4000           ; 2*Ri_dot_N. Could be
 ↪  implemented by fadd $18, $18, $18

fmul $19, $isect_normal_x, $18
fsub $dir_x, $dir_x, $19
fmul $19, $isect_normal_y, $18
fsub $dir_y, $dir_y, $19
fmul $19, $isect_normal_z, $18
fsub $dir_z, $dir_z, $19
```

```



subi $rec_left, $rec_left, 1
bpos $rec_left, raytraceloop

welldone:

stat 2                              ; request access to resbuf

resbuf_w:
bceq 0 resbuf_w                     ; waiting for 'resbuf_gnt'

push $color_r
push $color_g
push $color_b

stat 0
goto waitreq
```

Appendix C

Simulation and synthesis results

This chapter presents all simulation and synthesis results, whereas Chapter 5 only presented parts of these.

C.1 Simulation results

The number of clock cycles executed in rendering the frame is shown in Table C.1. Plots of the number of active threads over time for all configurations in this table is shown below.

Table C.1: Number of clock cycles needed to render one VGA frame.

N	**Reqcycles: 2**	**Reqcycles: 5**	**Reqcycles: 10**
1	63 869 699	63 869 690	63 869 665
2	31 935 484	31 935 728	31 935 985
4	15 968 742	15 968 870	15 969 051
8	7 985 524	7 985 501	8 024 148
16	3 994 252	4 013 757	4 312 303
20	3 195 919	3 253 425	3 678 770
32	1 997 994	2 157 575	3 074 812

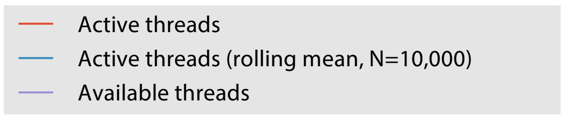
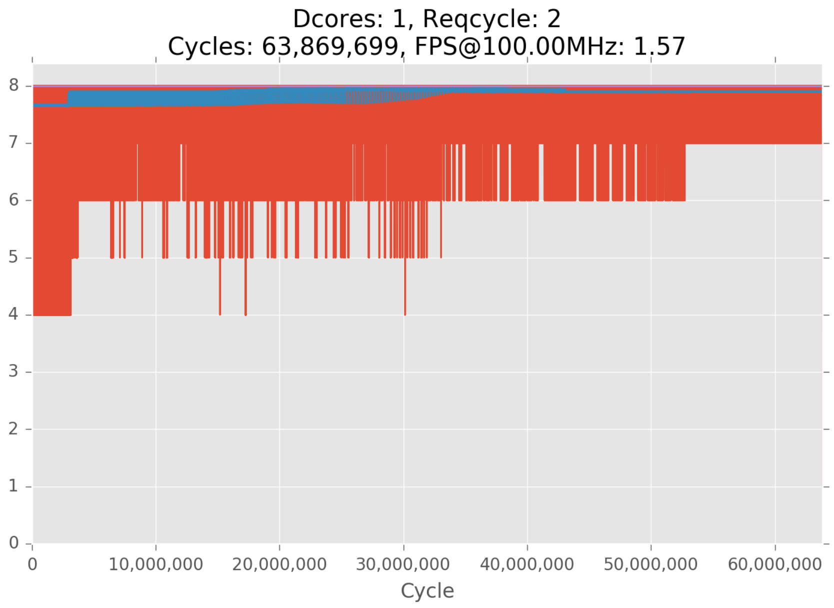
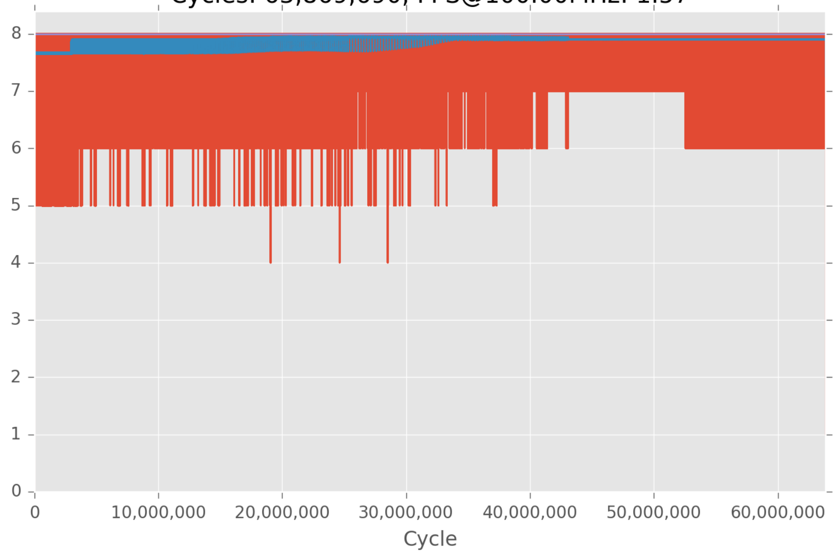


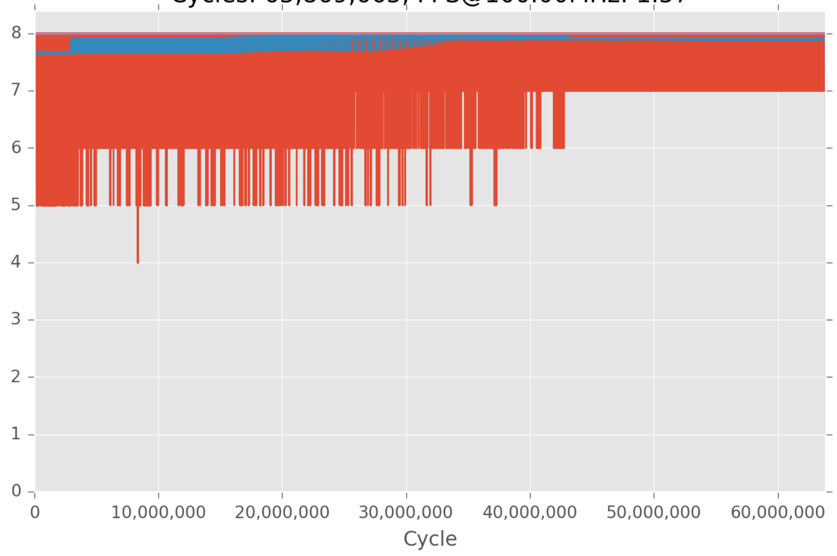
Figure C.1: Legend for all plots of active threads vs. time



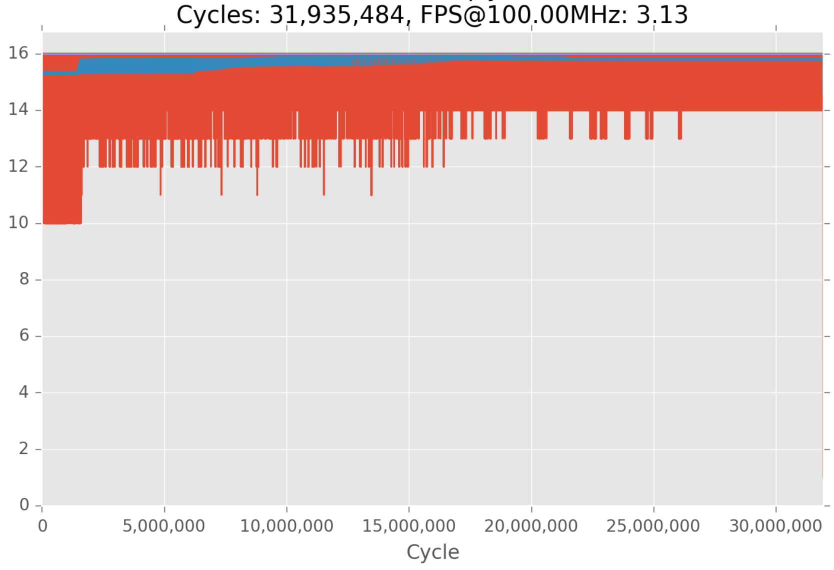
Dcores: 1, Reqcycle: 5
Cycles: 63,869,690, FPS@100.00MHz: 1.57



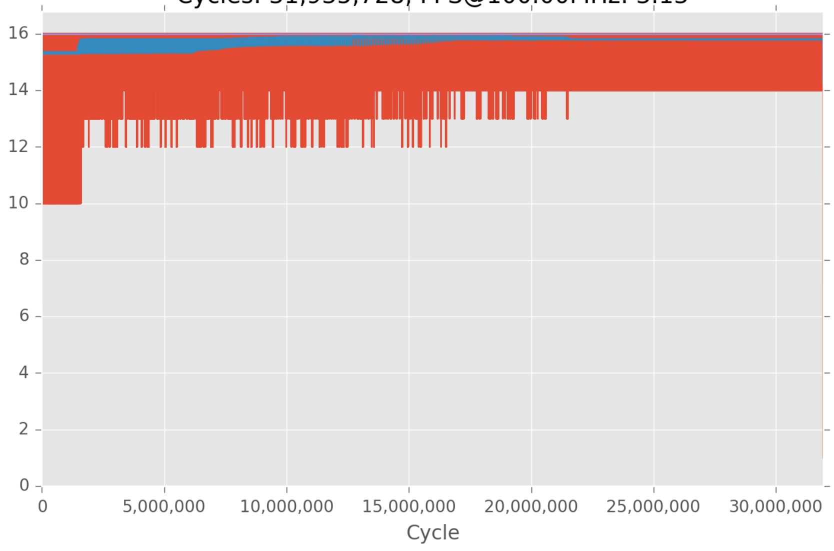
Dcores: 1, Reqcycle: 10
Cycles: 63,869,665, FPS@100.00MHz: 1.57

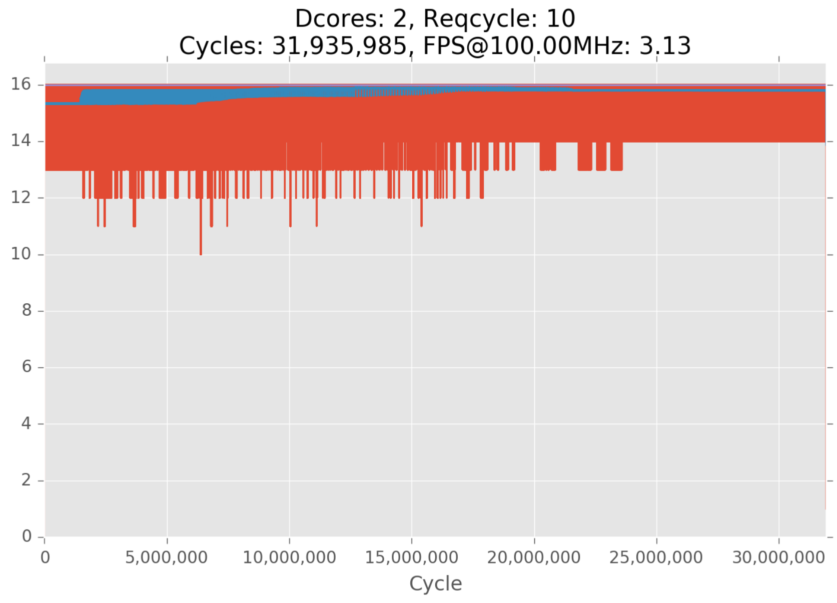


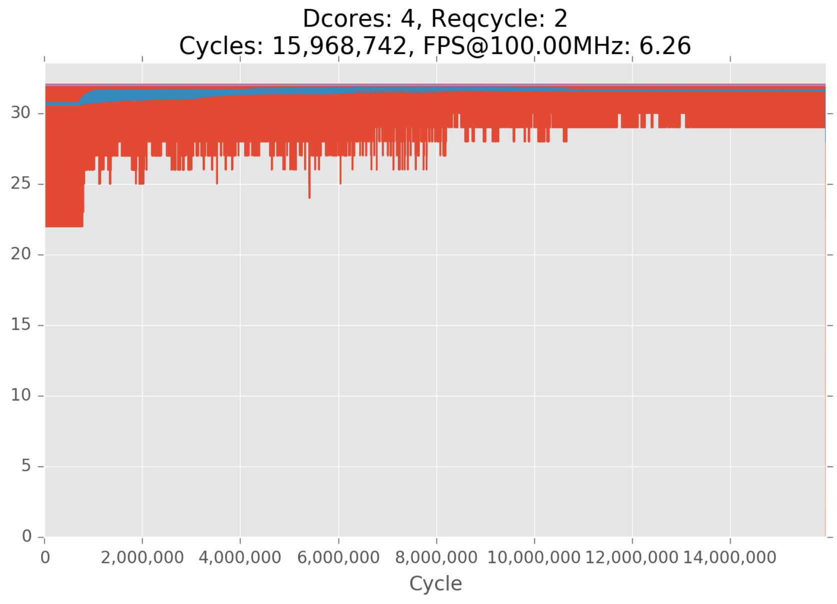
Dcores: 2, Reqcycle: 2
Cycles: 31,935,484, FPS@100.00MHz: 3.13

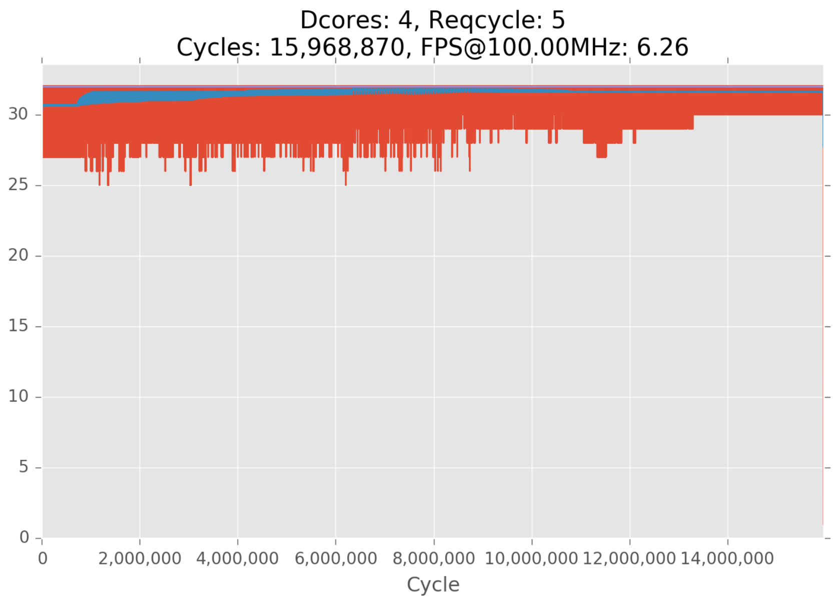


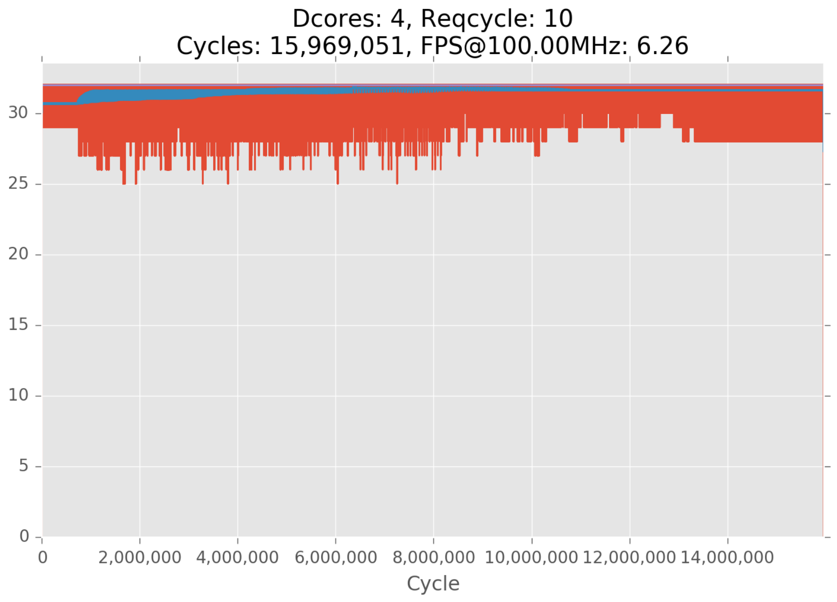
Dcores: 2, Reqcycle: 5
Cycles: 31,935,728, FPS@100.00MHz: 3.13

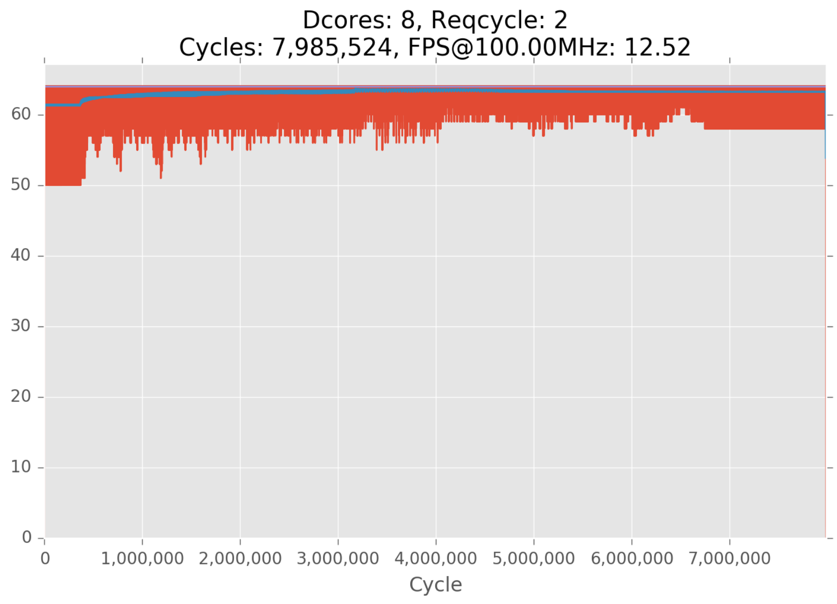


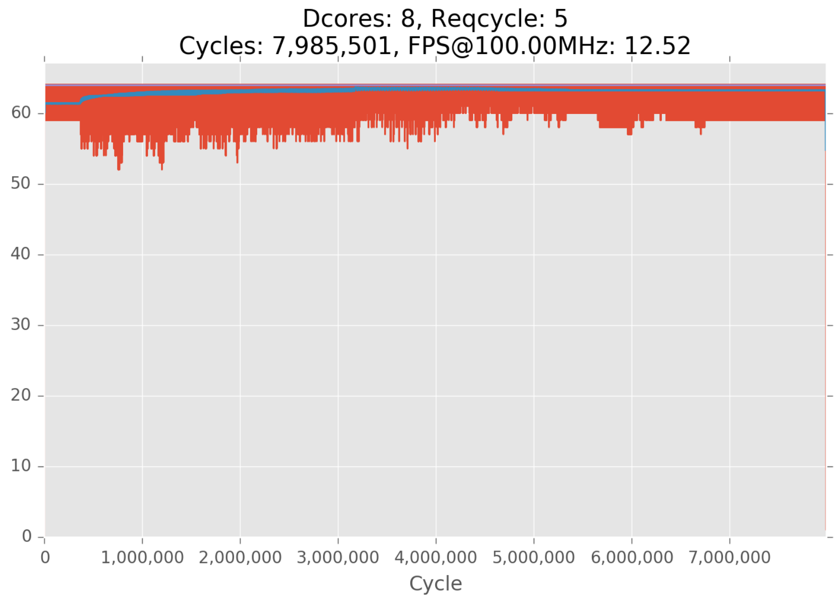

Dcores: 2, Reqcycle: 10
Cycles: 31,935,985, FPS@100.00MHz: 3.13
16
14
12
10
8
6
4
2
0
0
5,000,000
10,000,000
15,000,000
20,000,000
25,000,000
30,000,000
Cycle

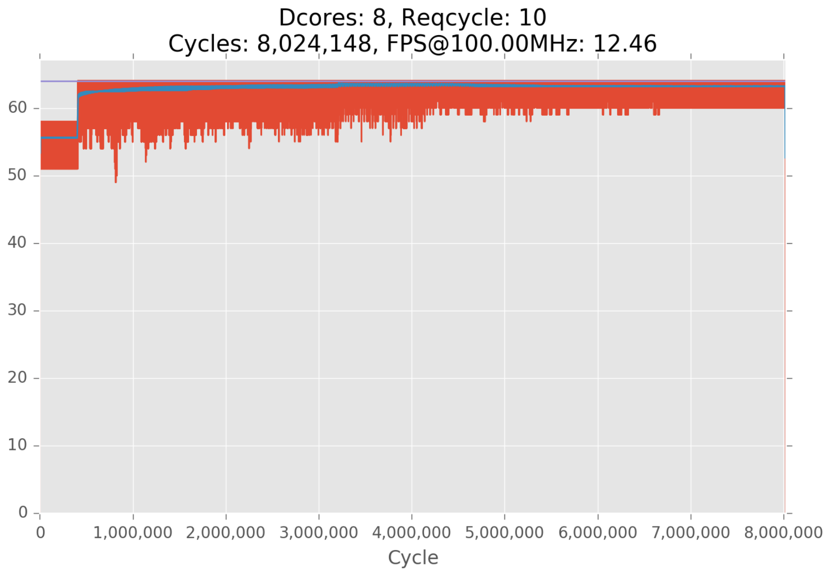

Dcores: 4, Reqcycle: 2
Cycles: 15,968,742, FPS@100.00MHz: 6.26
30
25
20
15
10
5
0
0
2,000,000
4,000,000
6,000,000
8,000,000
10,000,000
12,000,000
14,000,000
Cycle

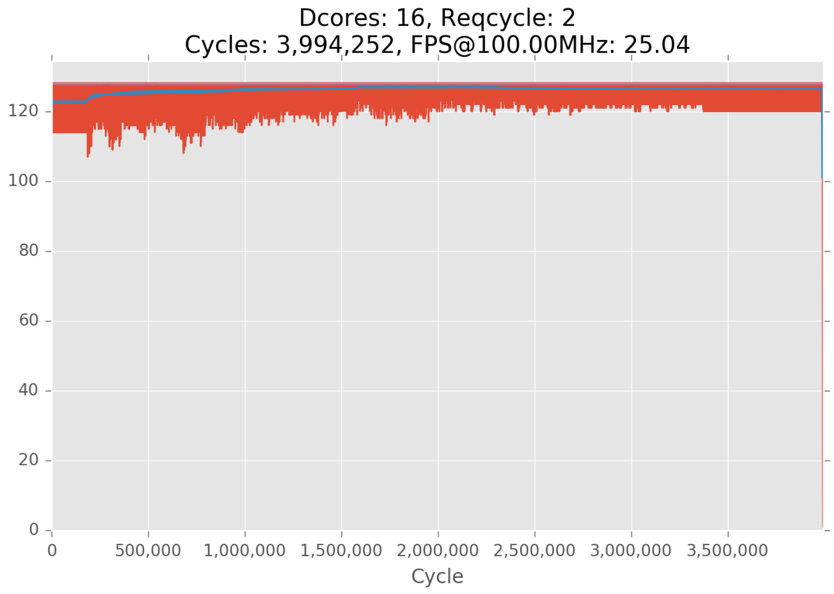

Dcores: 4, Reqcycle: 5
Cycles: 15,968,870, FPS@100.00MHz: 6.26
0
5
10
15
20
25
30
0
2,000,000
4,000,000
6,000,000
8,000,000
10,000,000
12,000,000
14,000,000
Cycle

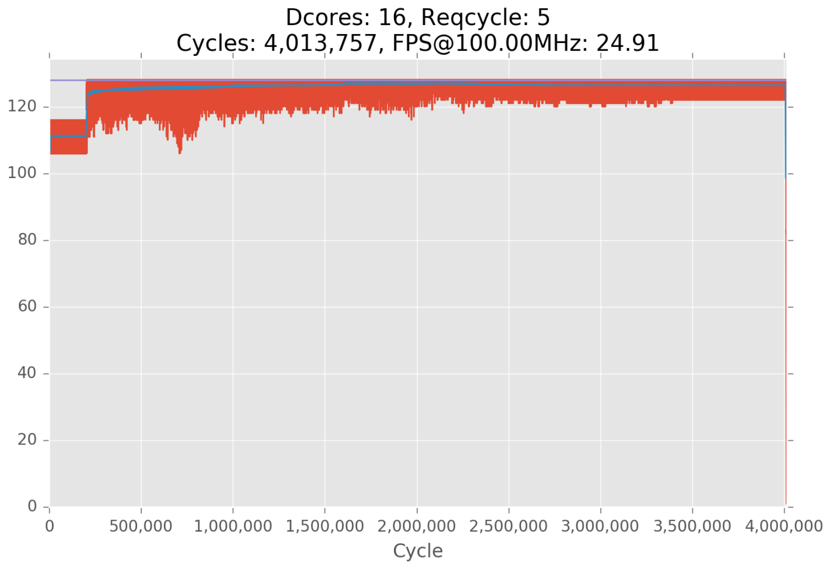

Dcores: 4, Reqcycle: 10
Cycles: 15,969,051, FPS@100.00MHz: 6.26
0
5
10
15
20
25
30
0
2,000,000
4,000,000
6,000,000
8,000,000
10,000,000
12,000,000
14,000,000
Cycle

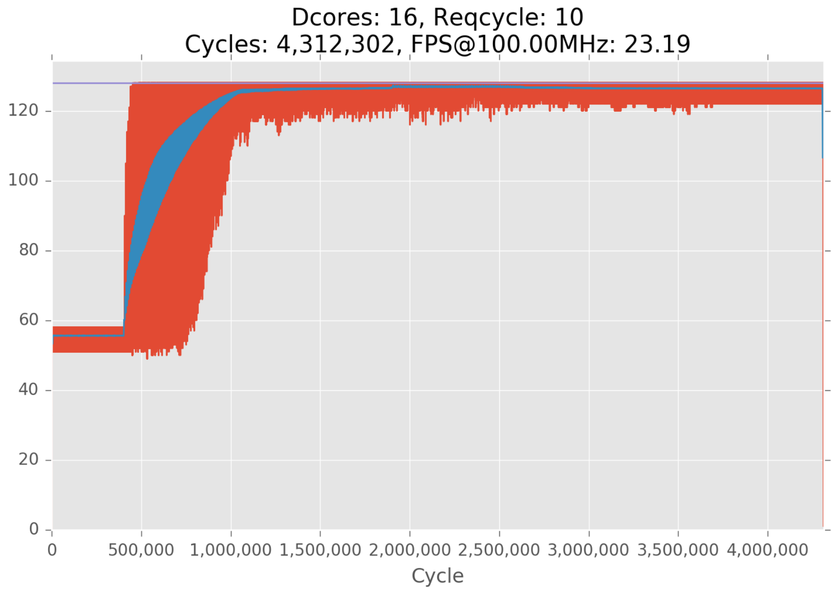

Dcores: 8, Reqcycle: 2
Cycles: 7,985,524, FPS@100.00MHz: 12.52
0
10
20
30
40
50
60
0
1,000,000
2,000,000
3,000,000
4,000,000
5,000,000
6,000,000
7,000,000
Cycle

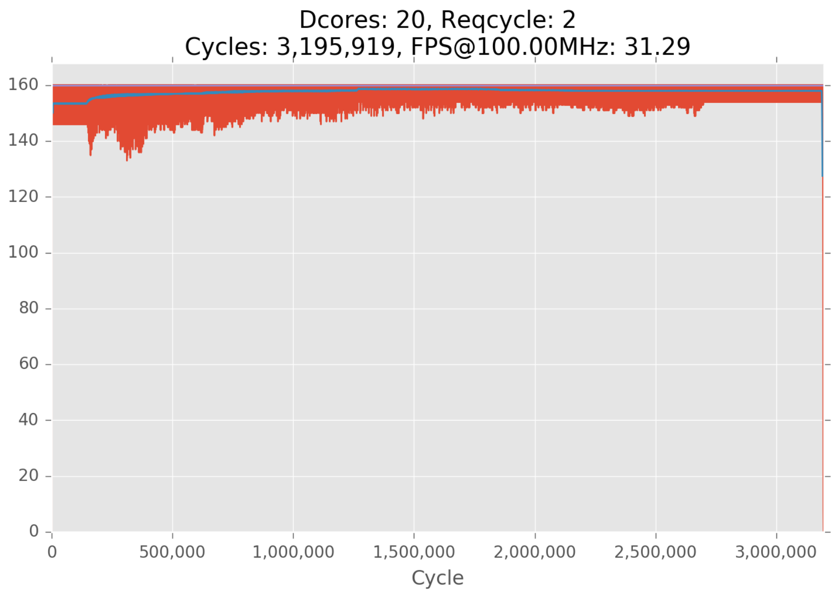

Dcores: 8, Reqcycle: 5
Cycles: 7,985,501, FPS@100.00MHz: 12.52
0
10
20
30
40
50
60
0
1,000,000
2,000,000
3,000,000
4,000,000
5,000,000
6,000,000
7,000,000
Cycle

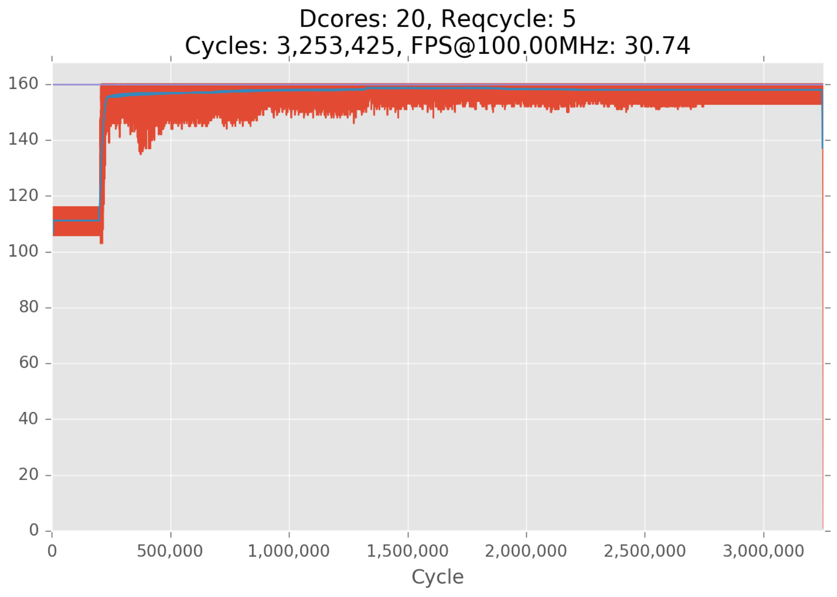

Dcores: 8, Reqcycle: 10
Cycles: 8,024,148, FPS@100.00MHz: 12.46
0
10
20
30
40
50
60
0
1,000,000
2,000,000
3,000,000
4,000,000
5,000,000
6,000,000
7,000,000
8,000,000
Cycle

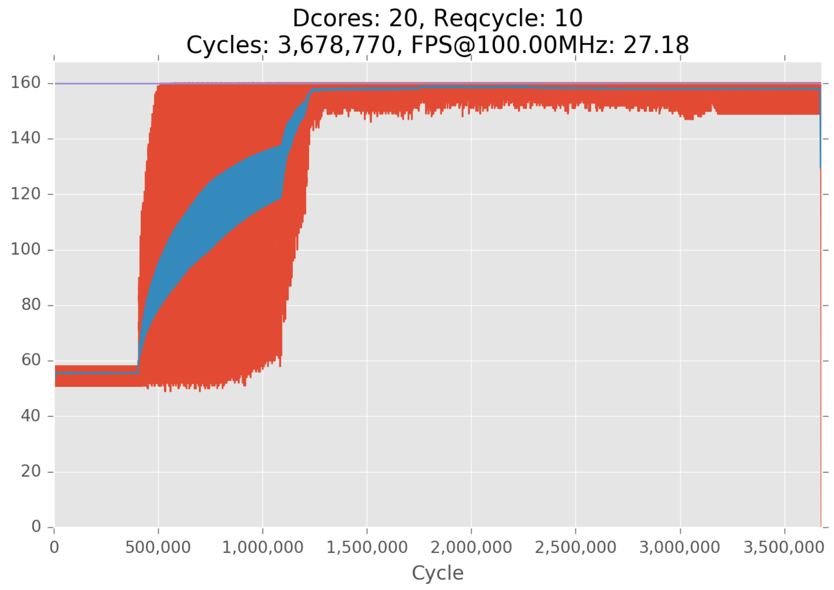

Dcores: 16, Reqcycle: 2
Cycles: 3,994,252, FPS@100.00MHz: 25.04
0
20
40
60
80
100
120
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000
3,000,000
3,500,000
Cycle

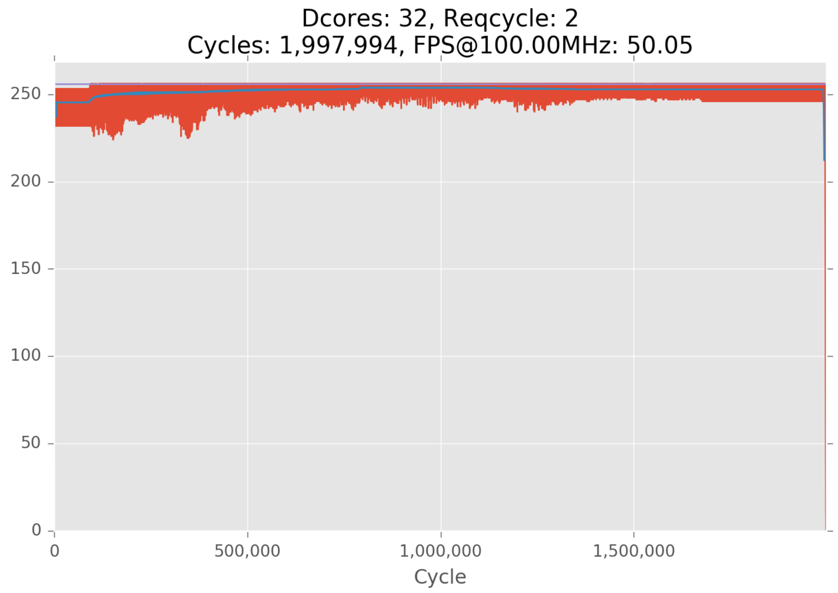

Dcores: 16, Reqcycle: 5
Cycles: 4,013,757, FPS@100.00MHz: 24.91
0
20
40
60
80
100
120
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000
3,000,000
3,500,000
4,000,000
Cycle

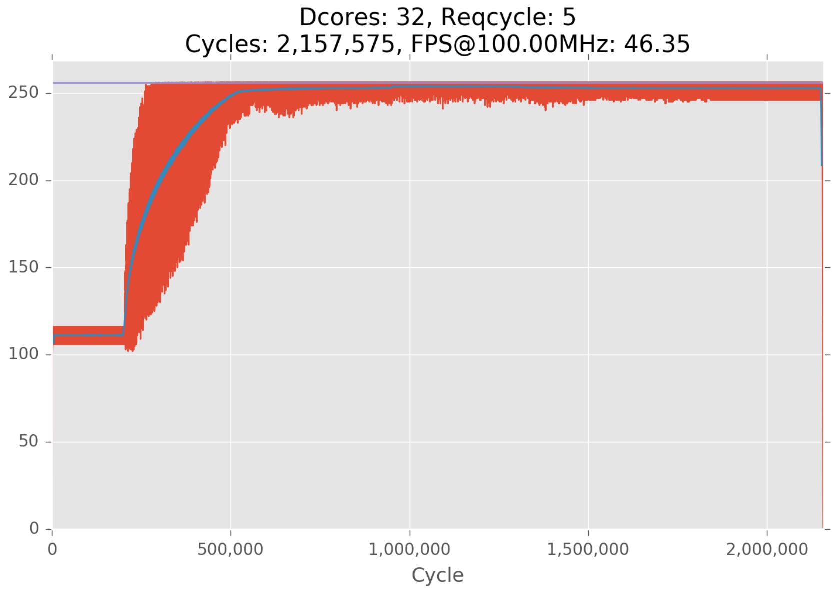

Dcores: 16, Reqcycle: 10
Cycles: 4,312,302, FPS@100.00MHz: 23.19
0
20
40
60
80
100
120
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000
3,000,000
3,500,000
4,000,000
Cycle

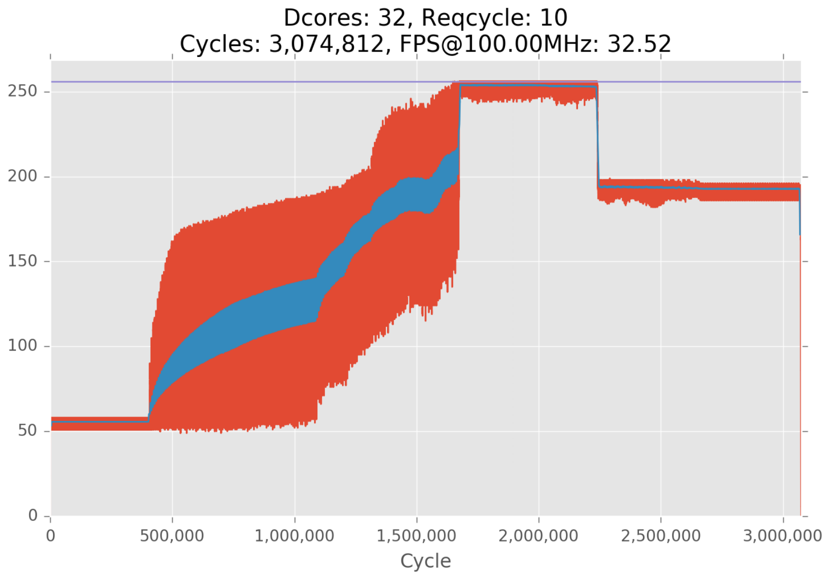

Dcores: 20, Reqcycle: 2
Cycles: 3,195,919, FPS@100.00MHz: 31.29
0
20
40
60
80
100
120
140
160
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000
3,000,000
Cycle


Dcores: 20, Reqcycle: 5
Cycles: 3,253,425, FPS@100.00MHz: 30.74
0
20
40
60
80
100
120
140
160
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000
3,000,000
Cycle


Dcores: 20, Reqcycle: 10
Cycles: 3,678,770, FPS@100.00MHz: 27.18
160
140
120
100
80
60
40
20
0
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000
3,000,000
3,500,000
Cycle


Dcores: 32, Reqcycle: 2
Cycles: 1,997,994, FPS@100.00MHz: 50.05
250
200
150
100
50
0
0
500,000
1,000,000
1,500,000
Cycle


Dcores: 32, Reqcycle: 5
Cycles: 2,157,575, FPS@100.00MHz: 46.35
0
50
100
150
200
250
0
500,000
1,000,000
1,500,000
2,000,000
Cycle


Dcores: 32, Reqcycle: 10
Cycles: 3,074,812, FPS@100.00MHz: 32.52
0
50
100
150
200
250
0
500,000
1,000,000
1,500,000
2,000,000
2,500,000
3,000,000
Cycle

C.2 Synthesis results

Here, the number of LUTs and the achieved frequency f_{max} for different target frequencies (f_{target}) is shown in Tables C.2 and C.3. The number of LUTRAMs, FFs, BRAMs and DSP elements stay the same for all target frequencies. Refer to Table 5.7 for these numbers.

Table C.2: Number of LUTs in the dual core array when synthesised with different target frequencies. N refers to the number of dual cores in the array.

f_{target} (MHz)	$N = 1$	$N = 2$	$N = 4$	$N = 8$	$N = 16$	$N = 20$
50.00	2571	5158	10 323	20 558	41 021	51 114
66.67	2570	5160	10 327	20 563	41 015	51 132
71.43	2569	5158	10 328	20 557	40 992	51 113
76.92	2573	5162	10 336	20 553	40 989	51 083
83.33	2569	5159	10 329	20 560	41 008	51 100
90.91	2569	5159	10 322	20 547	40 991	51 105
100.00	2574	5165	10 320	20 548	41 002	51 098
111.11	2581	5168	10 337	20 558	41 023	51 098
125.00	2617	5231	10 436	20 766	41 419	51 590
142.86	2653	5308	10 559	21 004	41 949	52 104

Table C.3: f_{max} of dual core array in MHz when synthesised with different target frequencies. N refers to the number of dual cores in the array.

f_{target} (MHz)	$N = 1$	$N = 2$	$N = 4$	$N = 8$	$N = 16$	$N = 20$
50.00	80.88	75.86	76.06	78.58	68.14	58.22
66.67	87.82	85.37	83.20	84.77	69.22	72.03
71.43	90.79	84.02	84.60	83.75	74.41	75.77
76.92	88.92	83.02	82.44	84.20	80.08	82.75
83.33	90.35	89.14	85.44	88.06	88.29	85.84
90.91	94.88	92.43	94.84	92.94	92.55	92.00
100.00	101.60	102.00	103.23	100.33	100.05	99.26
111.11	104.63	104.28	104.16	101.28	99.85	97.14
125.00	105.36	104.54	100.18	100.05	98.08	98.87
142.86	108.30	107.49	104.86	105.14	102.78	100.08

Appendix D

Attachment overview

Here, an overview over the structure of the attached zip file is given.

D.1 Design files

The SystemVerilog design files are in this folder. Before this can be synthesised, the assembler has to be run, generating the instruction memory.

```
design/
 |__ alu.sv
 |__ drain.sv
 |__ drain_full.sv ......... Drain breaking timing in both directions
 |__ dual_core.sv
 |__ dual_core_array.sv
 |__ dual_core_array_synthwrapper.sv .......... Used for synthesis
 |__ feeder.sv
 |__ feeder_full.sv........Feeder breaking timing in both directions
 |__ fp32_add.sv
 |__ fp32_add_2s.sv ..........................Used in Section 5.7.4
 |__ fp32_mul.sv
 |__ fp32_mul_2s.sv ..........................Used in Section 5.7.4
 |__ fpu.sv
 |__ fpu_2s.sv................................Used in Section 5.7.4
 |__ fx2fp.sv .............................Generated by fx2fp.py
 |__ instruction_memory.sv
```

```
├── object_buffer.sv
├── ray_core.sv
├── ray_datapath.sv
├── ray_datapath_2s.sv ...................... Used in Section 5.7.4
├── register_file.sv
├── reverse_regslice.sv .........Register slice breaking ready-path
└── types_pkg.sv ...............Type package holding constants etc.
```

D.2 Verification

Before running simulations of the Verilog code, the ISA-simulator must be run first. This generates requests, object buffer data and the expected results. Tools for analysis of results are included.

```
verification/
├── active_threads.py ... Tool used in analysing # of active threads
├── drain_property.sv
├── dual_core_array_tb.sv
├── dual_core_tb.sv
├── feeder_property.sv
├── fp32_add_tb.sv
├── fp32_mul_tb.sv
├── fpu_tb.sv
├── fx2fp_tb.sv
├── object_buffer_tb.sv
├── ray_core_property.sv
├── ray_core_tb.sv
├── ray_datapath_tb.sv
├── responseparser.py ....... Used in verifying correctness of results
└── reverse_regslice_property.sv
```

D.3 Code generators

These Python scripts are used to generate code.

```
generators/
├── assembler.py......................Assembler from Section 5.6.2
├── fp2uint.py .......................... Used in designing fp2uint
└── fx2fp.py................Generates Verilog code for fx2fp module
```

D.4 ISA-simulator

Running the ISA-simulator simulates the system functionality. It runs the assembly program in `program.txt`. During runtime, statistics are generated. Additionally, requests, results and object buffer contents are logged and saved to `verilog/`, for use in Verilog simulations.

```
isasim/
├── fpu/ ............................ Copies of FP-operator models
│   ├── fp32_add.py
│   └── fp32_mul.py
├── rayman/
│   ├── color.py
│   ├── core.py ............................... Ray datapath model
│   ├── cpu.py
│   ├── isectpoint.py
│   ├── material.py
│   ├── memory.py
│   ├── objbuffer.py
│   ├── program.txt ............. Assembly program (same as Ap. B)
│   ├── ray.py
│   ├── raymanager.py ......................... Ray manager model
│   ├── scene.py .................................. Reads scene file
│   ├── sphere.py
│   ├── uvnframe.py
│   └── vector.py
├── results/ ...................... Results/statistics are stored here
│   ├── comparator.py .................. Compares results from runs
│   └── statsreader.py ........ Tool used in generating e.g. Fig. 5.16
└── run.py .................................... Runs the simulator
```

D.5 Scene files

The sample scene shown in Figure 4.4 is attached. A low-resolution version for faster simulations is also attached.

```
scenes/
├── scene_rgb_simple ...................... Scene seen in Figure 4.4
└── scene_rgb_simple_lowres ....... 100 × 100 version of same scene
```

D.6 Data structure documentation

Data structure documentation generator by [Gje17]. Updated in accordance with [Egg16].

`doc/`

`Makefile`	Generates and opens html
`datastructures.html`	
`datastructures.xml`	Data structure specification
`datastructures.xsl`	Stylesheet

D.7 FP-operator modelling

These models were used in exploring and verifying the floating-point operators. They were discussed in Section 5.5. `fp32_sqrt.py` has not been discussed in the thesis. It was used in experiments with a dedicated $\frac{1}{\sqrt{x}}$ and $\sqrt{x}$-unit.

`fpu/`

`addsub_sign.py`	Generates sign truth table
`fp32_add.py`	Model of FP-adder
`fp32_mul.py`	Model of FP-multiplicator
`fp32_sqrt.py`	Experiment with $\frac{1}{\sqrt{x}}$ and $\sqrt{x}$-unit

Appendix E

Coverage report

```
Coverage Report Summary Data by file

==================================================================
=== File: /home/jonasae/src/rayman/verilog/alu.sv
==================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           50        50         0     100.0
    Branches                        48        48         0     100.0
    Toggle Bins                    131       131         0     100.0

==================================================================
=== File: /home/jonasae/src/rayman/verilog/drain.sv
==================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           12        12         0     100.0
    Branches                         4         4         0     100.0
    FEC Expression Terms            17        17         0     100.0
    Toggle Bins                    944       784       160      83.0

==================================================================
=== File: /home/jonasae/src/rayman/verilog/dual_core.sv
==================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------          ------      ----    ------ ---------
    Stmts                          8         8         0     100.0
    Branches                       2         2         0     100.0
    FEC Expression Terms          14        14         0     100.0
    Toggle Bins                 1182       961       221      81.3

==================================================================
=== File: /home/jonasae/src/rayman/verilog/dual_core_array.sv
==================================================================
    Enabled Coverage          Active      Hits    Misses % Covered
    ----------------          ------      ----    ------ ---------
    Stmts                          6         6         0     100.0
    Toggle Bins                  806       656       150      81.3

==================================================================
=== File: /home/jonasae/src/rayman/verilog/dual_core_array_tb.sv
==================================================================
    Enabled Coverage          Active      Hits    Misses % Covered
    ----------------          ------      ----    ------ ---------
    Stmts                         41        41         0     100.0
    Branches                      12        12         0     100.0
    FEC Condition Terms            4         3         1      75.0
    FEC Expression Terms           3         3         0     100.0
    Toggle Bins                  812       530       282      65.2

==================================================================
=== File: /home/jonasae/src/rayman/verilog/fpu.sv
==================================================================
    Enabled Coverage          Active      Hits    Misses % Covered
    ----------------          ------      ----    ------ ---------
    Stmts                         29        29         0     100.0
    Branches                      26        26         0     100.0
    FEC Condition Terms            2         2         0     100.0
    Toggle Bins                  343       343         0     100.0

==================================================================
=== File: /home/jonasae/src/rayman/verilog/fpu/fp32_add.sv
==================================================================
    Enabled Coverage          Active      Hits    Misses % Covered
    ----------------          ------      ----    ------ ---------
    Stmts                        169       169         0     100.0
    Branches                     371        371          0     100.0
    FEC Condition Terms            2          2          0     100.0
    FEC Expression Terms          10         10          0     100.0
    Toggle Bins                 1036       1028          8      99.2

==================================================================
=== File: /home/jonasae/src/rayman/verilog/fpu/fp32_mul.sv
==================================================================
    Enabled Coverage          Active       Hits     Misses % Covered
    ----------------          ------       ----     ------ ---------
    Stmts                         22         22          0     100.0
    Branches                       6          6          0     100.0
    FEC Condition Terms            3          3          0     100.0
    FEC Expression Terms           7          7          0     100.0
    Toggle Bins                  694        688          6      99.1

==================================================================
=== File: /home/jonasae/src/rayman/verilog/fx2fp.sv
==================================================================
    Enabled Coverage          Active       Hits     Misses % Covered
    ----------------          ------       ----     ------ ---------
    Stmts                        131         29        102      22.1
    Branches                     130         28        102      21.5
    Toggle Bins                   64         26         38      40.6

==================================================================
=== File: /home/jonasae/src/rayman/verilog/feeder.sv
==================================================================
    Enabled Coverage          Active       Hits     Misses % Covered
    ----------------          ------       ----     ------ ---------
    Stmts                         13         13          0     100.0
    Branches                       6          6          0     100.0
    FEC Expression Terms          18         18          0     100.0
    Toggle Bins                  458        448         10      97.8

==================================================================
=== File: /home/jonasae/src/rayman/verilog/instruction_memory.sv
==================================================================
    Enabled Coverage          Active       Hits     Misses % Covered
    ----------------          ------       ----     ------ ---------
    Stmts                          4          4          0     100.0
    Toggle Bins                    124        124          0     100.0

=====================================================================
=== File: /home/jonasae/src/rayman/verilog/object_buffer.sv
=====================================================================
    Enabled Coverage            Active       Hits     Misses % Covered
    ----------------            ------       ----     ------ ---------
    Stmts                            9          9          0     100.0
    Branches                         6          6          0     100.0
    Toggle Bins                    236        234          2      99.1

=====================================================================
=== File: /home/jonasae/src/rayman/verilog/ray_core.sv
=====================================================================
    Enabled Coverage            Active       Hits     Misses % Covered
    ----------------            ------       ----     ------ ---------
    Stmts                           93         93          0     100.0
    Branches                        64         64          0     100.0
    FEC Condition Terms             16         16          0     100.0
    FEC Expression Terms             4          4          0     100.0
    FSMs                                                          82.1
        States                       9          9          0     100.0
        Transitions                 14          9          5      64.2
    Toggle Bins                   1100        922        178      83.8

=====================================================================
=== File: /home/jonasae/src/rayman/verilog/ray_datapath.sv
=====================================================================
    Enabled Coverage            Active       Hits     Misses % Covered
    ----------------            ------       ----     ------ ---------
    Stmts                           71         71          0     100.0
    Branches                        45         45          0     100.0
    FEC Expression Terms            10          6          4      60.0
    Toggle Bins                    916        826         90      90.1

=====================================================================
=== File: /home/jonasae/src/rayman/verilog/register_file.sv
=====================================================================
    Enabled Coverage            Active       Hits     Misses % Covered
    ----------------            ------       ----     ------ ---------
    Stmts                            4          4          0     100.0
    Branches                            6          6          0      100.0
    Toggle Bins                       156        156          0      100.0

TOTAL ASSERTION COVERAGE: 100.0%  ASSERTIONS: 64

Total Coverage By File (code coverage only, filtered view): 88.5%
```