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

2011

MSc THESIS

SIMD Floating Point Extension for Ray Tracing

Yunus Ökmen

CE-MS-2011-30

Abstract

In the last decade, the importance of graphics capabilities have become very important in the mobile market. As a result low power embedded solutions for mobile devices have been developed to run computationally intensive graphics applications, which extensively uses floating point calculations. The work proposed in this thesis target the extension of the Silicon Hive processors capabilities for graphics applications. The Silicon Hive core generation flow that allows to introduce a very high degree of parallelism can be efficiently used to generate a processor for graphics. In order to achieve that, in this thesis, we present an hybrid VLIW/SIMD floating point processor derived from the base Silicon Hive VLIW architecture, Pearl Ray. The hardware implementation of floating point functional units is realized using the Synopsys DesignWare building blocks, which are designed in a way that allows the efficient use of register retiming option in the Design Compiler flow, in order to introduce pipeline stages and improve the timing. The proposed architecture can process 8-way vectors, consisting of 32-bit vector elements. To evaluate the efficiency of the proposed architecture a Ray Tracing algorithm, has been mapped on the developed processor. We have shown that the ray tracing algorithm efficiently exploits the full power of floating point vector instructions and also the instruction level parallelism provided by both the VLIW and the SIMD (Vector) nature of our processor. The results shown that a close-to-linear speed-up can be achieved for the Ray Tracing algorithm using the proposed architecture. Finally, the performance of the proposed extended VLIW/SIMD floating point processor has been compared with a very high-end graphic processing unit and a general purpose processor, in terms of number of cycles, total execution time and power consumption on the same Ray Tracing algorithm. The results show that proposed extended Silicon Hive processor can compete with both the GPU and the CPU in terms of execution times. Furthermore, it overperforms the 8 core machine after the execution time is normalized with respect to corresponding clock frequencies. The overall performance on the GPU is slightly better than the proposed processor. However, the advantage of our extended embedded processor becomes clear when the area and the power consumption values are taken into account. Whereas the GPU and the CPU consumes around 140 watt and 85 watt power, respectively, our floating point VLIW/SIMD processor consumes only 0.2-0.3 watt.

Faculty of Electrical Engineering, Mathematics and Computer Science

SIMD Floating Point Extension for Ray Tracing

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Yunus Ökmen
born in Gölcük, Turkey

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

SIMD Floating Point Extension for Ray Tracing

by Yunus Ökmen

Abstract

In the last decade, the importance of graphics capabilities have become very important in the mobile market. As a result low power embedded solutions for mobile devices have been developed to run computationally intensive graphics applications, which extensively uses floating point calculations. The work proposed in this thesis target the extension of the Silicon Hive processors capabilities for graphics applications. The Silicon Hive core generation flow that allows to introduce a very high degree of parallelism can be efficiently used to generate a processor for graphics. In order to achieve that, in this thesis, we present an hybrid VLIW/SIMD floating point processor derived from the base Silicon Hive VLIW architecture, Pearl Ray. The hardware implementation of floating point functional units is realized using the Synopsys DesignWare building blocks, which are designed in a way that allows the efficient use of register retiming option in the Design Compiler flow, in order to introduce pipeline stages and improve the timing. The proposed architecture can process 8-way vectors, consisting of 32-bit vector elements. To evaluate the efficiency of the proposed architecture a Ray Tracing algorithm, has been mapped on the developed processor. We have shown that the ray tracing algorithm efficiently exploits the full power of floating point vector instructions and also the instruction level parallelism provided by both the VLIW and the SIMD (Vector) nature of our processor. The results shown that a close-to-linear speed-up can be achieved for the Ray Tracing algorithm using the proposed architecture. Finally, the performance of the proposed extended VLIW/SIMD floating point processor has been compared with a very high-end graphic processing unit and a general purpose processor, in terms of number of cycles, total execution time and power consumption on the same Ray Tracing algorithm. The results show that proposed extended Silicon Hive processor can compete with both the GPU and the CPU in terms of execution times. Furthermore, it overperforms the 8 core machine after the execution time is normalized with respect to corresponding clock frequencies. The overall performance on the GPU is slightly better than the proposed processor. However, the advantage of our extended embedded processor becomes clear when the area and the power consumption values are taken into account. Whereas the GPU and the CPU consumes around 140 watt and 85 watt power, respectively, our floating point VLIW/SIMD processor consumes only 0.2-0.3 watt.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2011-30
Committee Members	:	

Advisor:	Georgi Gaydadjiev, CE, TU Delft
Advisor:	Menno Lindwer, Silicon Hive, Intel BV.
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Wauter Serdijn, ME, TU Delft
Member:	Carlo Galuzzi, CE, TUDelft

I dedicate this thesis to my beloved parents,
İsmail Hakkı Ökmen and Berrin Ökmen

Contents

List of Figures

List of Tables

Acknowledgements

I owe sincere thankfulness to my supervisor, Menno Lindwer, for the supervision and the support he gave throughout the course of my thesis. I would like to thank Georgi Gaydadjiev for his encouragement, patience and time. It would not have been possible without his guidance. I am also indebted and truly thankful to Carlo Galuzzi for his help and encouragement during the most critical stages of my dissertation writing. I would also like to thank my other committee members, Koen Bertels and Wouter Serdijn.

I am obliged to the all people at Silicon Hive, Intel BV, who spared their time generously to help me. It was a pleasure for me to share the same environment with them during these nine months. Especially, I would like to mention Mauro Cocco, Allessandro Paschina, Harm Peters, Carlos Alba Pinto and Alexandre Nery for their support.

My gratitude also goes to my friends Emre Buyukcerci, Onur Kaya and Halil Kukner for their support to write my thesis. I appreciate my house-mates Aashini Gulati, Gokturk Cinserin and Alper Kemal Koc for standing by me in every difficult situation. I would also like to thank my friends in Delft for all the good times we have had together, especially Sinan Tufekcioglu hosting me in his house during my stays in Delft. Finally, I would like to thank my friends and cousins in Turkey, for their support.

At last and truly the most, I would like to thank my parents, İsmail Hakkı and Berrin Ökmen for their love and support. I cannot express what I feel towards my parents with the word ”thanks”, for providing everything they have, to support me all the time. Their dedication and encouragement has made this work possible. If where I am now is an achievement, that is something that belongs to my parents, and I would be more than honored to dedicate this thesis to them.

Annecim ve babacım, sizin sonsuz desteğiniz olmadan bu tezi tamamlayamazdım. Bugüne kadar elde ettigim tüm başarılardaki payınız yadsınamaz ve her şeyden değerli. Sizi çok seviyorum.

Yunus Ökmen
Delft, The Netherlands
October 28, 2011

Introduction 1

The technology is moving very fast in the mobile world. The devices shrink in size and, at the same time, their need for complex computation expands. In the last decade, the importance of graphics has increased in the mobile market and today's market offers many low power embedded solutions for mobile devices to run computationally intensive graphics applications. These applications extensively use floating point calculations. As a result, a floating point calculation capability plays an important role to meet the performance requirements in the embedded domain.

Silicon Hive is a company that targets the mobile multimedia domain and develops low power programmable parallel processor technology solutions. This technology is embedded and applied in high performance camera systems, video systems, and wireless communications. Its innovative processors, which combine features from different architectural styles, enable system-on-chips boasting throughput of 100's of Giga-operations per second.

In this thesis, our target is the extension of the capability of a Silicon Hive processor for graphics applications. The Silicon Hive core generation flow that allows to introduce a very high degree of parallelism can be efficiently used to generate a processor for graphics. In order to achieve that, in this thesis, we present a VLIW/SIMD floating point processor based on this technology.

1.1 Overview

The proposed generic VLIW floating point vector processor is implemented via the Silicon Hive development flow. The proposed architecture can process 8-way vectors, consisting of 32 bit vector elements, in which both integers and single precision floating point numbers are treated. A basic Silicon Hive VLIW processor,the Pearl Ray processor, with three instructions wide issue slot, is used as the baseline processor to extend with new capabilities. First, we had introduced scalar floating point functional units into the processor. We have used IEEE 754 compliant 32-bit single precision floating point format. Included floating point are addition, subtraction, multiplication, division, square root, float-to-integer and integer-to-float conversions. Finally, a vector issue slot, which includes both integer and floating point vector operations, a vector memory and vector register files, are also included. The proposed VLIW architecture can process 8-way vectors, consisting of 32 bit vector elements. The final result is a hybrid VLIW/SIMD floating point processor.

The hardware implementation of floating point functional units is realized using the Synopsys DesignWare building blocks. These blocks consist of full combinational logic. However, they are designed in a way that allows efficient use of the register retiming option in the Design Compiler flow. Without any register retiming, they cannot reach our target frequency of 333 MHz. In order to achieve that frequency, the register retiming technique has been used. For those functional units failed to satisfy the timing constraints, a number of registers are added to their input side. By using the Synopsys' retiming engine, these registers are propagated through the combinational logic to build optimal pipeline stages. In order to find the minimum number of registers required to meet the timing constraints, experiments are conducted for various number of pipeline stages.

To evaluate and show the efficiency of the proposed architecture, the Ray Tracing algorithm, has been mapped on the developed processor. The Ray Tracing algorithm intensively uses floating point operations and it provides a high level of data parallelism. As part of the work in this thesis, we have shown that the parallel nature of the ray tracing algorithm can efficiently utilize the floating point computation resources provided by the proposed processor. It efficiently exploits the full power of floating point vector instructions and the instruction level parallelism provided by both the VLIW and the SIMD (Vector) nature of our processor. The results shown that a close-to-linear speed-up can be achieved for the Ray Tracing algorithm using the proposed architecture.

In order to have better matching of the architecture and the algorithm, they are both co-developed. First, the ray tracing algorithm is vectorized to exploit SIMD instructions. Additionally, the algorithm is further optimized to increase the instruction level parallelism. Recursions, that are the bottleneck for the parallel nature of the algorithm, are removed using methods like function inlining and loop unrolling are utilized. Finally, the architecture is refined by including multiple vector issue slots rather than having single one that includes all the vector functional units. The copies of functional units, which are frequently used by the ray tracing application, are introduced in more than one issue slot. The final configuration of the multi-issue slots is determined after analyzing the algorithm performance for various issue slot configurations.

Lastly, the performance of the ray tracing algorithm on our architecture is compared with other platforms. The same algorithm is implemented on a contemporary GPU and on a CPU. For the Nvidia Quad 4000 GPU, the algorithm is rewritten for the Nvidia CUDA environment and simulated. Similarly, an OpenMP version of the same algorithm is implemented and executed on an 8 core Intel Xeon processor. The results showed that the proposed extended Silicon Hive processor can compete with both the GPU and the CPU in terms of execution times. Furthermore, it over performs the 8 core machine after the execution time is normalized with respect to the corresponding clock frequencies. The overall performance on the GPU is slightly better than the proposed processor. However, the advantage of our extended embedded processor appears, when the area and the power consumption values are taken into account. Whereas the GPU and the CPU consumes around 140 watt and 85 watt power, respectively, our floating point VLIW/SIMD processor consumes only 0.2-0.3 watt.

1.2 Outline of the Thesis

The rest of the thesis is organized as follows:

- In Chapter 2, we present the background information. The Silicon Hive technology and the ASIP solutions, which offer a competitive performance in the embedded domain, are introduced. Additionally, the Silicon Hive processors, system solutions, and development flow are briefly explained. The details of the Pearl Ray processor, which is the baseline processor for the work presented in this thesis are given. Additionally, the Synopsys Environment and the features of DesignWare Library of Synopsys are described.

- In Chapter 3, the extensions of the baseline processor are described. First, we present the targeted floating point operations. After that, we present the hardware implementation of these operations using DesignWare floating point building blocks. The method for introducing floating point capability using the Silicon Hive core generation flow is explained in detail. Secondly, vector extensions are covered and we explain how the vector (SIMD) instructions are introduced into the baseline processor. Finally, the general overview of the proposed floating point VLIW/SIMD processor is presented and the synthesis results are introduced and described.

- In Chapter 4, we describe the Ray Tracing algorithm that we used to evaluate the performance of the proposed processor. The vectorization of the ray tracing, in order to exploit SIMD instructions, is explained. Further optimizations to exploit the ILP of the VLIW architecture is presented. Finally, the refinement of the processor for the algorithm is shown. Finally, we present results on the execution time improvements of the processor after each optimization step.

- Chapter 5 contains the details regarding the experimental results. The performance of our processor is compared with those of a GPU and a CPU. Implementation details on the CPU and the GPU are given as well. Finally, a comparison between the power and the execution time results is presented.

- In Chapter 6, we finally conclude with the conclusive remarks and a summary of the work presented in this thesis. Possible ideas to further extent the work presented in this thesis are also discussed.

Background Information 2

In the introductory chapter of this thesis, we state the goal of this work: The extension of a baseline processor with floating point capabilities. In this chapter, we first introduce Silicon Hive and the ASIP solutions it offers for the embedded domain. Afterthat, the Silicon Hive processors, system solutions, and development flow are briefly explained. The details on the baseline to extend via the work presented in this thesis, are provided. After that, we describe the Synopsys Environment and the features of the DesignWare IP of Synopsys.

2.1 Silicon Hive Technology

The performance needs in the embedded domain is growing day by day. However, general purpose processors are very costly in terms of area and power to satisfy the need for performance. As a consequence, application specific embedded computing systems gained popularity like the Application Specific Integrated Circuits (ASIC) in the embedded world. ASICs are dedicated hardware specifically designed for a domain of applications. As a result, they can provide very efficient and optimized solutions in terms of power and area. However, a variety of factors makes difficult and expensive the design and manufacturing of traditional ASICs. For example, bugs can be founded after the production of the silicon and without programmability, in case there is a bug, the hardware has to be redesigned and reproduced. This increases the overall time-to-market and cost of the design. Programmability provides higher volume to amortize design and manufacturing costs, as the same platform can be used over multiple related applications, as well as over different versions of an application. The flexibility provided by programmability comes with a performance and power overhead. This can be significantly mitigated by using application specific platforms, also referred as Application Specific Instruction Set Processors (ASIPs) [19].

On the near extreme, the ASIC (Application Specific Integrated Circuit) offers low-cost, fast turnaround, and tailored hardware solutions. On the far extreme, the programmability of processors offers flexibility, but sometimes at a high cost. The ASIP (Application Specific Instruction-Set Processor) is a balance between these two extremes. ASIPs offer the availability of custom sections for time critical tasks (e.g. a Multiply-Adder for real-time DSP), and offer flexibility through an instruction-set. They can be finely tuned to run a small range of applications very efficiently, while keeping the ability to run other tasks through a micro-code program [21]. The relative position of the flexibility and performance of ASIP designs taken in comparison to programmable DSP processors and ASIC solutions is depicted in Figure 2.1. Retargetable ASIP based designs represent an alternative to ASIC and general-purpose based solutions. They aim at matching the programmability of general purpose solutions while keeping the efficient

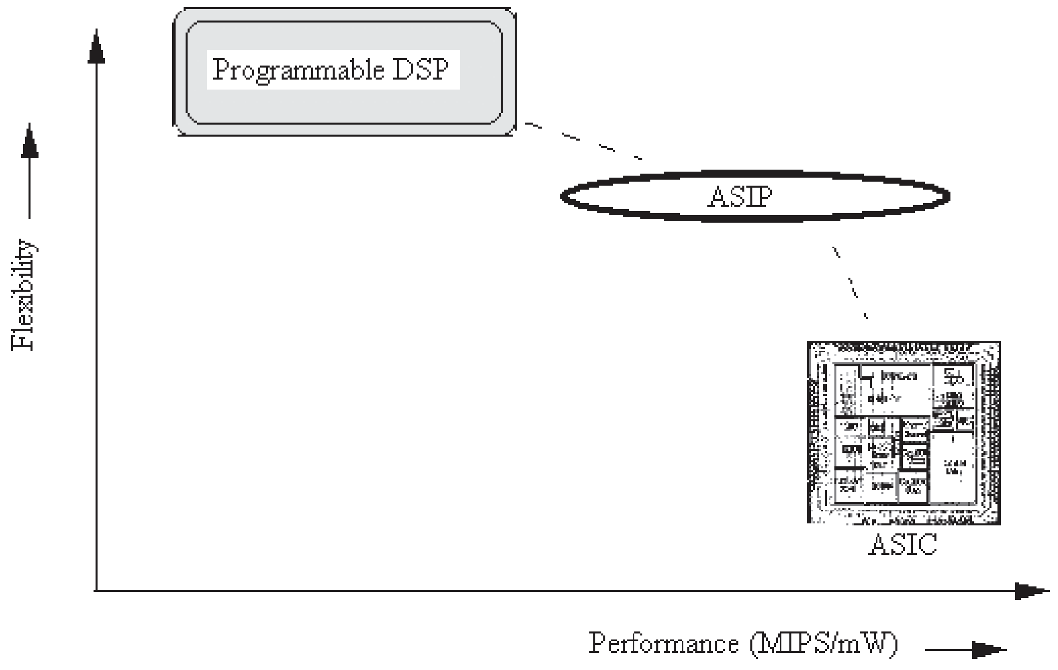

Figure 2.1: ASIP programmability and performance comparison. [8]

performance t of ASIC solutions [8].

The Silicon Hive produces and licences ASIP processors. The target application domains of the Silicon Hive are image and video solutions. It also provides advanced software development tools and application libraries to enable semiconductor companies to create fully programmable System on Chips (SoCs). Moreover, The Silicon Hive environment offers a high level of configurability for the processors and systems, which can be easily adapted to different kind of application domains [15]. A detailed description of Silicon Hive processors, tools and environment is provided in the rest of this section.

2.1.1 The Silicon Hive Processors

It is very difficult to categorize a Silicon Hive processor, as it presents features from different architectural styles [13]. This type of processors are built according to the load/store architecture and use compiler-directed scheduling (RISC). They have a user configurable instruction set, like the ARM processors, a massive number of issue slots (VLIW) and single instruction multiple data issue slots (SIMD).

The main feature these processors is probably the Very Long Instruction Word (VLIW) capability. Dozens of issue slots can be deployed in a Silicon Hive processor. For this reason, a Silicon Hive processor is sometimes called also Ultra Long Instruction Word (ULIW) architecture, which is an apt description. These processors execute

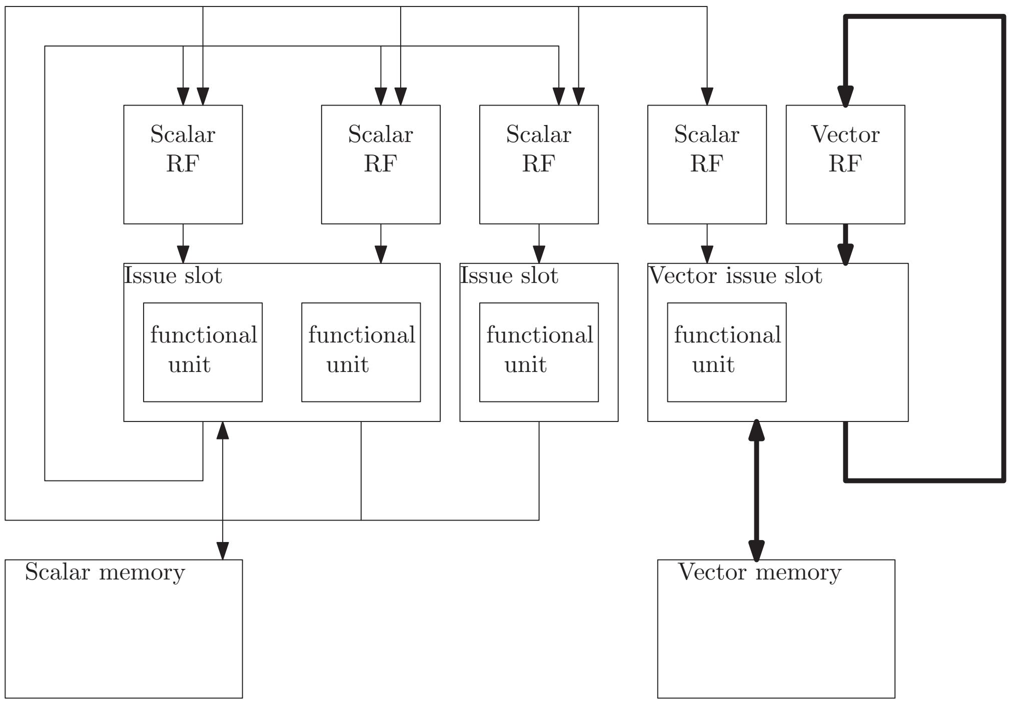

Figure 2.2: Example of a VLIW Silicon Hive processor with multiple scalar and a vector data path.

long instructions that can contain multiple operations, where an operation, in our context, corresponds to what is otherwise called an instruction. As a result, a high level of instruction level parallelism can be achieved. The hardware executes an instruction by executing all its operations in parallel, without having to check for dependencies or resource conflicts among them, as the compiler handles all the dependencies and the control overhead. Each operation in this very long instruction word is issued by a separate issue slot consisting of many functional units. The capabilities of the functional units can be configured by including new instructions. Additionally, the configuration of the issue slots can be modified by adding or removing different functional units. The performance of the processor can be further increased by including more issue slots. Nevertheless, an increase in the performance depends on the ILP of the application and the efficiency of the compiler used to extract it [13].

Besides the main VLIW capability, Silicon Hive processors can be deployed with Single Instruction Multiple Data (SIMD or vector) issue slots as proposed in [23]. An N-way SIMD vector issue slot can perform N operations on N vector elements in parallel. Figure 2.2 illustrates an example of a Silicon Hive Processor with multiple scalar and a vector issue slot, register files, and memory. In the figure, it is shown how the functional

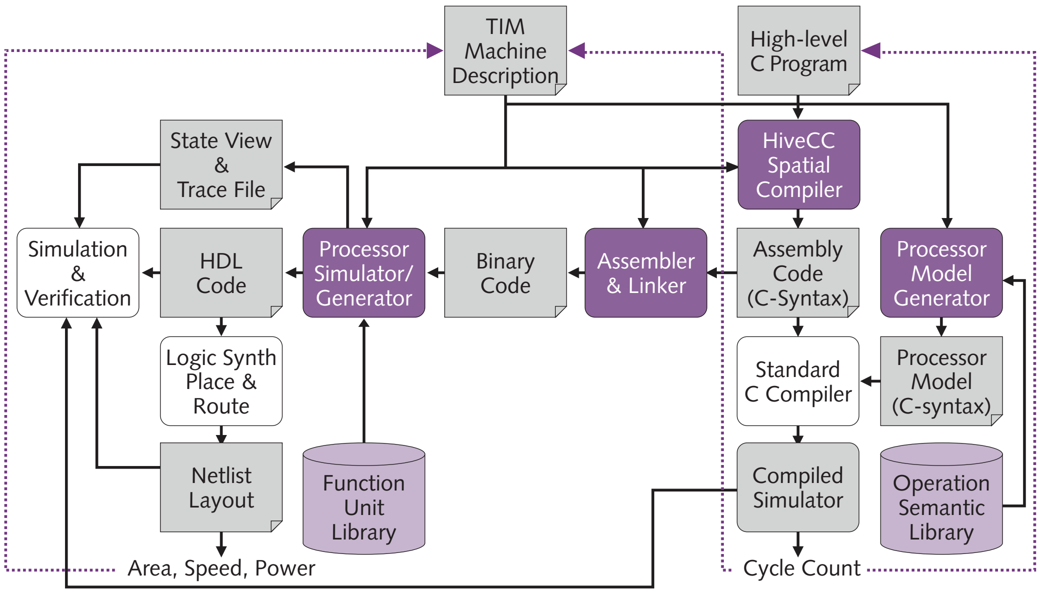

Figure 2.3: The Silicon Hive core generation flow for the generation of optimized implementations of the CPU architecture.

units are located within the issue slots, how the register files are connected to these issue slots and how the interconnections are organized for this example.

2.1.2 The Processor Generation Flow

One of the main strengths of Silicon Hive cores is the capability of being generated and configured in a very short time. The Silicon Hive claims that an optimized ULIW (Ultra Long Instruction Word) architecture can be created in a matter of hours, depending on the amount of fine tuning required. The Silicon Hive has an entire tool chain for the rapid design of custom VLIW cores [15]. A flexible architecture template is used as starting point to generate and/or configure cores. The number of processing units, function units, register files, interconnects, and local memories is customizable and, additionally, new function units, registers and instructions can be added. New issue slots can be added to the template and the lengths of the instruction words as well as the lengths of the operations within the instruction words are configurable.

Figure 2.3 depicts the Silicon Hive system design flow. The flow starts with a TIM (The Incredible Machine) description file, which lies at the heart of the Silicon Hive processor generation flow. By using this language it is possible to specify all the relevant information for the generation, programming, and simulation of a processor. This includes, for example, register file sizes and widths, interconnect, issue slots, operation sets, custom operations, memory and I/O subsystem of the processor.

TIM is a proprietary hardware design language. It is a higher level language than VHDL or Verilog. TIM language allows the configuration of a core by specifying high level parameters, such as the number of function units, register files, interconnects and issue slots, the list of instructions that each function unit can execute, the cycle time information of each function units and others. With this information, TIM also derives other parameters like, for example, the length of the instruction words [13]. Additionally, it invokes the compiler by providing necessary information about the operation semantics, the latencies of the operations for scheduling, etc. By using the TIM language the entire processor can be described with few code lines. As a matter of fact, a mere 300 lines of TIM can result in 100.000 lines of code VHDL code. It can describe a register file by simply specifying the number of registers, the widths of the registers and the number of read/write ports [7]. From these descriptions, TIM invokes pre written blocks of VHDL. TIM also drives the development-tool generator that creates a matching assembler, linker, C compiler, instruction set simulator and cycle accurate simulator [13]. The detailed design flow and co-simulation strategy for the generation of the processor architecture and also the application correlation are shown in Figure 2.3.

Once a TIM file is created, it is tested with representative programs from the application domain. This provides important feedbacks to the designer, such as the scheduling of the instructions according to the processor resources (i.e. register files, issue slots, interconnects), which reflects into the resource utilization. Once the design has been verified, a complete synthesizable RTL hardware description of the processors is generated. Pre-written blocks of VHDL or Verilog (stored in the component library depicted in the flow) are invoked from TIM description and the processor is generated [20]. This flow has several properties, which can help designers to generate processors:

- this flow allows to quickly generate a processor, including the VHDL code;
- this flow allows to have a fast design-space exploration of a processor;
- the generated processors are tuned for specific application domains in terms of area, performance and power trade-offs.

2.1.3 The System Description

Silicon Hive processors must reside in a system, which is need to test the processor and fulfill its functionality. In a Silicon Hive system, multiple processors can be placed together with other components, such as peripherals, memories, and interconnections. In Silicon Hive terminology, a system is considered as the combination of a number of Silicon Hive processors, a host processor, a control bus, FIFO adapters, external memories, and/or custom devices.

Figure 2.4 shows a simple example of a Silicon Hive system including some of these components. A host processor, typically an ARM, ATOM or a similar processor, is responsible for controlling the system. The host (control interface) fulfills the role of system

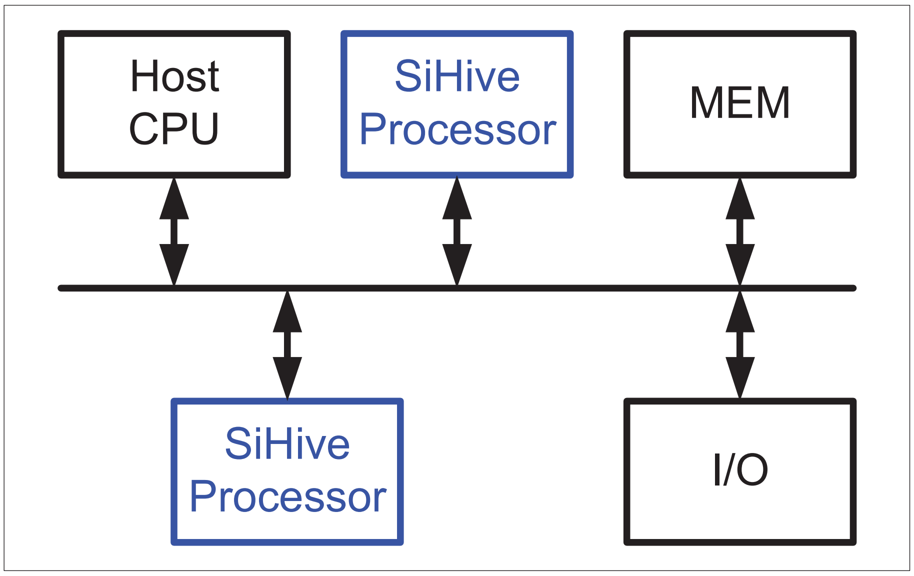

Figure 2.4: The main components of a Silicon Hive system.

controller, where the main tasks are: the initialization of the system components, the download of the applications into the processors, the upload of the required parameters to the processors' local memory or the external memory, and, finally, status checks.

Silicon Hive systems are described in a high-level language called HSD. The properties of the system, the components and the connections can be easily configured using HSD in a hierarchical way. The system description is exploited to generate a system simulator containing all the system-specific information, such as bus address mappings and connectivity [7].

2.1.4 The HiveCC Compiler

The performance of VLIW architectures depends on the instruction-scheduling compiler used. The challenge of a VLIW compiler is to extract the ILP in the application by picking operations that can be executed in parallel and map them efficiently onto the computational resources of a target processor. As a result, a massive number of instructions can be executed in parallel.

HiveCC is the Silicon Hive compiler, which supports innovative code generation, efficient scheduling and resource allocation techniques for VLIW cores. HiveCC, itself, statistically handles all of the processor's pipeline management and forwarding control tasks. This allows for much more efficient processors, where most of the resources are dedicated to process the data, instead of control overhead [7]. Like a VLIW compiler, however HiveCC must use inactive NOPs (No OPerations) to pad any issue slots that it cannot be filled with useful operations. Unlike most VLIW compilers, HiveCC uses constraint-solving scheduling techniques that are capable of dealing with large number of issue slots, register files, and interconnect. The scheduling techniques used by HiveCC

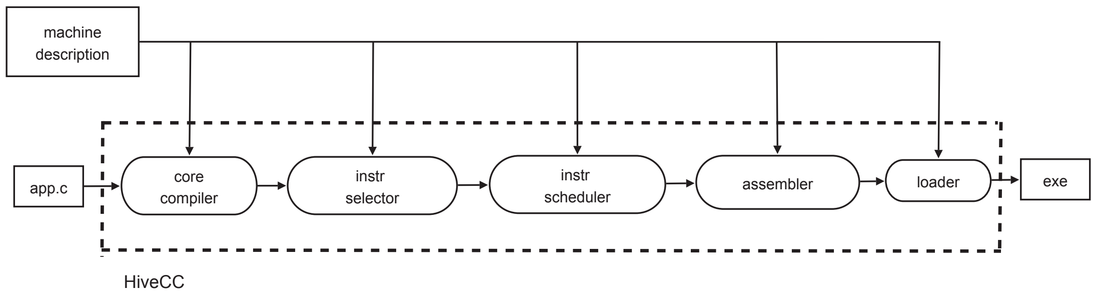

Figure 2.5: The Silicon Hive Compilation Flow.

Simulation Type	Execution Method	Execution Time	Accuracy
C-run	Native	Native	Functionality only
Unsched	Native	$>$ 3 Mops/s	Bit-accurate
Sched	Native	3 Mops/s	Cycle-accurate
System_vhdl	RTL	100 ops/s	Signal-accurate

Table 2.1: Different abstraction levels for the simulations in the Silicon Hive environment.

guarantee that the generated code is optimal, given sufficient compilation time. HiveCC provides two kinds of schedulers, namely *hivesched* and *manifold*. *Manifold* guarantees the optimal solution for the scheduling problem, whereas *hivesched* is based on greedy algorithms, which give a solution in a much shorter time, but does not guarantee the optimality of the solution. In the work presented in this thesis, *hivesched* is used due to the large number of simulations, which are time consuming.

To conclude, the instruction scheduler is a very important part of the HiveCC compiler as it determines the resource utilization of a VLIW processor. If the compiler would not be able to fill the instructions with sufficient parallel operations, the resources not utilized would be wasted. In this respect, HiveCC compiler performs quite well by optimizing the use of resources.

2.1.5 The Silicon Hive Simulation Environment

The Silicon Hive's development environment allows the simulation of the applications on different levels. The abstraction levels differ in timing accuracy and simulation/execution speed [6]. The simulation/execution levels are shown in Table 2.1.

- In *C-run*, both the host program and the codes to be run on a Silicon Hive processor are compiled through the *gcc* compiler into a native simulation code that can be executed by the host. *C-run* is the fastest method, as it only simulates the functionality of the application and excludes many details of a Silicon Hive core flow. These includethe location of the variables, which, in this case, are located in C-memory and not in the Silicon Hive cell memories; additionally no instruction selection or scheduling takes place and no statistics or listing available.

- The next simulation type is *unsched*, the unscheduled simulation. In *unsched*, the host code is compiled with *gcc* and the Silicon Hive code is compiled with the HiveCC compiler, although it is not scheduled. This means that the instruction selection is performed for the Silicon Hive cores, but issue slot allocation, cycle allocation and register allocation do not take place.
- In *sched*, the Silicon Hive code is fully compiled with HiveCC and a cycle accurate simulation takes place. The operations are scheduled according to the architecture of the target core. The compiler also generates an html output, listing, and statistic files, which give detailed information about execution cycles, processor resource utilizations, memory usage, etc.
- The last type of simulation is *system_vhdl* which is the most accurate one. It is a RTL simulation. Testbench files created within the core generation flow are linked with the compiler and a signal level simulation is performed. Besides cycle accurate simulation, cycle count in the core I/O level is taken into account within this simulation.

2.1.6 The Base Processor: Pearl Ray

As a starting processor for the work presented in this thesis, we used the Pearl Ray processor of Silicon Hive. This processor consists of two building blocks, namely Pearl and Ray. Those blocks are referred as Processing and Storing Elements (PSE) in Silicon Hive terminology. PSEs are the most coarse-grained building blocks in the Silicon Hive processor building block library. These blocks can be used serve as a starting point and more fine-grained building blocks can be constructed and included. These include functional units, issue slots, register files and memories, in order to implement the target floating point vector architecture.

A PSE is a VLIW-like data path consisting of several Interconnection Networks (IN), a plurality of operation Issue-Slots (IS), including the corresponding Functional Units (FU), distributed Register Files (RF), and local MEMory storage (MEM) accompanied by a Load/Store Unit (LSU). A PSE allows a high level of configurability. More fine grained building blocks could easily be introduced in a PSE. A general schematic of such a PSE is depicted in Figure 2.6.

Figure 2.7 depicts the architecture of the baseline processor, which is mainly a VLIW processor. Compared to other processors of Silicon Hive, which have tens of issue slots, Pearl Ray is a simpler core, in terms of number of issue slots. However, it carries all the important features of a generic Silicon Hive processor. It contains 3 issue slots in which 3 operations can be issued simultaneously. The first 2 issue slots (bp_pearl_s1 and bp_pearl_s2) belong to the Pearl PSE. The issue slot 1 (bp_pearl_s1) hosts 8 FU's. The BRanch Unit (BRU) and Status Update Unit (SUU) are both used for program control purposes. The instructions for general-purpose processing are provided by the LoGic Unit (LGU), ARithmetic Unit (ARU), SHift Unit (SHU), Sign-Extension Unit (SEU) and PaSs/immediate Unit (PSU). The Send/Receive Unit (SRU) is used for token-based synchronization with external devices (via bidirectional FIFO) or for data streaming from/to external devices. Each of these two issue slots has an associated register file. The

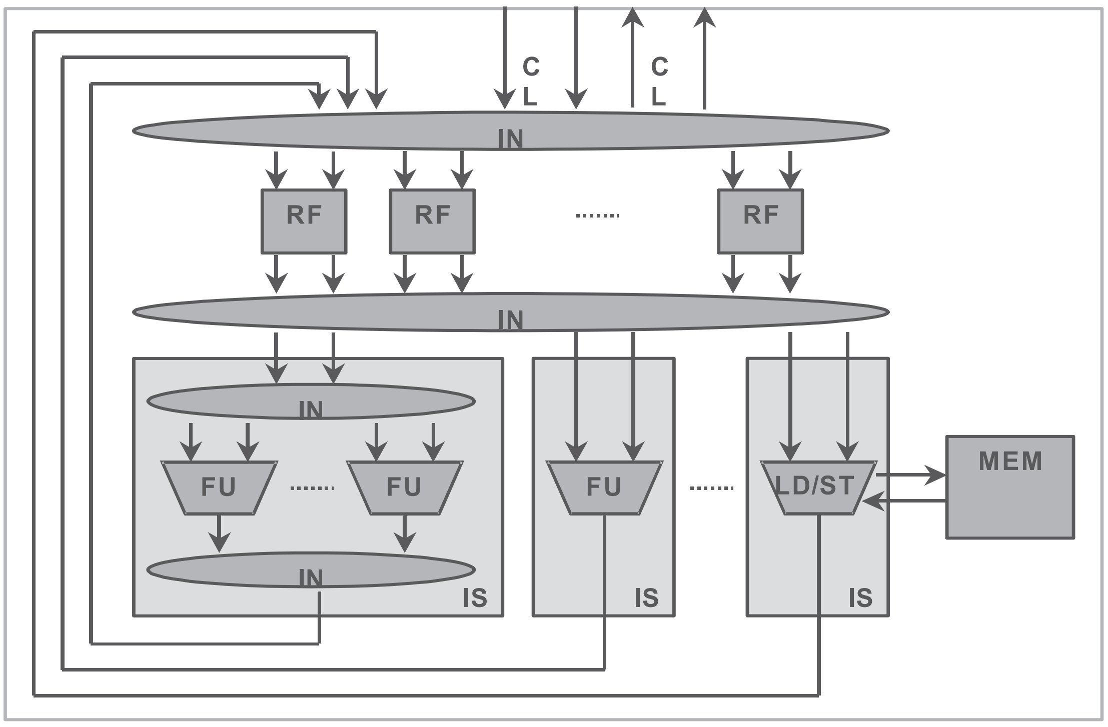

Figure 2.6: General schematic of a Programming and Storage Element (PSE).

size of register file is 16x32 bits. In the issue slot 2 (bp_pearl_s2), 5 FU's are instantiated: a LoGic Unit (LGU), an ARithmetic Unit (ARU), aMULtiply-unit (MUL), a Load-Store Unit (LSU) to access the default memory and a PasS/immediate Unit (PSU).

The Ray PSE contains only a single issue slot (ray_ray_s1, see Figure 2.7). It contains 4 functional units; a Load/ Store Unit (LSU), a LoGic Unit (LGU) and an ARithmetic Unit (ARU). The main difference between the Pearl and the Ray PSE is that in the Ray PSE, the load/store unit is connected to a master interface that accesses to devices outside of the processor.

In the proposed work, we aim at configuring issue slots of the baseline processor by introducing new functional units capable of executing floating point arithmetic operations. Additionally, the Silicon Hive core generation flow allows us to deploy the new issue slots. By using that feature, a SIMD issue slot can be introduced with the introduction of the proper register files, load/store units and a vector memory.

2.2 The Synopsys Environment

Synopsys, is a world leader in Electronic Design Automation (EDA). Moreover, It is a leading provider of high-quality, silicon-proven IP solutions for SoC designs used in semiconductor design, verification and manufacturing. In this thesis, the DesignWare IP library of Synopsys, and the Synopsys Design Compiler for hardware synthesis and optimization have been utilized. In the following sections, we will describe them in detail.

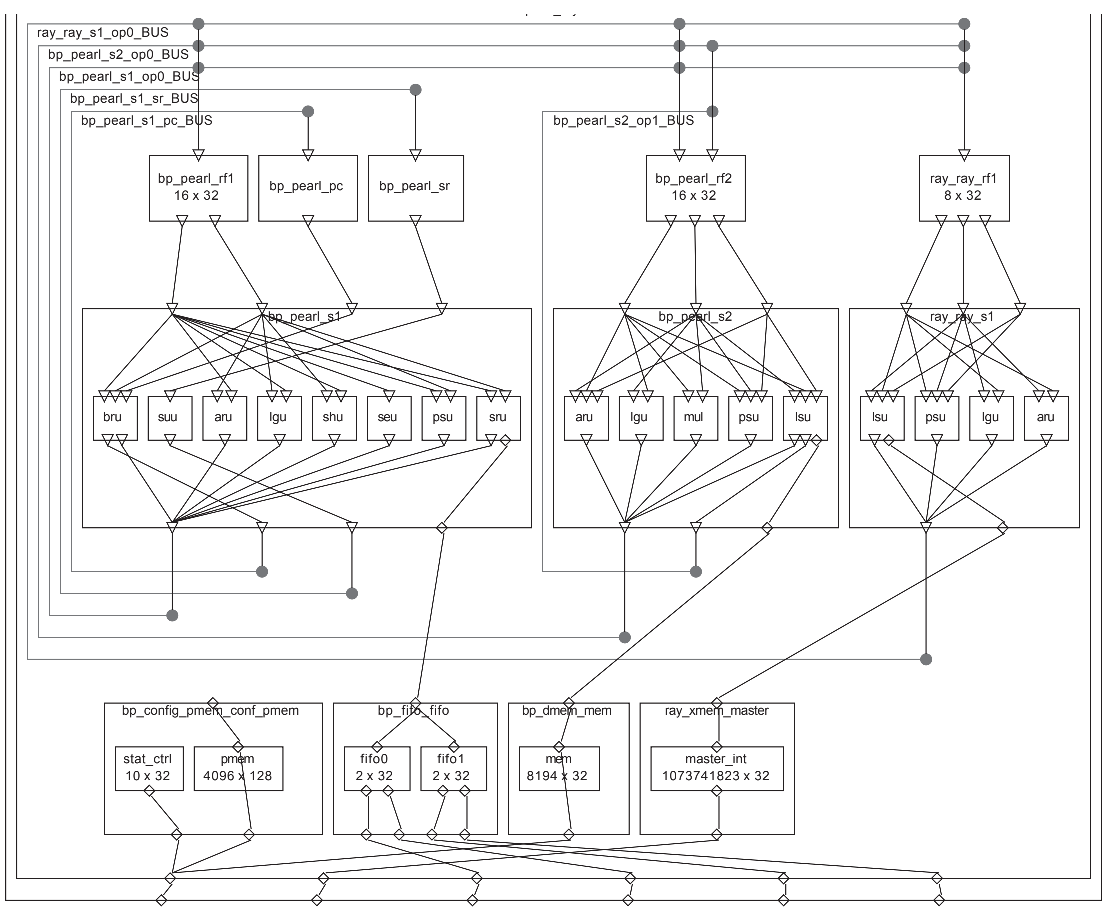

Figure 2.7: Schematic of the Pearl Ray Base Processor

2.2.1 The DesignWare Library

Synopsys provides an IP library called DesignWare which includes highly optimized RTL for arithmetic building blocks. Design Compiler can automatically determine when to use the DesignWare components and it can then efficiently synthesize them into gate-level implementation [35]. The DesignWare Building Block library is a collection of reusable intellectual property blocks, which are tightly integrated into the Synopsys synthesis environment.

By using the DesignWare Building Block IP, it is possible to significantly improve the productivity and performance improvement through pre-designed, pre-verified, highly optimized critical building blocks for high end ASICs. additionally, it allows a transparent high-level optimization of the performance during the synthesis. In this study, the Floating Point Arithmetic library from the DesignWare library has been used.

2.2.1.1 Floating Point Components

The Floating Point components comprise a library of functions used to synthesize floating point computational circuits in high end ASICs. The functions mainly deal with

Sign	**Exponent**	**Fraction**
(e+f)	(e+f-1)......f	(f-1)........0

Table 2.2: Floating Point Number Bit Positions [35].

arithmetic operations in floating point format, format conversions and floating point comparisons [35]. The main features of this library are the following:

- the format of the floating point numbers, which determines the precision of the number that they represent, is parameterizable. The user can select the precision based on the number of bits in the exponent and significant (or mantissa). The parameters cover all the IEEE formats;

- the accuracy conforms to the definitions in the IEEE 754 Floating Point standard for the basic arithmetic operations. Improved accuracy is obtained with multi-operand Floating Point components.

Floating point numbers are signed-magnitude numbers encoded with three unsigned integer fields: a sign bit, a biased exponent, and a fraction as shown in Table 2.2. All the floating point formats are defined by two integer parameters: *e* and *f*. The parameter *e* determines the number of biased exponent bits and *f* determines the number of normalized fraction bits. The most significant bit *(msb)(e+f)* corresponds to the sign of the floating point number. Therefore, a floating point format based number is always e+f+1 bits long. Finally, this definition is consistent with the IEEE 754 Standard. This option allows to implement any precision format choice in the design.

2.2.1.2 The IEEE 754 Floating Point Standard

The IEEE 754 standard is a technical standard for floating point computation which was first published in 1985 and established by IEEE. Before the establishment of this standard, floating point arithmetic was implemented in a very different number of ways in terms of, word sizes, precisions, rounding procedures and over/underflow behaviors. The IEEE standard defines floating point data, rounding schemes, operations, and exception handling (such as division by zero, overflow, etc.). This standard provides a method for computations with floating point numbers that will yield the same result whether the processing is done in hardware, software, or a combination of the two [1].

The floating point components in the DesignWare library cover more cases than the IEEE 754 floating point standard. For a given set of parameters, the components will use floating point formats that correspond to the ones defined in the standard (for example, single precision floating point format uses *f=23* and *e=8*). The use of denormalized numbers (denormals) and "Not A Number" (NaN) is controlled by a parameter (ieee_compliance). When this parameter is allowing denormals and NaNs, the components are completely compatible with the IEEE standard. When denormals and NaNs are not used, denormalized numbers are considered zeros and NaNs are considered infinities.

2.2.2 Synthesis

The synthesis is a design automation task in which RTL descriptions are transformed into a gate-level net list and then area, power and timing results are produced accordingly. A gate-level net list is basically a circuit implementation of the design made of library components (both combinational and sequential cells) available in the technology library and their interconnections. A synthesis tool takes an RTL hardware description, a standard cell library, and design constraints as input and produces a gate-level as output.

2.2.2.1 The Design Compiler

The Design Compiler tool is the core of the Synopsys synthesis products. The Design Compiler optimizes designs to provide the smallest and fastest logical representation of a given function. It includes tools that synthesize the HDL designs into optimized technology-dependent, gate-level designs. It supports a wide range of flat and hierarchical design styles and it can optimize both combinational and sequential designs for speed, area, and power [34].

A synthesis tool performs many optimizations, such as steps high-level RTL optimizations, arithmetic optimizations, technology independent optimizations, timing and area optimizations. In this study, the register retiming optimization technique is mainly used to optimize area and, more importantly, timing of the DesignWare IPs.

2.2.2.2 Retiming

Register retiming is a sequential optimization technique that moves the registers through the combinational logic gates in order to optimize the timing of design without changing the behavior of the circuit at the primary inputs and primary outputs [31]. With register retiming, the locations of the flip-flops in a sequential design can be automatically adjusted to match as close as possible the delays of the stages. This capability is particularly useful when some stages of a design exceed the timing goal, while other stages fall short. If no path exceeds the timing goals, retiming can be used to reduce the number of flip flops, whenever possible.

Purely combinational designs can also be retimed by introducing pipelining into the design. In this case, we need first to specify the desired number of pipeline stages and the preferred flip-flop from the target library. The appropriate number of registers is added at the outputs of the design. Then the registers are moved through the combinational logic to retime the design for optimal clock period and area.

During retiming, registers are moved forward or backward through the combinational logic of the design. Figure 2.8 illustrates an example of delay reduction through backward retiming of a register. In this example, Figure 2.8a shows the circuit before the retiming. Before register retiming, there are four levels of combinational logic and only one register at the endpoint of the critical path. However, in Figure 2.8b, the single register, has been replaced by four registers, moved back through two levels of logic and, therefore, the critical path now consists of two stages. The critical path delay in each stage is less than the critical path delay in the initial single stage design. As shown in this example,

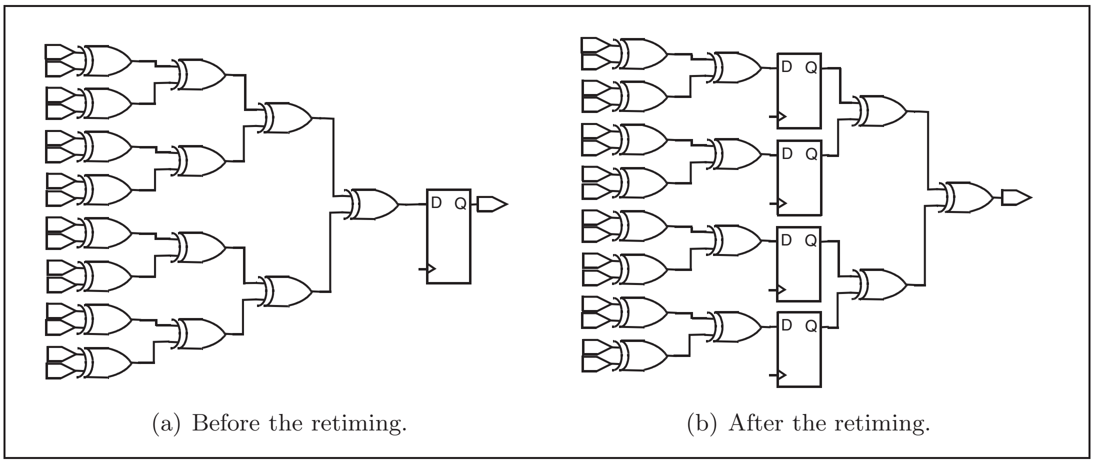

Figure 2.8: Backward retiming.

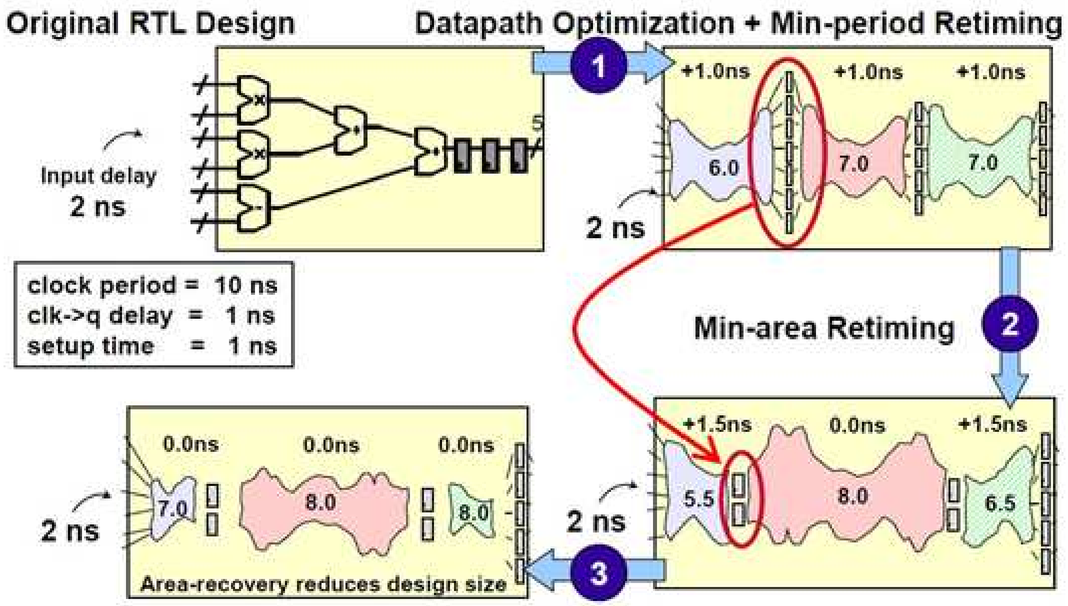

Figure 2.9: The Synopsys Retiming Process.

delay reduction through retiming can lead to an increase in the number of registers in the design. Anyhow, usually, this increase is small.

Besides reducing the delay of a design, register retiming presents another advantage: area optimization. After registers are moved through the design and established equal pipeline stages, if there is a positive timing slack, those combinational paths with positive slack can be optimized in terms of area by a min-area retiming step, as shown in Figure 2.9.

The example in Figure 2.9 shows the steps of the register retiming engine of Synopsys. The original RTL design has 3 registers in the output stage. First, these registers are moved such that a minimum period for each path is satisfied and the data paths are optimized at the same time. As it can be seen, the number of registers is increased after this step. After that, the next step is (min-area Retiming), where the number of registers required is decreased by moving them into the stage in which less number of fan in/out takes place. In the third step, the are is optimized. After step 2, some of the data paths have positive slack, which makes possible to decrease the area of the combinatorial logic. As a result, the area-recovery reduces the design size.

2.3 Conclusions

In this chapter, we presented an overview of the Silicon Hive processor development flow, its tool set and architecture. Innovative ultra long instruction word architecture from the Silicon Hive is the basis for this study. We had presented the out-of-box simple baseline core, the Pearl Ray core of Silicon Hive.

The Synopsys synthesis tool and optimization process, used for synthesizing the proposed processor and improve the timing of the floating point units, are explained in detail. Register retiming technique is described by giving examples. Additionally, the features of the Synopsys DesignWare building blocks, which are used to implement floating point functional units, are also given. We presented the floating point standard, IEEE 754 single precision, used in this study. Finally, we described DesignWare floating point IP blocks that can be configured for this floating point format.

In the next chapter, we will present the floating point extensions on the baseline Pearl Ray processor. Additionally, the timing optimizations of DesignWare blocks by introducing pipeline stages using register retiming will be presented.

3 Floating Point and Vector Extension

In this chapter, we will describe the floating point and vector extension of the Pearl Ray processor described in Chapter 2. At first, our focus will be on the floating point extension. The format of floating point numbers, targeted operations, and other design choices about floating point numbers will be covered. After that, in Section 3.2, we present the implementation details of the scalar floating point units and the DesignWare components that we used. Additionally, the timing improvement of the DesignWare floating point building blocks, by utilizing register retiming, is explained. In Section 3.3, the register retiming experiments on each floating point functional unit are explained by introducing the synthesis results. In Section 3.4, the methodology to introduce these functional units into the Pearl Ray processor is explained. Section 3.5 focuses on the vector extension. A new vector issue slot capable of executing both integer and floating point SIMD operations is presented. Details on the vector issue slot and the other SIMD components, like vector register file, vector memory and vector arithmetic units, are also presented. Finally, in Section 3.6 a final overview of extended processor is presented and synthesis results are given.

3.1 Floating Point Arithmetic Extension

One of the main goals of the work presented in this thesis is the introduction of a floating point arithmetic support in the Pearl Ray processor. So that, graphics algorithms can be efficiently executed. Graphics applications deal with real numbers that require a computational accuracy. Real numbers can be efficiently represented with the floating point format and the floating point format implementations take fewer cycles to execute than fixed point code (assuming, of course, that the fixed-point code offers similar precision).

3.1.1 Floating Point vs. Fixed Point

The choice on which format to use in the processor, a fixed point format or a floating point format, in a processor is usually related to the use of floating point operations in the targeted application domain. More specifically, the question boils down to *what is the degree of computational precision required by the target application*? Floating point applications require greater accuracy than what the fixed point can offer and the representation of data in the floating point format is more accurate than in fixed point format [11].

For application data sets that require real arithmetic, a greater precision and a wider dynamic range, the floating point format offers the best solution [11]. In this study, our aim is at extending the capabilities of a Silicon Hive processor, so to efficiently execute

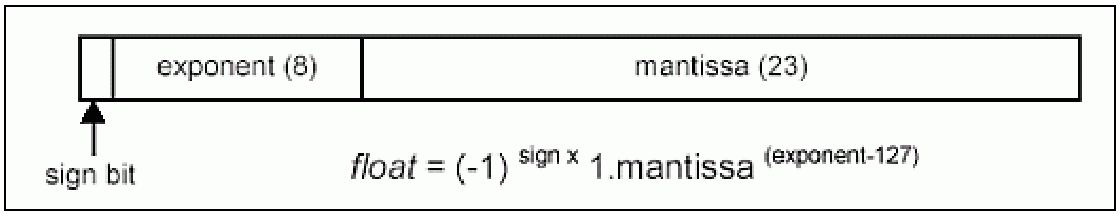

Figure 3.1: IEEE 754 single precision floating point format.

applications in the graphics domain. As the accuracy of floating point format plays a key role in graphics, we decide to adopt it.

3.1.2 Floating Point Format

Today's floating point processors are designed to handle two different data types: the single-precision floating point format and the double-precision floating point format. These two data types cover all the various data accuracies necessary for the markets, which is classically driven by digital signal processing. However, other floating point precision formats are available.

The floating point precision is an important design choice, which is always a trade off between performance and accuracy. In our work, we adopted a single precision format, as only very high end architectures, for scientific graphics applications that consumes huge amount of power, use a double precision floating point format. As mentioned in [11], the use of double-precision, 64-bit arithmetic results in a significant decrease in terms of performance. As a result, anyone planning to use the current generation of graphics processors for scientific computation must address the following question "*How important is single-precision compared to double-precision (64-bit) arithmetic for the application in use*? In the proposed architecture, we target performance rather than accuracy. As a result, single-precision format floating point units are implemented.

In our architecture, we have implemented floating point hardware that fully supports the IEEE 754 Single Precision floating point format. The baseline processor is designed for 32-bit integer arithmetic operations. It uses 32 bit data path with register files, load/store units and memory. As a result, the 32-bit single-precision floating point number representation can share the same data path, register files and memory with integer operations without a significant change in the control architecture.

The IEEE single precision floating point standard representation requires a 32-bit word. The first bit is the sign bit, the next 8 bits are the exponent bits, and the final 23 bits are the fraction bits (see Figure 3.1). The rounding mode can be changed by a hardware switch to round-to-zero, round-to-even and round-to-odd. There is no software control to change the rounding scheme. We have fixed the rounding scheme as "round to the nearest significant" to have the same simulation results with the simulator in which the same rounding scheme is utilized by default.

3.1.3 Targeted Operations

The floating point operations that we target are: addition, subtraction, multiplication, division, square root, absolute value, float-to-integer and integer-to-float conversions. According to the profiling in [41], floating point multiplication, addition and subtraction operations are the most commonly used operations in graphics algorithms. Therefore, we had introduce those operations in the first hand.

Although the square root and the division operations are very complex and costly operations that designers usually avoid to introduce, we aim at introducing these operations as well. First of all, square root and division operations are commonly used in our ray tracing algorithm, as explained in Chapter 4. Secondly, our aim is at having a generic floating point processor. Therefore, we cover as much as floating point operations as possible. If the design becomes very costly in terms of area, the processor can be easily reconfigured by removing these operations. Additionally, since we are using the DesignWare floating point IP library, we do not spend time for implementing full custom design of these units. Therefore, all floating point operations provided by the library can be easily introduced to the Silicon Hive base core. There is no difference in our architecture between integer and floating point load/store and pass operations. As a result, the same instructions already present in the basic processor are used.

Besides the operations just mentioned, more powerful application specific floating point operations could also be introduced as presented in [42]. For instance, there are many cross and dot product operations in graphics algorithms. A specific hardware for these kinds of operations would increase the overall performance of the processor. However, in this study, our aim at extending a Silicon Hive architecture for more generic operations. More specific floating point extensions are left as a future work.

3.2 Implementation of Floating Point Units

The implementation of hardware floating point data paths is a difficult task. The IEEE 754 standard makes it even more complicated due to different rounding schemes, normalization and de-normalization process, NaN values, etc. There are processors that provide IEEE 754 support through software extensions. However, we aim at having a fully IEEE compatible hardware. Since the implementation of the hardware for all floating point operations is not in the scope of this thesis, we have used predesigned blocks from the DesignWare library previously described in Section 2.2.1.

3.2.1 DesignWare Floating Point Units

In order to synthesize floating-point circuits, the designer must provide the detailed circuit description using Verilog or VHDL. Instead, a model-based design description using the DesignWare Library of Synopsys environment is used to provide a sufficient level of abstraction to allow the automated synthesis of floating-point data paths. As mentioned before, the design of hardware for all floating point operations would be very time consuming and not in the scope of this thesis. As a result, the use of predesigned and pre-verified floating point hardware can considerably reduce the development time. The

DesignWare library provides the floating point hardware with different parameterizable configurations. In our study, we tuned the parameters so that the floating point units can support the 32-bit IEEE 754 single precision format.

The following DesignWare Building Blocks from the DesignWare library have been used: *dw_fp_addsub, dw_fp_mul, dw_fp_sqrt, dw_fp_div, dw_fp2int, dw_int2fp and dw_fp_cmp.* The blocks are described in detail in the following sections.

3.2.2 Improving Timing of DesignWare Units

In this section, we describe the method to improve the timing of DesignWare functional units. Our target frequency is 333 MHz (3 ns clock period). As a standard cell library, we use the TSMC 40 Low Power Technology. The synthesis of complex floating point units at 333 MHz in TSMC 40 LP technology results in a negative slack, due to the long critical path delays exceeding the clock period constraints. This happens because the floating point building blocks of the DesignWare library are fully combinatorial logic, which means there are huge combinational paths from the input to the output. As a result, the achievement of a 3 ns clock period constraints is not feasible especially for more complex operations, such as the division and the square root.

Our approach is to improve the timing of these units and, at the same time , to satisfy the clock period constraints. Unfortunately, we do not have access to the source RTL code of the DesignWare units. As a result, the manual insertion of pipeline stages is not possible. However, Synopsys claims that full combinational DesignWare units are designed in such a way that the register retiming option invoked by the *optimize_registers* command at Synopsys Design Compiler can be effectively used to improve the timing of these units [35]. This command allows the insertion of pipeline stages for combinational DesignWare Building Blocks during the synthesis process.

For the pipelining of combinational data paths, the pipeline registers have to be placed at inputs or outputs of the RTL implementation. Retiming, described in detail in the previous chapter, is utilized to move registers to the optimal locations. As a result, the timing of these units is improved by having a pipeline design. As we started with a full combinational design, the throughput of the pipelined units is 1. In other words, it can produce output at every cycle. The latency will be equal to the number of registers included.

The number of registers required depends on how big is the delay of the starting combinational design. More complex operations will require more register stages. In order to determine how many registers are required for each floating point component, we synthesized them for different number of register stages, starting from zero. The target is to find out what is the smallest number of registers that satisfies the timing constraints. Additionally, by exploring the area improvement obtained by the retiming, we may balance latency and the area by trading off the number of registers, i.e. the latency, and the area. The number of registers is increased till the are cannot be further optimized.

3.3 Floating Point Units

3.3.1 Faru Unit

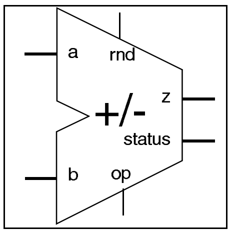

Figure 3.2: The *dw_fp_addsub* unit employed in the *Faru* unit.

The *Faru* functional unit is responsible for the floating point addition and subtraction operations. The *dw_fp_addsub* (see Figure 3.2) unit is used from the DesignWare library. Two 32 bit IEEE 754 compatible single precision floating point numbers can be added or subtracted in this functional unit. The input, *"op"*, determines subtraction or addition of the number depending on the operation code. The rounding scheme is round to the nearest significand.Status output of the DesignWare unit is left unconnected, as there is no flag support for floating point exceptional cases in our processor. These exceptional cases are handled by the compiler. Finally, the result is connected to 32 bit register file.

Number of stages	**com** (μm^2)	**seq** (um^2)	**Total Area** (μm^2)	**WNS (ns)**
0	6257	0	6257	-1.505
1	3471	481	3952	**MET**
2	2988	659	3647	**MET**
3	3009	869	3879	**MET**

Table 3.1: Area and worst case negative slack results for the Faru unit with respect to the number of register stages.

The area results after the synthesis are presented in Table 3.1. The process technology is TSMC 40 LP and the target clock period is 3 ns. The table also presents the corresponding Worst Case Negative slack values (WNS) reported by the DC compiler. In the table, MET means that the timing is satisfied for that design. The positive slack results are not given since they are very close to zero. The synthesis tool uses positive slack for area optimization and uses bigger logic for the combinational paths that has positive slack. The first row shows the case when no registers are included which turns into a -1.505ns negative slack on the critical path for the initial implementation. The clock period is 3 ns. By summing up the two values, the overall latency of this combinatorial addsub unit becomes 4.505 ns. This means that the timing of the unit must be improved by introducing register stages and utilizing the retiming option as it explained before..

In Table 3.1, the first column shows the number of register stages introduced to have an optimum design. We conclude that the introduction of 1 register stage is enough to meet the timing constraints for this unit. There is no Worst Case negative slack reported by the tool in case of 1 stage. This result is somehow expected from the previous theoretical WNS calculation. If we assume that the registers will be moved into the middle of the critical path, the new critical path delay will be around 2.250ns (4.505ns divided by 2) plus the delay coming from the registers. As a result, there is

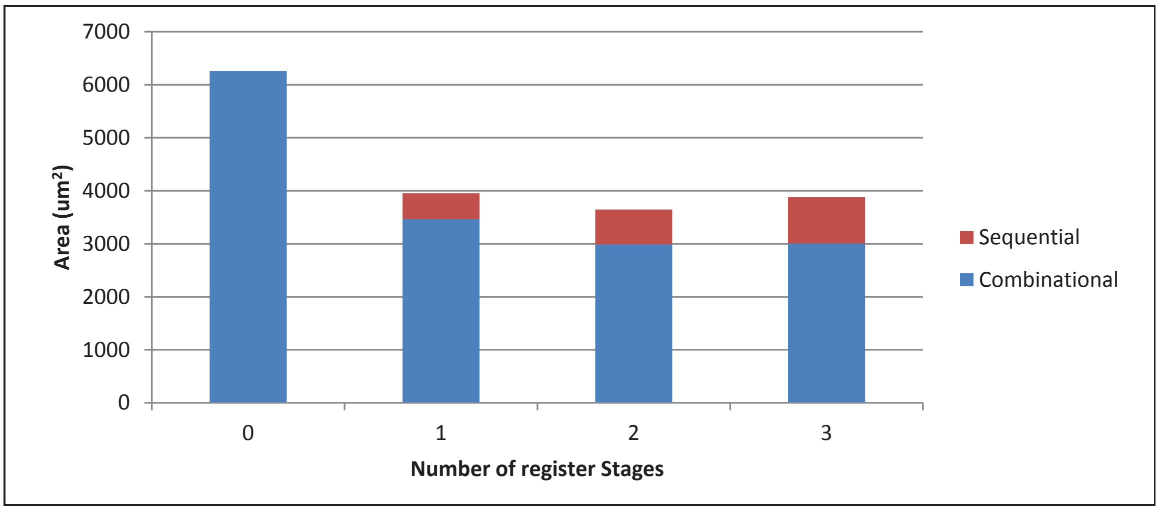

Figure 3.3: Combinational and sequential area results for the *Faru* unit after retiming.

enough positive slack for a clock period of 3ns.

Consequently, this positive slack has been used for area optimization by the DC compiler. In Figure 3.3, we can observe that the area improves around 37 % when 1 register stage is introduced . As the graph depicts, this improvement is due to a combinational area optimization.

Although the required timing constraints are met with 1 stage, we kept introducing more register stages in order to observe if we can achieve a better area result that may lead us to a better trade off between latency and area. However, we observed that the improvement is limited, if more register stages are introduced, as depicted in Figure 3.3. In conclusion, the *dw_fp_addsub* unit extended with 1 pipeline stage has been chosen for the implementation of the Faru unit in our processor. As a result, the execution latency of the floating point addition and subtraction operations is 1 clock cycles.

3.3.2 Fmul Unit

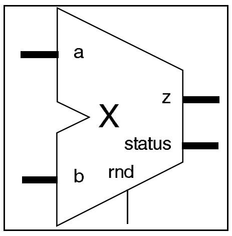

Figure 3.4: The *dw_fp_mul* unit employed in the *Fmul* unit.

The *Fmul* functional unit is responsible for the floating point multiplication operation. The *dw_fp_mult* building block from the DesignWare floating point library is used in order to realize the implementation of the *Fmul* unit. It can multiply two 32-bit single precision floating point numbers and gives the result again in 32-bit single precision format in the IEEE 754 standard. The result is rounded to the nearest value. *dw_fp_mult* has also a status output, which reports exceptional cases. However, the status output is left unconnected since we do not have a floating point flag for such an exceptional cases yet in our architecture.

As for the *Faru* unit, the *Fmul* unit is synthesized using the same approach at 333MHz clock frequency with TSMC 40 LP technology. Table 3.2 shows the results of the experiments. When the floating point multiplier is synthesized

Number of stages	**com** (μm^2)	**seq** (μm^2)	**Total Area** (μm^2)	**WNS (ns)**
0	10927	-	10927	-1.169
1	6111	763	6875	**MET**
2	5656	844	6501	**MET**
3	5570	1057	6627	**MET**

Table 3.2: Area and worst case negative slack results for the *Fmul* unit with respect to the number of register stages.

without including any register stages (when number of stages is 0), it is observed that there is 1.169 ns worst case negative slack in the design. By the preview analysis for *Faru*, we can expect that 1 register stage can be enough for Fmul unit too. We include pipeline stages and perform the retiming in order to improve the timing to satisfy 3 ns clock period constraint. As before, 1 register stage is enough to satisfy the timing constraints (see Table 3.2).

As shown in Figure 3.5, there is a significant improvement in the area after introducing 1 pipeline stage. The improvement is around 38 %. A similar analysis to do one done for the *Faru* unit can be done for the Fmul unit as well. As before, we introduced also 2 and 3 levels of register stages, in order to further explore possible area optimizations. However, the area does not change much compared to 1 register stage (see Figure 3.5). Consequently, 1 stage is chosen for the *Fmul* unit. As a consequence, the final execution latency for the multiplication floating point unit is 1 clock cycle.

As a matter of fact, in the literature, the floating point addition operation is considered as a more complex operation than the multiplication [22], as the significands are represented in signed value format but not in 2's complement. However, our results contradicts this statement. According to our results, multiplication operation occupies much more area than addition. While the area of the *dw_fp_addsub* unit is 6257 μm^2 (see Table 3.1), the *dw_fp_mult* unit's area is 10927 μm^2 (see Table 3.2).

3.3.3 Fdiv Unit

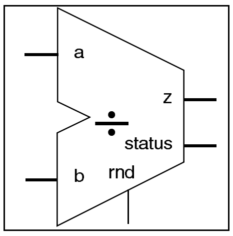

Figure 3.6: The *dw_fp_div* unit employed in the *Fdiv* unit.

The *Fdiv* functional unit is responsible for the division operation. The *dw_fp_div* building block from the DesignWare library is used to build the *Fdiv* unit. The parameters are chosen so to support the IEEE single precision floating point format as explained before. Like the *dw_fp_mul* block, the *dw_fp_div* building block has a 8-bit status output. It reports overflows, division by zero, NaN values and other exceptional cases. However, it is left unconnected, as there is no floating point number flag implemented in our processor. The compiler is mainly responsible to handle such exceptional cases. For example it reports the division by zero. For this unit, rounding scheme is chosen as round-to-zero. Our software simulation and RTL simulation result comparisons show that we should use this rounding scheme to

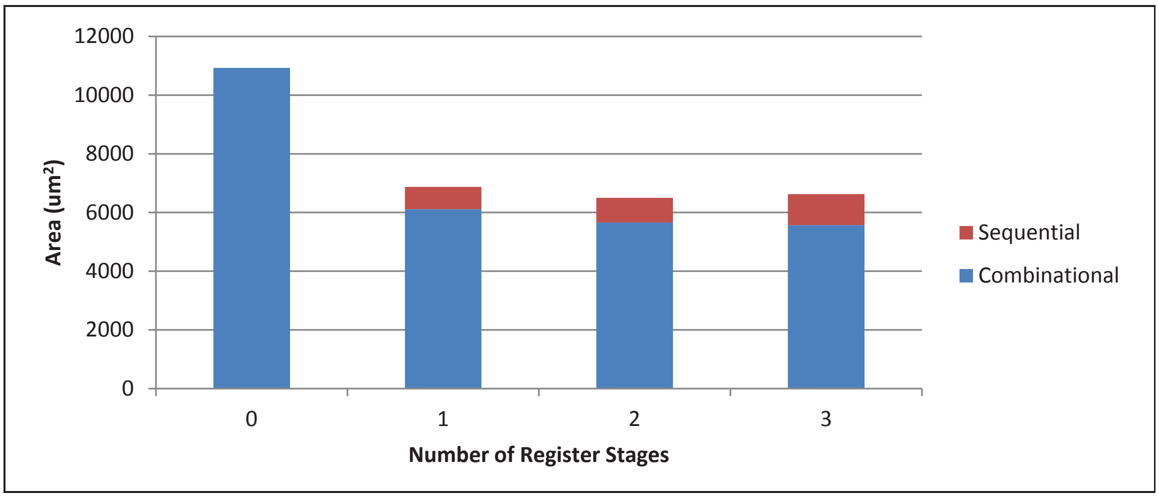

Figure 3.5: Combinational and sequential area results for the *Fmul* unit after retiming.

have matching results.

The floating point division operation is a more complicated operation than both the addition and the multiplication. As a result, more pipeline stages should be needed to satisfy the timing constraints for this unit. Again, we started with the original design without introducing any register stage and without using any register retiming. The synthesis results in Table 3.3 show a worst case negative slack of -7.557ns in that case. If we consider also the clock period, the overall latency becomes around 10.55ns. In ideal case, if 3 registers could be moved to establish a perfect 3 level pipeline stage, than the critical path delay would be approximately 2.64ns (here we assume no extra delay resulted from the registers.). If we approximate delay for each register stage to be 200ps, it seems improbable to achieve the timing constraint.

Number of stages	**com** (μm^2)	**seq** (μm^2)	**Total Area** (μm^2)	**WNS (ns)**
0	19446	0	19446	-7.557
1	19380	1174	20554	-2.703
2	19279	2929	22208	-1.224
3	18430	3127	21557	-0.378
4	14107	4211	18318	-0.080
5	12637	3374	16011	**MET**
6	12722	3595	16318	**MET**
7	12666	3790	16457	**MET**
8	12681	3993	16675	**MET**

Table 3.3: Area and worst case negative slack results for the *Fdiv* unit with respect to the number of register stages.

As the results in Table 3.3 show, 5 registers are required to achieve the target fre-

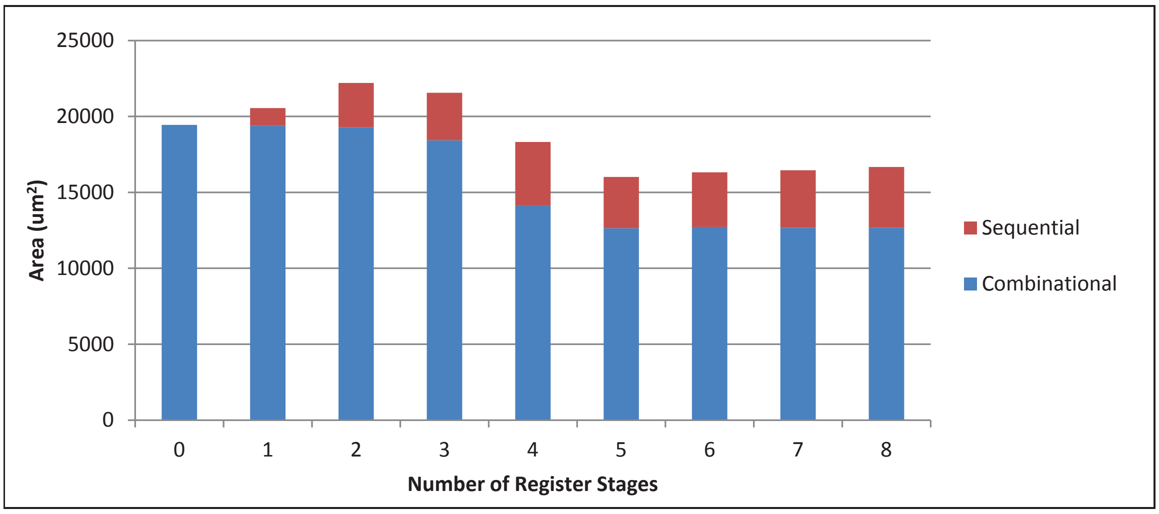

Figure 3.7: Combinational and sequential area results of *Fdiv* unit after retiming.

quency of 333 MHz. It could be claimed that 4 registers are also enough, since the worst case negative slack is quite close to zero, namely 0.08ns. However, the area of 5 pipeline stages is 13% smaller. Moreover, 0.08ns is very critical. Therefore, we used 5 register stages implementation in our processor for *dw_fp_div* block. Consequently, the latency of the floating point division is 5 clock cycles.

As shown in Figure 3.7, for a small number of the register stages (1 or 2) or no register stage at all, where high worst case negative slack occurs, the inclusion of more registers results in an increase in the total area. The reason is that, as the initial WNS is very high, the positive slack achieved for individual paths is limited. If there is no positive slack, the combinational area cannot be further optimized. Additionally, the deployment of the new register stages increases the sequential area as well. As a result, the overall area increases, as shown in the graph in Figure 3.7. While WNS gets closer to zero, the area starts to improve. The reason is that there is more playground for the DC compiler's area optimization phase, due to more positive slack achieved on the individual combinational paths.

3.3.4 Fsqrt Unit

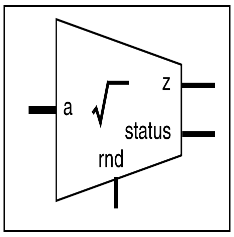

Figure 3.8: The *dw_fp_sqrt* unit employed in the *Fsqrt* unit.

The floating point square root operation is implemented in the *Fsqrt* functional unit. The *dw_fp_sqrt* is the DesignWare building block is used to build this unit. It supports 32-bit single precision floating point format and it is Fully IEEE 754 compliant, as the other units. Round to nearest significant is used as a rounding scheme. The input and the output are connected to a 32-bit register file.

The *dw_fp_sqrt* building block is synthesized using the same constraints as the previous units. The synthesis result of the original implementation are shown in Table 3.4. It can be concluded, from the table, that the design is far away from satisfying the timing constraints. This is something expected because the floating point square root design is a very complex operation and the hardware is more complicated. The worst case negative slack of the original unit is -10.288ns at 333 MHz. The overall latency is 13.288ns. This gives a hint that more register stages are required to achieve the target frequency. In theory, 4 stages do not seem feasible. Under an ideal pipelining conditions, the delay of the critical path would be 2.656ns, without considering the clock-to-Q delay, hold and setup time delays of the flip flops. Under these circumstances, the most possible number of registers that meets the timing requirements is probably 5.

Number of stages	**com** (μm^2)	**seq** (μm^2)	**Total Area** (μm^2)	**WNS (ns)**
0	11013	0	11013	-10.288
1	12024	504	12528	-4.013
2	12174	1088	13262	-2.177
3	11212	1538	12751	-0.871
4	10754	2100	12855	-0.306
5	8089	2515	10604	**MET**
6	6258	2813	9072	**MET**
7	5350	2813	8163	**MET**
8	5483	2921	8405	**MET**

Table 3.4: Area and worst case negative slack results for the *Fsqrt* unit with respect to the number of register stages.

The results (see Table 3.4) show that the previous assumption is correct. There is a negative slack until the 5th pipeline stage is included into the *dw_fp_sqrt* unit. When more registers are included, the total area continues to decrease as it can be observed in the graph in Figure 3.9, as combinational logic can be optimized more exploiting the achieved positive slack by the retiming, as explained before. However, the inclusion of even more registers, for example 8 registers, does not bring to an area decrease. This happens because the area optimization reaches its limits.

For our processor, we decided to introduce 7 register stages for the implementation of the *Fsqrt* functional unit, due to the small area. The area improvement with 7 stages respect to 5 stages is 25%. Considering the fact that, the latency increases from 5 to 7, which is 40%, this decision could be criticized at first. However, there are two reasons that lead us to chose 7 pipeline stages. First of all, the square root unit is the largest floating point unit. As a result, a 25% increase is a lot compared to the other units. For instance, 25% of the Fsqrt unit is equal to half of the *Faru* unit. Secondly, as we stated earlier, the square root operation is not a very common operation like the addition or the multiplication. Therefore, it would not affect the overall performance of our processor, when it runs the targeted graphics algorithms. Additionally, since our floating point unit is pipelined, the throughput will always be 1 and the latency can be hidden by efficiently scheduling the instructions. Anyhow, a faster implementation could be preferred, if the square root operation is a very critical operation for the target application domain.

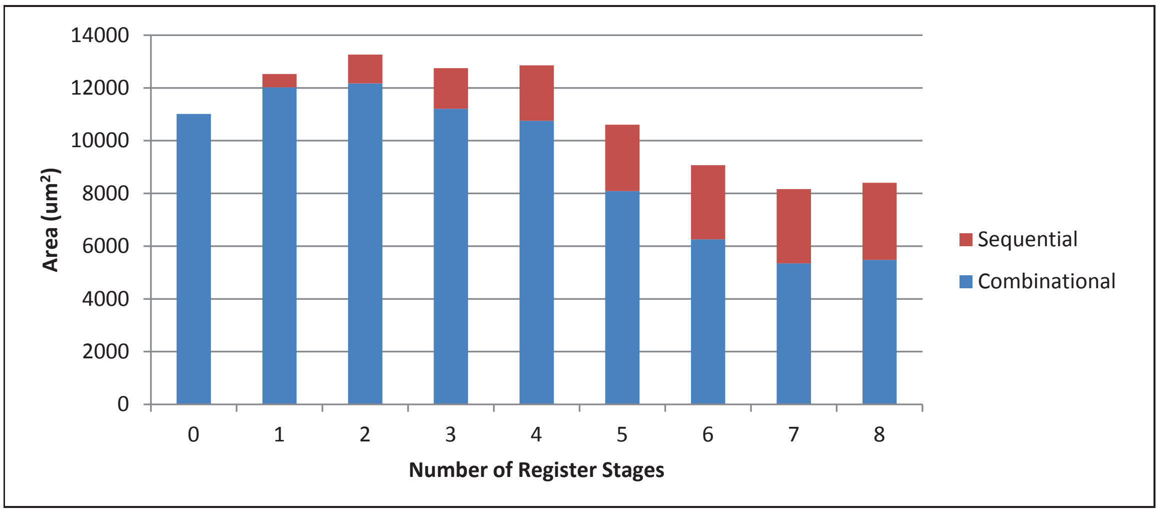

Figure 3.9: Combinational and sequential area results for the *Fsqrt* unit after retiming.

3.3.5 Fcmp Unit

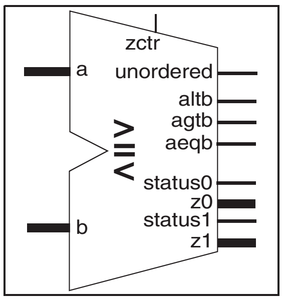

Figure 3.10: The *dw_fp_cmp* unit employed in the *Fcmp* unit.

The *Fcmp* unit is the floating point comparator unit. It compares two floating point numbers. Depending on the operation code, the output is chosen by the multiplexing network for the corresponding operation. The results are written to the same 32-bit register file, even though these might be one bit boolean results in the case of comparison operations. The accuracy conforms to the IEEE 754 floating point standard. Status flags are unused in our implementation. Figure 3.10 shows the *dw_fp_cmp* unit taken from the DesignWare library and Table 3.5 gives details about the inputs and the outputs.

Pin Name	Width	Function
a	32	Floating-point number
b	32	Floating-point number
altb	1	High when a is less than b
agtb	1	High when a is greater than b
aeqb	1	High when a is equal to b
unordered	1	High when one of the inputs is NaN and ieee_compliance = 1
z0	32	Min(a,b) when zctr=0 or open, otherwise, Max(a,b)
z1	32	Max(a,b) when zctr=0 or open, otherwise, Min(a,b)

Table 3.5: The *dw_fp_cmp* building block pin descriptions.

The floating point comparison unit has a small design compared to the other floating point units. As Table 3.6, shows there is no need to include any register stages to do retiming, as the design itself can satisfy the timing constraints. By including more register stages, the are does not improve at all. Anyway, the area is much smaller than the other units. In conclusion, the unit has been used without including any register stage in the processor. The resulting latency is 0 cycles.

Number of stages	**com (μm^2)**	**seq (μm^2)**	**Total Area (μm^2)**	**WNS (ns)**
0	423	-	423	**MET**
1	455	321	876	**MET**
2	458	675	1134	**MET**

Table 3.6: Area and worst case negative slack for the *Fcmp* unit with respect to number of register stages.

3.3.6 Fxalu Unit

Simple operations like the floating point absolute value, get sign, negate and float to integer conversions are introduced in the *Fxalu* functional unit. The implementations of *fabs*, *fnegate* and *fsign* operations are very straightforward. For those operations, we have not use pre-build blocks and the implementation has been done manually. However, for the *fp2int* and *int2fp* operations pre-build blocks are used. The list of operations in the *Fxalu* unit is shown in Table 3.7. The area of this unit is around 100 μm^2 which is negligible compared to the other floating point units. The latency of all these operations is 0.

Operation Name	**Output Width**	**Function**
fabs	32	Gets the absolute value of a floating-point number
fnegate	32	Negates a floating point number
fsign	1	Gets the sign bit of the floating point number
fp2int	32	Converts SP floating point number to integer value
i2fp	32	Converts integer value to SP floating point format

Table 3.7: List of operations in the *Fxalu* unit.

3.4 Method of Introducing FPUs into Base Processor

In this section, we explain how to introduce the floating point units into the Silicon Hive baseline processor, the Pearl Ray VLIW processor, described in the previous chapter. The configuration of a processor using the Silicon Hive core generation flow is pretty straightforward. The TIM description language allows us to define new operations, to introduce new functional units or issue-slots easily. As depicted in Figure 2.3, the TIM machine description lies at the heart of the core generation. The architecture description, compilation and simulation process starts with TIM. In the following, the process of introducing new floating point functionality into the baseline processor step-by-step is presented.

3.4.1 Operation Semantics

First of all, the floating point operation semantics need to be defined. This is required for the simulation of the processor. The floating point operations are defined in a C-like description. The simulator gets the operation semantics through the TIM description. In this way, the simulator is informed on how it should treat the new floating point operations. Additionally, a specific operation code is assigned for each operation automatically by the flow.

Besides operation definitions, another important issue is how to handle the scheduling of the floating point operations. This can be done statically by a compiler, similar to the way in which operations from static code schedules are scheduled and issued on VLIW processors [10], or dynamically, similar to the way in which out of-order processors issue instructions, when their operands become available [30] . In our processor, like other VLIW architectures, there is no hardware scheduler that resolves data dependencies or schedules the instructions. The compiler is responsible for the analysis and to resolve the data dependencies. Instead of doing the scheduling in hardware, where the scheduling logic is power hungry, the scheduling for a Silicon Hive architecture is done by the compiler. The operations are scheduled in a way that optimizes both the resource utilization and the pipeline depth to achieve maximum execution throughput by the compiler. The compiler must be informed about the latencies of each operation pipeline. As a result, the latencies of the newly defined floating point units must be introduced to the compiler. In Section 3.3, we have presented the pipeline depth (latency) of each operation. Table 3.8 shows an overview of the floating point operations and the corresponding latency values.

Operation	Latency
Addition/Subtraction	1
Multiplication	1
Division	5
Square root	7
Comparisons	0
Other fp oeprations	0

Table 3.8: Floating point operation latencies.

The TIM language allows to inform the compiler by presenting the timing of operations. This information is given by the cycle time parameter in the TIM language. The

Cycle Time information of the operations are presented according to the table. Hence, the compiler knows when the result will be ready. As a result, the operations can be effectively and efficiently scheduled by the compiler.

3.4.2 Functional Unit Definitions

Functional units group the operations, which are comparable from a hardware point of view. For instance, the addition and the subtraction operations are implemented in the same functional unit. Set of operations can be performed by one functional unit. All FUs in our architecture are fully pipelined so that one instruction can be issued at each cycle even when the latency of that instruction is more than one cycle. Different implementations may lead to different latencies, which can be specified in the operation description (see Section 3.4.1) and it is supported by the compiler.

We have already presented the floating point functional units and operations involved in each of them. The implemented Floating Point Units (functional units) are deployed into the issue slot 2 (bp_pearl_s2) of the baseline processor (Figure 3.11). The baseline architecture is designed for a 32-bit data path. Similarly, our floating point units operate on a 32-bit format. Therefore, 32-bit integer and floating point functional units can share the same data path, memory, register files, load/store unit and pass unit in the same issue slot.

The VLIW feature of the Silicon Hive baseline processor makes possible and easy to introduce these floating point units in multi issue slots. Actually, the inclusion of all the FPUs into the same issue slot is not very good design choice. The use of multi issue slots will immediately increase the theoretical peak of floating point operations per second of the architecture, as multiple floating point operations can be issued in parallel. However, at first, the main target of this study is at having a generic floating point processor, which is able to execute graphics applications. Additionally, the decision on how to distribute FPUs into multi issue slots depends on the application. As a result, the application has to be well explored in order to find an efficient distribution of the FPUs. The high level of reconfigurability in the Silicon Hive cores gives space to improve by exploiting the achievable ILP in the algorithm and by correlating the operations that can execute in parallel for that code. Such an approach is presented in Section 4.6, during the analysis of the architectural refinements, where we first examine the Ray Tracing algorithm as a case example and then tune the architecture by distributing the floating point units into different issue slots to efficiently execute the Ray Tracing algorithm.

3.4.3 Register File Connections

The register files are part of the important storage elements in our VLIW architecture. In the baseline processor we have 3 register files. The read ports of those registers are directly connected to FUs inside the corresponding issue slots (see Figure 3.11). Normally, VLIW architectures need multiple port register files for high performance. On the other hand, keeping the register files as simple as possible is important in terms of area and power. Simpler and smaller register files are not only faster but also consumes less power. Therefore, in our architecture, we use the same register files for integer and floating point format storage. The 32-bit data registers of the baseline processor can

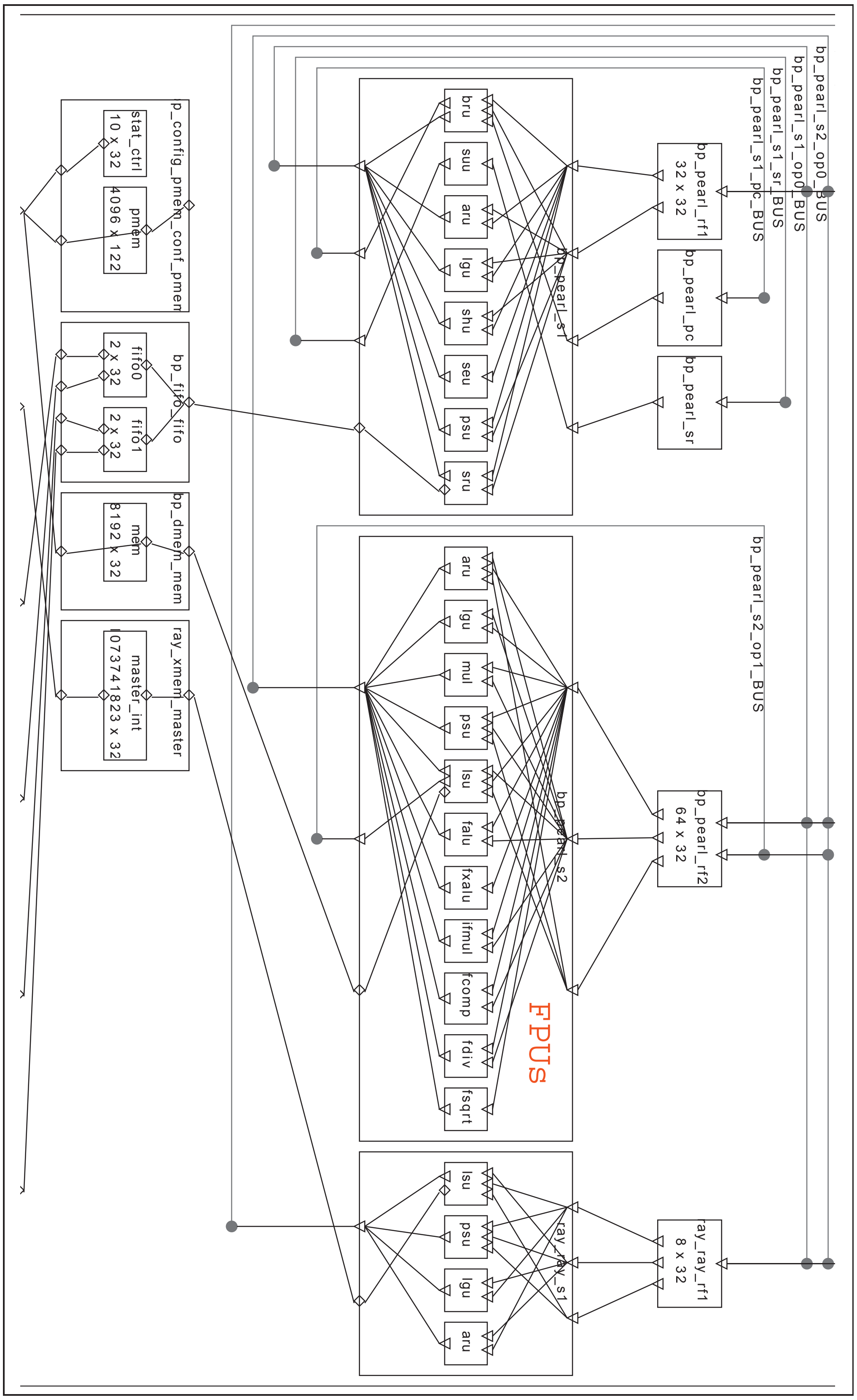

Figure 3.11: The Pearl Ray baseline processor with the floating point unit extension.

support both. For each issue slot there is one register file deployed. The write ports of the register files are connected to each other through the bus. However, the read ports are only connected to the corresponding issue slot. Each register file can be only read by the issue slot it belongs to. When a data requires to be transferred from one register file to another, it should pass through the pass unit in that issue slot. A pass unit can read from a register file, which is in the same issue slot with, and write to other register files through the write bus.

Register file 1 lies in issue slot 1. It (bp_pearl_rf1) has 1 write and 2 read ports. 2 read ports are adequate since in that issue slot there is no FU that requires 3 inputs from the register file. It is only connected to integer arithmetic units. The length of the rf1 register is 32 registers, which is the same with initial baseline processor. Register File 2 (bp_pearl_rf2) has 2 write and 3 read ports. More write ports are included, considering the fact that it will provide data to many FPUs. Additionally, the length of the register file 2 is doubled to 64 registers, after the inclusion of floating point units, assuming that more registers would be required for better performance. Lastly, there is ray_rf1, which consists of 8 32-bit registers and has 1 write port and 3 read ports.

3.5 Vector (SIMD) Extension

The target processor should have a sufficiently high floating point capability, as the very high performance required by the target application domain and a great number of floating point operations are executed, . This can be achieved by taking advantage of the data level parallelism existing in graphics applications. Most of the traditional techniques actually focus on exploiting more Instruction Level Parallelism (ILP) [26]. We already explained the importance of the VLIW property, in order to exploit ILP in the applications. However, in this section, our focus will be more on the data level parallelism.

The data Level Parallelism, which can also be addressed to Single Instruction Multiple Data (SIMD) paradigm refers to a single instruction that can specify a large number of operations to be performed on independent data words. This method traditionally is exploited in the super computing domain by vector [28] and array [27] processors. However, it has also been shown that SIMD instructions can be efficiently deployed in VLIW processors [29]. In order to exploit the data level parallelism in graphics algorithms, the SIMD version of the floating point operations and the other in general operations (RISC-like) are implemented on the proposed architecture.

In the rest of this section, the SIMD operation extensions will be described. Both the integer and the floating point vector instructions are introduced. The implementation of vector arithmetic units, load/store unit, register files and vector memory is described. The vector extension allow our VLIW processor to perform 8-way SIMD operations. This support is provided by including a vector issue slot. As a result, the final architecture becomes a hybrid VLIW/SIMD processor.

3.5.1 Vector Arithmetic Functional Units

Arithmetic operations are defined for vectors in terms of the corresponding scalar operations. SIMD versions of floating point units are the *Fvaru* unit (addition/subtraction), the *Fvdiv* unit(division), the *Fvmul* unit(multiplication), the *Fvsqrt* unit(square root), the *Fvxalu* unit (absolute value, get sign, negate), and the *Fvcmp* unit (comparison), which are all supported by our architecture. Although floating point vector operations are our primary target, some operations on integers and logical operations are also included. These are defined in the *Varu* functional unit. The addition, the subtraction, and the comparison operations are also introduced to operate on integers. Integer multiplication and division are omitted, since they are more complex and less commonly used operations. The floating point division and multiplication are used for integer arithmetic as well by first converting the integers to the floating point numbers.

FU Name	**Operation**	**Type**	**Operand1**	**Operand2**	**Result**	**Latency**
varu:	add/sub	int	vector	vector	vector	0
	add_c/sub_c	int	vector	scalar	vector	0
	compare	int	vector	vector	flag	0
	compare_c	int	vector	scalar	flag	0
fvaru:	add/sub	fp	vector	vector	vector	1
	add_c/sub_c	fp	vector	scalar	vector	1
fvmul:	multiply	fp	vector	vector	vector	1
	multiply_c	fp	vector	scalar	vector	1
fvdiv:	divide	fp	vector	vector	vector	5
	divide_c	fp	vector	scalar	vector	5
fvsqrt:	sqrt	fp	vector	vector	vector	5
	square root_c	fp	vector	scalar	vector	5
fvcomp:	compare	fp	vector	vector	flag	0
	compare_c	fp	vector	scalar	flag	0
fvxalu:	absolute	fp	vector	vector	vector	0
	get_sign	fp	vector	vector	vector	0
	negate	fp	vector	scalar	vector	0

Table 3.9: Vector Operations.

The sizes of integer and floating point vector data are the same, 32 bits. Hence, both integer and floating point vector arithmetic units can be included in the same issue slot and share the same data path and vector register files. Vector by scalar operations are also supported by vector arithmetic units. For example, a floating point vector data can be multiplied by a scalar floating point number and the result can be written to a vector register file. Table 3.9 shows the vector arithmetic operations supported by our processor, their operand and result types, and the latency values.

For the comparisons of two floating point vectors, there is the *Fvcomp* unit. The *Fvcomp* takes two floating point vectors as input and produces and 8-bit flag vector.

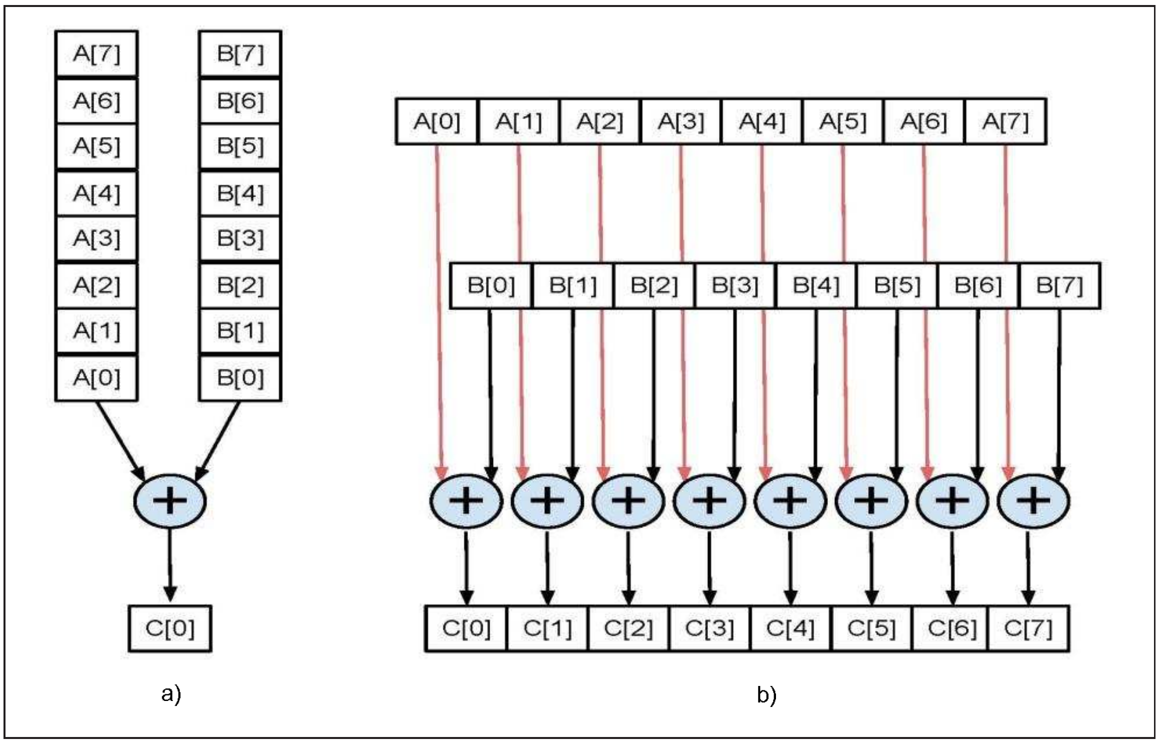
A[7]
A[6]
A[5]
A[4]
A[3]
A[2]
A[1]
A[0]
B[7]
B[6]
B[5]
B[4]
B[3]
B[2]
B[1]
B[0]
C[0]
a)
A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7]
B[0] B[1] B[2] B[3] B[4] B[5] B[6] B[7]
C[0] C[1] C[2] C[3] C[4] C[5] C[6] C[7]
b)

Figure 3.12: Vector Architecture Types: a) Single pipelined unit. b) Multiple pipelined units.

The use of flag vector will be described while explaining the mux operations in Section 3.11. There is also a flag register file to store resulting comparison flags.

In general, there are two common techniques to implement vector arithmetic units as depicted in Figure 3.12. The first one is a single pipeline architecture, where each element of a vector is driven into a single pipelined execution unit. The second technique is via multiple parallel pipelined execution units. Figure 3.12 shows the two different techniques. The approach in Figure 3.12b is followed to implement the vector arithmetic units. Basically, each floating point vector arithmetic unit is eight complete replicas of the pipelined scalar arithmetic functional units mentioned in Section 3.3. This implementation is very straightforward compared to other techniques. The advantage of this approach is that the processor throughput increases without a significant change in the control unit complexity [14]. However, it increases the area since the total area of a vector functional unit is 8 times of the one of a scalar functional unit for the same arithmetic operation. However, our primary target is to increase the floating point performance of our processor, since a lot of floating point operations are executed in the graphic algorithms we target. As a result, we trade area for performance. The hardware implementation of the *Varu* functional unit is automatically generated by the Silicon Hive core generation tools.

3.5.2 Vector Load/Store Unit

The Vector Load/Store Unit (VLSU) moves the vector data between vector register files and vector memory. The vector load/store unit in our processor is quite simple. In fact, there is not much difference between a standard and a load/store units in our architecture. The only difference is the size of the operands. Since it loads and stores 8 vector elements, the read and wrote data are 256-bit wide. Similarly, vectors can consist of integers or floating point numbers. There are no differences from a load/store units point of view between the two formats.

Operations	**Operand 1**	**Operand 2**	**Operand 3**	**Result**	**Latency**
load	Vector memory	Base address	-	vector	1
load offset	Vector memory	Base address	offset	vector	1
store	Vector register	Base address	-	memory	1
store offset	Vector register	Base address	offset	memory	1

Table 3.10: Vector load/store operations.

Basically, the load/store unit directly loads or stores the entire 256 bit wide vector data at once. Therefore, unlike the traditional vector load/store units, in which vector elements are loaded one by one in the pipeline, the latency of the load/store unit does not depend on the vector length. However, this type of load/store architecture necessitates vector memories. As a result, we introduced a vector memory in our processor. Our vector memory is 256-bit wide and it allows only vector accesses. One vector load/store unit can access one vector memory. The vector load/store unit does not support complex scatter and gather memory operations. As, we use a simple 256-bit wide vector memory there is no access for the individual elements of the vectors in the vector memory. It only allows regular and aligned memory access. The latencies of vector load/store operations is 1 clock cycle. In Table 3.10, the supported load/store operations are presented.

3.5.3 Vector Pass Unit

A vector pass unit is introduced to pass vectors between register files and to establish the connection between vector and scalar registers. It operates 4 different vector operations: *vec_pass, vec_get, vec_set* and *vec_clone*. *vec_get* reads an element from the vector register, depending on the given index and writes it to a scalar register, vice versa *vec_set* writes to a particular element of a vector for scalar register file. *vec_clone* replicates a vector data, and *vec_pass* reads a vector from one register file and writes it to another one if there is any. Table 3.11 summarizes the vector pass unit operations.

Additionally, the vector pass unit includes vector multiplex operations. The *vec_mux* operation is required to handle conditional executions in SIMD. In the SIMD programming, there are cases when the operation to be performed on each element of vector is dependent on the element itself. A solution for that problem can be found by using the *vec_mux* (mask) operations. First, a flag (mask) vector is created by the comparison unit. Thereafter, the *vec_mux* operation is executed. Basically, it multiplexes two

Operations	**Operand1**	**Operand2**	**Operand 3**	**Result**	**Latency**
vec_pass	vector	-	-	vector	0
vec_get	vector	index		scalar	0
vec_set	scalar	index		vector	0
vec_clone	vector	-	-	vector	0
vec_mux	vector	vector	flag	vector	0
vec_mux_c	vector	scalar	flag	vector	0

Table 3.11: Vector pass unit operations.

vectors, or one vector and one scalar, according to the previously calculated flag vector. The flag vector is 8-way 1-bit vector, which determines which vectors element will be picked. This operation is needed to resolve conditional statements. In Section 4.4.2, we will explain conditional executions by giving an example.

3.5.4 Flag Units

Unlike scalar processing, vector processing requires other forms of conditional execution to vectorize code containing conditional statements. Therefore, flag (mask) vector operations should be introduced to handle conditional executions. Previously it has been stated that the output of vector conditional operations are 8-bit flag vectors. A flag vector controls the element positions where a vector instruction is allowed to update the result vector. The flag vector may be held in one or more special flags or mask registers. As a result, another register file mechanism is introduced for flag registers that contains 8 by 1 bit flag data.

The operations on flag registers are done in 3 functional units: the *fpsu* unit, the *flgu* unit and the *fintra* unit. As the names suggests, the *psu* unit is responsible for flag pass operations, in a very similar way as the *vpsu* unit does. The *flgu* unit handles the logical operations on flags. This gives more flexibility on the efficient control of complex conditional statements. Finally, the *fintra* unit operates on the individual elements of a flag vector. It can calculate the maximum or the minimum element in a vector. The *fintra* unit can also do logical operations. These operations are not between two vectors, but within the elements of a vector. It outputs a single scalar result, but it is connected to a 32 bit register file.

3.5.5 Vector Register Files

A vector register file is an important component in vector architectures. The source of vector arithmetic operands and the destination results are registers in our architecture. The vector register file provides storage for vector elements and access ports to vector functional units. Since our architecture is a multiple lane architecture, the implementation of the vector register file is not as complicated as it is presented in [2]. There is no so much difference between the scalar and the vector register files, for the processor that we extend. The only difference is the width of the register file. However, this structure limits the flexibility of use of individual vector elements. In order to pick an element from a vector register, another operation must be executed, which is explained in the vector pass unit discussion.

The proposed architecture can process 8-way vectors, consisting of 32-bit vector elements, in which both integers and single precision floating point numbers are treated in the same way. As a result, there is no difference from the register file point of view between the two of them. The duty of the vector register file is to provide and store 256-bit wide data to the vector functional units and vector load/store unit. Vector functional units use 256-bit data as their operands in the vector register as their operands. Integer and floating point number vectors are stored in the same vector register file as 8 by 32-bit wide vectors.

The width of the vector register file is enough to make it costly. Therefore, we wanted to keep the port configuration simple. Two read ports are required to provide data for the vector functional units, as there is no operation that necessitates more than 2 vector data. Similarly, one write port connected to the vector load/store and arithmetic units is enough for a one vector issue slot implementation. If another vector issue slot is introduced, it is better to increase the number of write ports of the register file to make possible its use by more than one issue slots or another register file has to be introduced.

The size (length) of a register file is another important design choice. Silicon Hive simulator provides a switch to simulate the program for different register lengths. Therefore, the final sizes of the registers are fixed after evaluating the needs of the ray tracing algorithm in the next chapter. We started with a large size, which is 256 registers and decreased it gradually as much as possible to the point where similar performance results are still achieved. It has been observed that, until 32 registers, there is no much decrease in performance. As a result, we fixed the vector register file size to 32 registers.

A scalar register is also included to provide data for scalar-to-vector or vector-to-scalar operations. This register file is connected with other existing issue slots and register files. Additionally, a flag register has been included to provide storage for previously mentioned flag vectors. The width of the flag register file (frf) is 8 and the length is chosen to be 32 registers.

3.5.6 Vector Memory

The internal vector memory stores 256-bit wide data. The data to process using SIMD instructions are first stored into the vector memory through the host processor. After that, the vector load/store unit reads form vector memory and writes to the vector register files. It only allows regular access. Unaligned or stride access are not supported. Only vectors as 256-bit wide data can be read from our vector memory. Individual elements of a vector can be accessed by first storing them into the vector register file and than reading each element with the *vec_get* operation. The size of the vector memory should be limited, as it is costly in terms of area. As a matter of fact, since we kept access patterns quite simple our vector memory is not very costly. The size of our vector memory is 32 kB.

3.5.7 Vector Issue Slot

The aforementioned vector functional units are introduced in a vector issue slot. Figure 3.13 shows the instantiation of the vector units in the vector issue slot. The issue slot number 3 (bp_pearl_s3) is the vector issue slot in the figure. The total number of functional units in the vector issue slot is big. This type of distribution of functional units is not efficient. A better approach is to separate them in different issue slot. This will increase the SIMD instruction level parallelism. However, in this chapter our goal is to present a generic vector floating point support. In chapter 4.6, we will refine the issue slot to have better configurations by introducing functional units in multiple issue slots. Additionally, improvements on the ray tracing algorithm using this approach will be presented.

3.6 Overview of the Processor

Figure 3.13 depicts the structure of the proposed extended processor. Essentially, it is a VLIW processor with the addition of floating point support, a vector register file, a vector memory and a number of vector functional units. The processor can issue up to 4 instructions per cycle executed in 4 issue slots.

Issue slot number 1 and 4 are responsible for scalar integer operations and they have not been configured. They are exactly the same ones of the baseline processor, which is described before in Section 2.1.6. The issue slot number 2 is deployed with scalar floating point functional units. It can execute both integer and floating point arithmetic operations. Finally, the issue slot 3 is a vector issue slot that can perform SIMD operations. As it can be seen from the figure, both integer (*varu*) and floating point operations (*fvaru, fmul, fdiv* etc.) share the same data path.

It can execute 8-way SIMD operations. Both integer and vector support is 8x32 bits. Therefore, the width of the vector register files and the vector memory is 256 bits.

The floating point operations supported are addition, multiplication, division, square root, float conversion, comparison and absolute value operations, which are fully compatible with the IEEE 754 single precision floating point standard. It supports simple RISC operations and it has one integer multiplier unit. It provides vector registers of 32 256-bit words each, vector load/store operations to move data from/to memory, to/from the vector registers, and a set of computation operations that operate on vector registers.

3.7 Synthesis Results

The proposed processor is synthesized using the TSMC 40 um low power process technology. The basic gate area for this technology is 0.71 μm^2. The target clock frequency is 333 MHz. The synthesis tool is the Synopsys Design Compiler. The wire load model is chosen to be physical and the ultra effort mode is utilized. The synthesis area results are presented in Table 3.12. Issue slots, functional units, register files and memories are shown separately.

In this generic core, three memories are introduced, namely *dram*, *vram* and *pmem*. *dram* is connected to the scalar load/store unit, whereas *vmem* is read or written from

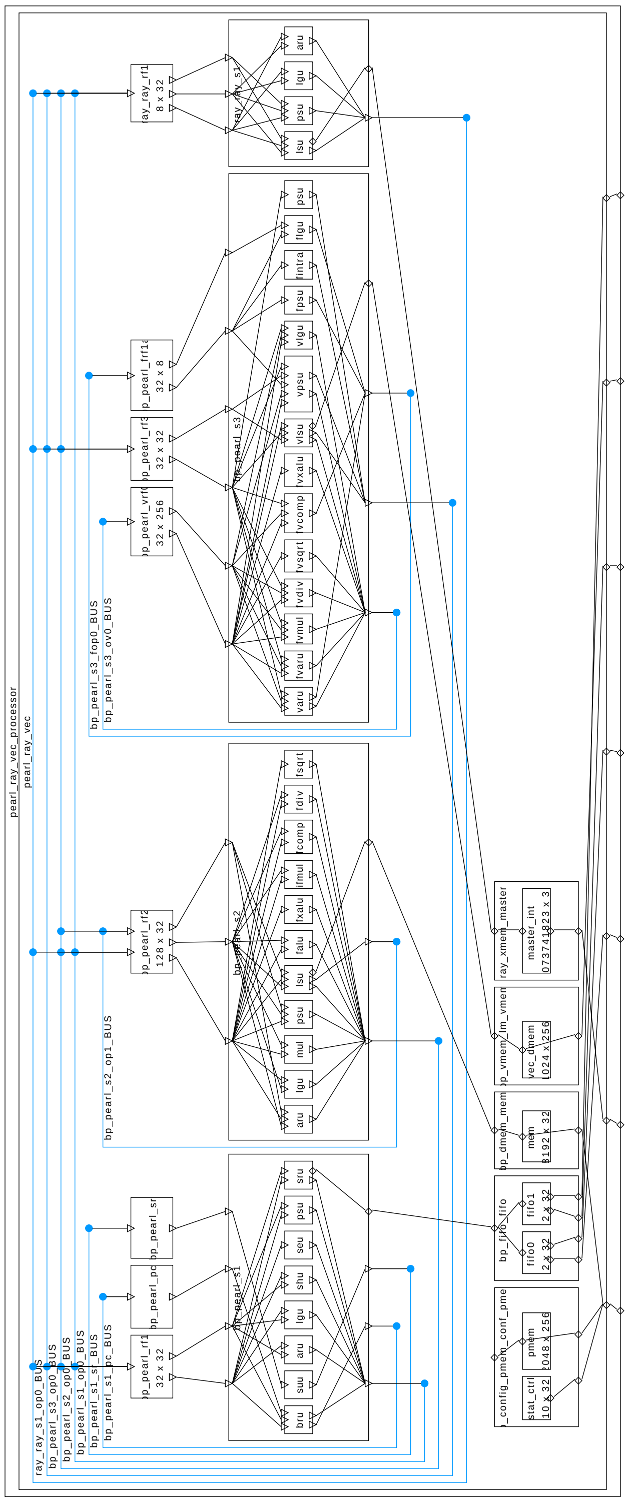

Figure 3.13: The Pearl Ray base processor extended with floating point units and vector units.

		Area (μm^2)	**Number of gates**	**percentage**
Issue Slot 0:	aru	1403	1978	0.16%
	lgu	219	309	0.02%
	shu	787	1110	0.09%
	bru	184	259	0.02%
	psu	133	188	0.02%
	lsu	50	71	0.01%
	Total IS0:	**4026**	**5677**	**0.46%**
Issue Slot 1:	fdiv	16390	23110	1.86%
	lgu	216	305	0.02%
	fsqrt	7536	10626	0.85%
	psu	210	296	0.02%
	lsu	1237	1744	0.14%
	falu	3227	4550	0.37%
	fxalu	71	100	0.01%
	fmul	6585	9285	0.75%
	fcmp	520	733	0.06%
	Total IS1:	**36978**	**52139**	**4.19%**
Issue Slot_2:	psu	87	123	0.01%
	fvaru	21100	29751	2.40%
	fpsu	8	11	0.00%
	flgu	48	68	0.01%
	fvmul	52743	74368	6.00%
	fvdiv	130571	184105	14.86%
	fvsqrt	58000	81780	6.60%
	fvcomp	3843	5419	0.44%
	fvxalu	13061	18416	1.49%
	vlsu	217	306	0.02%
	vpsu	1956	2758	0.22%
	vlgu	2116	2984	0.24%
	Total IS2:	**287019**	**404697**	**32.51%**
Issue Slot_3:	lsu	1237	1744	0.14%
	lgu	210	296	0.02%
	aru	1403	1978	0.16%
	Total IS3:	**3852**	**5431**	**0.44%**
Register Files:	rf0 32x32 2r 1w	8151	11493	0.93%
	rf1 128x32 3r 2w	40132	56586	4.57%
	rf2 32x256 2r 1w	26234	36990	2.98%
	rf3 32x32 2r 1w	8230	11604	0.94%
	rf4 32x8 2r 1w	152	214	0.02%
	rf5 8x32 2r 1w	2191	3089	0.25%
	Total Regs:	**85090**	**119977**	**9.64%**
Memories:	dmem 8192x32	111060	156578	12.58%
	vmem 1024x256	111228	156726	12.60%
	pmem 2048x256	222319	313131	25.18%
	Total Mem:	**444607**	**626832**	**50.36%**
Other Comp.		21232	29915	2.42%
Total AREA		**878952**	**1239322**	**100.00%**

Table 3.12: Area results.

the vector load/store unit. *pmem* stores the long instruction words. The size of the *dram* is 32 kB (8196x32), the vector memory is 32 kB (1024x256) and the program memory is 64 kB (2048x256). Approximately, half of the total area is occupied by the memories.

There are 6 register files (*rf*) in the processor. Their area depends on the length and width values, as well as on the number of read and write ports. Table 3.12 illustrates the properties of the register files and their area results. *rf1* is the largest one, since it has 128 entries, 3 read and 2 write ports. The total area occupied by the register files is about 10% of the total processor area.

Issue slot 0 and issue slot 3 are the ones existing in the Pearl Ray processor. Since they do not include any floating point or vector unit, their area is negligible. The sum of the issue slot 0 and the issue slot 3 is less than 1%. Around 4% is occupied by the issue slot number 1, which has scalar floating point operations in. The other slot in which vector functional units deployed, issue slot 3, covers 32% of the all area. The total area of the vector issue slot is 8 times the area of a scalar issue slot, as expected. We can see the same correlation on the functional unit base. For example, while the *Fsqrt* unit (scalar floating point square root unit) includes 10626 logic gates, *Fvsqrt* (vector floating point square root unit) has 81780. Similarly, the *Fmul* unit (scalar floating point multiplication) occupies 6585 μm^2, *fvmul* (vector floating point multiplication) occupies is 52743 μm^2. In conclusion, the total area of the extended processor is 880000 μm^2. In terms of total number of gates, this number corresponds to around 1.2 million basic gates.

3.8 Conclusions

In this chapter, firstly, we started by describing the floating point extensions of the baseline Pearl Ray processor. We presented a comparison between fixed point and floating point architectures, our floating point design choices and the targeted floating point operations. After that, we presented the implementations of the floating point functional units by using Synopsys DesignWare building blocks.

We showed that the Synopsys register retiming option can be efficiently utilized to improve the timing of the DesignWare floating point units so to introduce pipeline stages. Our experimental results show that this method allows the introduction of efficient pipelines on the DesignWare building blocks. In Section 3.4, a description on how to introduce these new floating point operations into the baseline processor using Silicon Hive core generation flow is shown.

In Section 3.5, the SIMD extension of the baseline processor is presented and the implementation details of vector floating point units are described. Additionally, we presented details on other vector units, such as the vector register files, the vector memory, the flag unit, vector load/store and the vector pass unit.

Finally, an overview of the proposed VLIW/SIMD vector processor is shown, together with the synthesis result. In the next chapter, we will present the Ray Tracing algorithm and evaluate the performance of the proposed processor when this algorithm is executed on it.

Ray Tracing Algorithm Customizations 4

In this chapter, we explain the mapping of a ray tracing algorithm on the implemented architecture. First, the general concept of ray tracing is given and the details of the algorithm are covered. After that, in order to map the algorithm on our processor more efficiently, we present some code optimizations. Additionally, the issue slot configuration of the processor is refined, in order to achieve a higher instruction level parallelism and increase the performance. We also present execution time improvements after each step of the algorithm and further processor optimizations.

4.1 What is Ray Tracing?

Ray Tracing is a method to create a two-dimensional image from a three dimensional scene by shooting rays from a virtual camera through a window of pixels [33]. It produces an image of the scene by launching rays from a virtual camera pointed toward the scene that you want to view, as depicted in Figure 4.1. For this, the algorithm computes the closest intersection between a ray and an object in the scene.

The rays are launched from the camera and each ray goes through its corresponding pixel in the image plane as seen in the figure. As a result, the number of rays cast matches the total number of pixels. The aim of the algorithm is at finding a color value

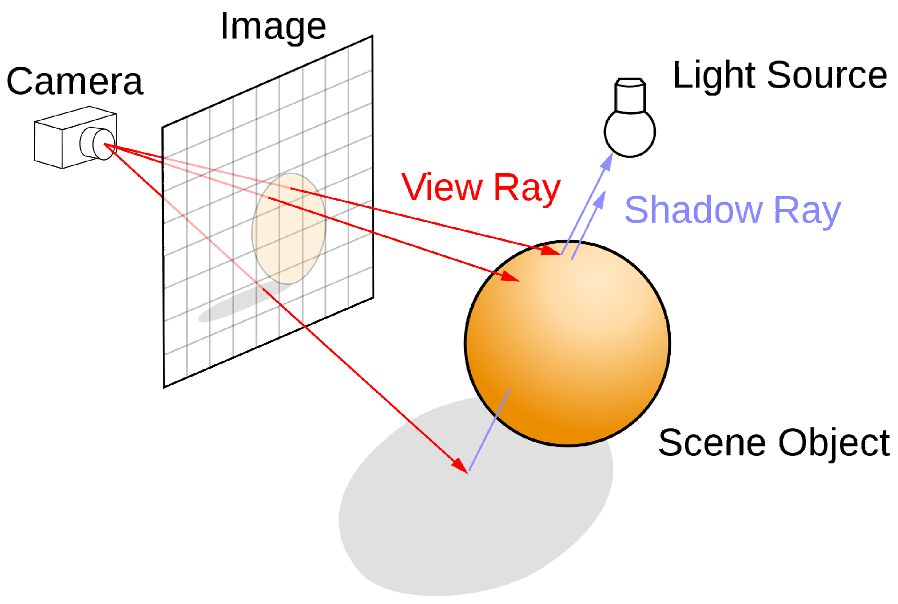

Figure 4.1: The Ray Tracing Schematic from Wikipedia [40].

for each pixel by using the corresponding ray. Once the closest intersection of a ray and an object is found, the color of this object is assigned to the pixel in question. This short part of the algorithm is called Ray Casting, and it does not comprehend illumination effects such as shadow reflections and other light affects.

When there is a hit, i.e. when a ray and an object intersects, the ray tracer computes the color value contribution of the pixel that the cast ray belongs to. First, a path of the light is traced back through the pixels. Then, interactions of this light ray with the objects at the same scene are simulated.

If the object that a ray hits is reflective, the ray can bounce off. In that case, a reflected ray is computed by the algorithm and the same process is done for the reflection rays until it ends up in a non reflexive object or until a certain number of iterations is reached. Finally, these reflected and/or refracted rays will eventually contribute to the color value of the pixel that the primarily cast ray belongs to. That kind of ray tracing algorithm was first presented in [39] and was called *recursive ray tracing*. In this thesis, the ray tracing is referred to as *recursive ray tracing*, as presented in [39].

Additionally, for each intersection point shadow rays are generated in the direction of every light source present in the scene. These are used to compute the effect of the light for that point. Shadow rays are used to verify if the intersection point receives the light. In addition, the algorithm calculates the contribution of each light source to the color of the surface at the point of intersection. That makes the illumination effects look more realistic in the image. This process of coloring by using light is called *shading*.

The ray tracing algorithm comprises a high degree of parallelism. It fits in the category of global illumination models. Unlike the local illumination models, it considers interaction between all elements in the scene for the composition of the image. Effects such as reflection, refraction, shadows, diffusion, specular are natural results of the ray tracing algorithm, differentiating it from the classical scan based rendering techniques. As a result, it brings a high level of realism.

4.2 The Main Steps of the Ray Tracing Algorithm

A ray tracer typically consists of the following steps:

1. Ray generation
2. Ray traversal
3. Intersection test
4. Shading

In the following, we analyze each of these steps in detail:

1. **Ray generation.** In the beginning, primary rays are generated through each pixel in the image window. A ray is defined by two points, namely an origin point and a direction point. The origin of the primary rays is the position of the virtual camera and the direction of the primary rays is the pixel, which the corresponding

ray goes through, in the image plane. The ray generation step can be done in two different ways. The first method is based on having a pre-processing step, in which all primary rays for every pixel are calculated and stored in the memory. Then, the algorithm reads sequentially each ray from the memory. Pre-processing of primary rays enhances the performance of the algorithm at the cost of an increase in the required memory size. The second way is to perform ray tracing after calculating each primary ray one after the other, by skipping any pre-processing. In this way, a large amount of memory space is saved.

2. **Ray Traversal.** After generating the primary rays, the next step is to search for closest intersections between rays and objects in the scene. The easy and näive way is to test every ray for all the objects in the geometric scene. This would work for low geometry scenes with a limited number of objects. However, for more complex scenes, the amount of intersection tests will become computationally expensive. Therefore, ray traversing algorithms are developed to speed up the overall performance. More specifically, they divide the scene into spatial subsets and create acceleration data structures like tree structures, grid structures or bounding volume hierarchies [36]. Instead of checking all possible intersections for a given ray for all objects in the scene, these data structures have to be traversed looking for subsets hit by a ray. As a result, the number of intersection tests is reduced significantly. However, in this study, we have not used any acceleration structures as the main objective of this thesis is the evaluation of the performance of ray tracing in terms of architectural optimizations rather than algorithmic ones.

3. **The intersection test.** The intersection test step calculates if an intersection exists between a ray and an object in the scene. If so, the point in space where the intersection takes place is calculated. The intersection test depends on the chosen geometric primitive in the scene. The scene can be represented with mathematical formulas or combination of geometric primitives. In our case, scenes consisting of spheres as geometric primitives are taken into consideration.

4. **Shading.** After finding the nearest intersection of a ray and an object, the color of the surface at the intersection point is determined and the color value is adjusted based on the light sources in the scene. This step is called shading. Determining the color at the intersection point also involves casting of extra rays for reflective or refracting surfaces. As a result, the same process is repeated recursively for the new refracted or reflected rays.

4.3 The Mapping of the Ray Tracing Algorithm

In this section, we describe the mapping of the ray tracing algorithm to the Silicon Hive processor. In Section 2.1.3, we have described the Silicon Hive system in detail. As a matter of fact, the host processor loads the program and the data into the memory of the Silicon Hive processor. In this section, we explain which parts of the algorithm are compiled as a host code and run on the host processor and which parts are mapped and run on the proposed Silicon Hive architecture.

The algorithm steps, which are not considered deserving the evaluation of the performance of the ray tracing algorithm, are compiled for the host processor. These are pre-processing steps, such as creating camera variables, setting up the scene, constructing the image window, etc. After these initializations, the host loads the camera and the scene information into the memory. As a result, they can be processed by the proposed processor. Additionally, the program compiled by the HiveCC compiler is loaded into the internal memory of the proposed Silicon Hive processor.

From that point on, our processor performs all the aforementioned steps starting from the generation of the primary rays till returning a color value for every pixel. The resulting color value is written back into the memory by the processor and the host reads the color values from the memory and constructs the image.

The initial aim was at running the algorithm using scalar floating point operations. This implementation is used as a reference design to compare the results, when the SIMD (vector) version of the algorithm is simulated using vector instructions in the vector issue slot of the processor. In Table 4.1, we present the results of the cycle count for the ray tracing algorithm on our processor using only scalar floating point operations.

Number of Iterations	**1**	**2**	**3**	**4**
Cycle count per pixel	357	526	582	601

Table 4.1: The number of clock cycles for the implementation of the ray tracing with respect to the number of iterations.

4.4 Vectorization of the Ray Tracing Algorithm

There are 2 important requirements to be able to vectorize an algorithm. First, there must be some opportunity to perform the same calculations on a stream of data elements and, second, each of the calculations must be independent by the results of the other calculations in the stream [25]. As the nature of the ray tracing is to shoot rays through each pixel and calculate ray-object intersections over all objects in a scene against all rays cast, the smallest independent element in the algorithm is the pixel. Thus, the parallelization of the rendering process is on per-pixel basis.

To exploit the vector instructions presented in Section 3.5, the code needs to be rewritten and vectorized in a single instruction multiple data format. In literature, there are several implementations of the ray tracing algorithm for single instruction multiple data format like, for example, [5].

Vectorization can be done by transforming loops of scalar operations into a sequence of vector operations. In fact, there are two possibilities to vectorize a ray tracing algorithm. The former is by vectorizing using spheres and the latter is by using packets of rays. The decision on which of the two to use needs to be taken. For an efficient vectorization of the ray tracing algorithm, correlation between the elements of the vector is very important, as it can reduce the overhead resulted from the mask operations used in SIMD implementations. In terms of ray tracing, correlation can be defined as the degree of spatial deviation within a set of vector elements, which can be possible rays or scene

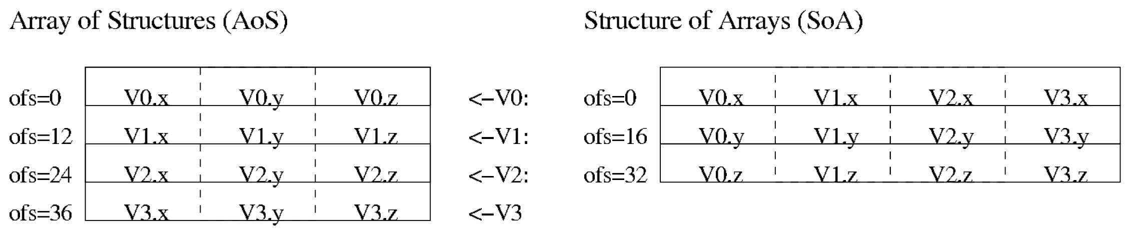

Figure 4.2: Array of Structures and Structure of Arrays comparison [36].

objects [4]. The details on how this affects the performance, will be discussed in Section 4.4.2.

In [38], the authors have shown that, there is a high correlation between rays. For primary rays the correlation will be higher. Nevertheless experiments show that the second, and the third level rays have a large correlation as well [37]. On the other hand, for geometric primitives the correlation is limited. In addition, some pre-computations are required to store spheres (objects), which are close to each other in the same vector. As a result, the packet of rays phenomena, first introduced in [38], is used to vectorize the ray tracing algorithm in this work.

4.4.1 Vectorizing by Packets of Rays

In packet ray tracing, rays are not traced sequentially. They are traced as a group of 8 rays. To exploit the vector instructions in our architecture, the data organization needs to be redesigned for vector of rays. This can be done by using 2 different configurations: an array of structures or a structure of arrays. Although the organization of the data as "array of structures" is more intuitive, an efficient SIMD programming requires the reorganization of such data into a more SIMD-friendly "structure of arrays" form (see Figure 4.3).

Any data used in the algorithm is replaced by an array of 8 values, one for each ray. As a result, these data can be used as operands for any vector operation defined in our instruction set. A ray consists of an origin and the direction data. As a result, the structure of a ray in the initial scalar algorithm was something like the code in Figure 4.3 .

```
struct ray{   point o_p; //origin
              point d_p; //direction};

struct point{ float x;
              float y;
              float z; }
```

Figure 4.3: Declaration of a ray structure for the scalar algorithm.

For the SIMD implementation, *fvector* type is used to define the points. *fvector* is a packed type and consists of 8 32-bit single precision floating point numbers. As a consequence, a point is defined by 3 `fvectors`, each defining the coordinate of 8 points.

Similarly, *vray* consists of *vpoints*, which represents 8 points in a structure of arrays. Figure 4.4 shows the *fvector* definition.

```
Type fvector   __packed8x32;

struct vray {   vpoint ovp;
                vpoint dvp;}

vpoint{    fvector x;
           fvector y;
           fvector z; };
```

Figure 4.4: Declaration of a vector ray structure for SIMD implementation.

Table 4.2 shows the improvement via vectorization compared to scalar floating point operations. The numbers denote the number of cycle counts per pixel and object. When the number of iterations is equal to 1, vector improvement is 7.4x. As we are using 8-way SIMD instructions, 7.4x. is a quite satisfactory result. However, when the number of iterations increase, the performance gain decreases. This happens because of the inefficient handling of the conditional checks in the algorithm. In the next section we present a solution to this problem.

Number of Iterations	**1**	**2**	**3**	**4**
Scalar	357	526	582	601
Vector	48.4	95.7	142.9	190.2
Improvement	**7.4x**	**5.5x**	**4.1x**	**3.15x**

Table 4.2: Comparison of the number of clock cycle for the scalar and the vector implementation of the ray tracing algorithm with respect to the number of iterations.

4.4.2 Conditional Execution on Vectors

The Ray Packet data structure represents 8 rays. Every calculation in the algorithm is done using these structures. Therefore the same operations have to be executed by each ray in the same vector. Anyhow, a question arises: *what will happen when those rays intersect with different objects?* In other words, *how the conditional statements will be executed for a vector?*

In a vectorized code, conditional branches are resolved using mask operations. First a flag vector is created, whose elements specify which branch is going to be taken for the corresponding elements of the vector. To do that, all possible results of each branch needs to be computed for all elements of the vector. Afterwards, based on the flag vector, a condition decision will be taken based on the correlation of the vector elements. If all the elements of a vector have the same behavior in conditional statements, it is advantageous.

In the initial code there are many conditionals. For vector operations, those conditions need to be removed, as the result of conditional statements can be different for each ray in the same vector. Therefore, mask operations are used. The example given in Figure 4.5 shows how the conditional statements are transferred to mask (multiplex) operations.

```
for (i=0; i<8; i++)
     if (A[i]>0) then
     A[i] = B[i];
     else
     A[i] =C[i];

flag_X = fvcmp(vec_A,0); //creates the boolean flag vector
vec_A = vec_mux(vec_B,vec_C,flag_x); // chooses elements with
                                     //respect to the flag vector
```

Figure 4.5: Conditional execution on vectors.

As the figure shows, flag vectors are used to resolve conditions in vector processor. We have 8 by 1 flag vectors to store the results of the conditional statements. First, the flag vector is assigned based on the condition. Then, two possible branches of the conditional statements are calculated and stored in two different vectors. After that, these two vectors are multiplexed using the *vec_mux* operation and the corresponding values picked as the final result, based on the flag vector. The advantage of using mask operations is that there is no more control dependency. Therefore, more instructions can be issued in parallel by the compiler, which increases the ILP of the algorithm. On the other hand, now, there is an extra mux operation and, more importantly, two possible results coming from different branches of the conditional statements must be executed, even though, one of them is not going to be picked. As a result, performance drops due to extra executions. This becomes more crucial when there are conditional function calls.

The function calls take place even when only one element of the ray vector satisfies the condition. This is a drawback of the use of vector operations. However, as it mentioned before, rays in the same vector have similar behavior because of the large correlation between them. Therefore, we can use the same conditional statements for all the elements in that vector. To do that, another vector operation is added to the architecture which checks all the elements of the flag vector. If all bits of the flag vector are the same, this operation returns this value and, based on this value, the conditional statement is handled.

The results (see Table 4.3) show that the use of condition checks for the vectors increases the performance of the algorithm, as expected. However, when the number of iterations increases, the contribution of using condition checks for the vectors decreases, since rays become less correlated, as shown in the table.

4.5 Code Optimizations

In this section, we describe the code optimizations. The importance of the ILP for VLIW processor has already been mentioned. As a result, the main goal of code optimizations

Number of Iterations	**1**	**2**	**3**	**4**
Vector Implementation	48.4	95.7	142.9	190.2
After condition check	43.8	67	74.6	77.8
Improvement percentage	**9.5%**	**29.9%**	**47.7%**	**59.0%**

Table 4.3: Improvement after utilizing the vector condition checks.

is to increase the instruction level parallelism, which, in turn, means that the program can fully exploit the VLIW architecture. The proposed architecture has 4 issue slots, which means that 4 instructions can be executed simultaneously. However, for the initial ray tracing algorithm, the instruction level parallelism was very limited. The algorithm was written in a highly sequential manner. There were many function calls, recursive functions and conditions, which prevented the scheduler to find optimal schedules to exploit the ILP. As a result, if we want to increase the ILP, we have to remove them. In the rest of this section, we will explain these algorithmic improvements, which to improve ILP.

4.5.1 The Use of Inlines

At first, function calls are redefined using the inline feature. The scheduler schedules the basic-blocks on a block-by-block basis. A basic block is a sequence of operations, where only the first and the last operations are allowed to be *target of a jump* and *a jump operation*, respectively. This means that, each function is scheduled individually. However, when inlines are used, bodies of those functions are copied to where these functions are called. Hence, the code ends up with larger basic blocks to schedule. Another advantage of using inlines is to remove the overhead of function calls, as in case of a function call, the current variables have to be stored into the stack and then reloaded. In case of inline usage, the overhead of the load/store operations is gone. Anyhow, one drawback of function inlines is the increase of the program size, which turns into more program memory. In our case, the functions replaced with inlines are small sized. Therefore, the increase in program size is negligible. When appropriate functions are exchanged with inlines, the performance of the overall algorithm increases, as presented in Table 4.4.

Number of Iterations	**1**	**2**	**3**	**4**
Vector + condition check	43.8 (8.1)	67.0 (7.9)	74.6 (7.8)	77.8 (7.7)
Vector + cond. check + inline	39.5 (9.0)	54.7 (9.6)	61.0 (9.5)	63.0 (9.5)
Improvement percentage	**9.80%**	**18.30%**	**18.20%**	**19%**

Table 4.4: Cycle count improvement after using inlines instead of function calls.

```
for(0<i<n)
y[i] = f(A[i]);
```

```
for (0<i<N/2){
y[2*i] = f(A[2*i]);
y[2*i+1] = f(A[2*i+1]);}
```

a)

```
for(0<i<n/4){
y[4*i] = f(A[4*i]);
y[4*i+1] = f(A[4*i+1]);
y[4*i+2] = f(A[4*i+2]);
y[4*i+3] = f(A[4*i+3]);}
```

b)

Figure 4.6: Loop unrolling by a) a factor of 2 and b) a factor of 4.

4.5.2 Recursions

Another factor that limits the ILP is the recursive function call. The trace function calls the shade function. Then, in the next iteration, the shade function calls the trace function and this recursion goes on until the number of iteration steps is reached. There are two problems caused by a recursion call: it avoids the usage of the inline approach explained before, and it also prevents the utilization of further optimizations, like loop unrolling, software scheduling, etc. Therefore, the recursive function calls have to be avoided to achieve a higher ILP. To prevent the recursion, the "*shade*" and the "*trace*" functions copies are defined and, instead of calling a function, we call these inline copies. Table 4.5 shows the improvement in number of clock cycles, after the removal of recursions.

Number of Iterations	**1**	**2**	**3**	**4**
Vector+cond. check+inline	39.5	54.7	61	63
Vector+cond. check+inline+norecursion	25.3	39	43.6	45.5
Improvement	**35%**	**28%**	**28%**	**27%**

Table 4.5: Improvement after removing recursions.

4.5.3 Loop Unrolling

Loop unrolling is another technique that can be applied to increase the ILP. It replicates the body of a loop depending on the loop unrolling factor. Consequently, it produces a larger basic block of instructions for the scheduler to work with. This gives to the compiler more flexibility to schedule the operations in a more optimal way. Therefore, the ILP increases and the number of clock cycles decreases. The increase in terms of program memory allocation is a trade-off. When the body of a loop is replicated multiple times, eventually, the number of instructions increases. Figure 4.6 shows the unrolling of a loop by a factor of 2 and by a factor of 4.

The Silicon Hive compiler supports automatic loop unrolling. It can be simply enabled by adding an unroll pragma to the loop to unroll. For example:

- #pragma hivecc unroll=2 means the loop will be unrolled 2 times.

- #pragma hivecc unroll=off means no unrolling.

Loop unrolling was used during the intersection test of the rays with the spheres in the algorithm. In the first experiment, the loops are unrolled by two times, resulting in 20% decrease in the number of clock cycles, and a 10% increase in the program word count. Then, loops were unrolled by a factor of 4. This improved the performance by 4%. However the program size increase by 17%. In addition, the simulation time increased, since the complexity of the scheduling is increased. An increase of the unroll factor of more than 4 times did not improve the results, as the size of the register files is exceeded and the compiler could not manage the scheduling of the code. As a result, unrolling with a factor of 2 is chosen. Table 4.6 shows the effect of loop unrolling in terms of cycle counts.

Number of Iterations	**1**	**2**	**3**	**4**
No unroll	25.3	39.0	43.6	45.5
Unroll 2	20.8	31.8	35.4	37,0
Unroll 4	19.5	29.7	33.2	34.6

Table 4.6: Cycle count improvement via the unrolling technique.

4.6 Architectural Refinement

It has been shown already that the data level parallelism can be efficiently exploited by having SIMD Ray tracing and by using vector instructions. Now, the intention is to show that ILP can also be exploited in a Ray Tracing algorithm, as opposed to what claimed in [36]. The *out-of-box* algorithm is optimized by using the techniques mentioned in the previous section to leave a more flexible play ground to the compiler to schedule more instructions in parallel. However, in order to take full advantage of the ILP, the architecture needs to be tuned in such a way that there are multiple vector issue slots that can execute multiple instructions in parallel using long instruction words.

4.6.1 Double Issue Slot

A first approach to increase the ILP is by creating a copy of the same vector issue slot. As a result, any two vector operations can be issued in parallel by our VLIW architecture. Therefore, the ideal ILP for vector operations is 2. Again, the compiler will handle the scheduling to find an optimal one to execute 2 vector instructions in parallel. The second vector issue slot is a replica of the vector issue slot previously explained. The only difference is that it does not have its own load/store unit and vector memory. Both share the same vector memory and vector load/store unit. However, it has the same register files with the same size and port configuration.

Table 4.7 denotes the resulting cycle count improvement after introducing the second issue slot. For the sake of simplicity, we present the results of only 3 iterations in the table. The improvement after the vectorization of the code and the use of only a single vector issue slot was 7.7 times compared to the scalar implementation. By using the double issue slots, it resulted in a 13.0 times improvement, in comparison to the scalar implementation. The ideal improvement for double vector issue slots would be 16 times that of the scalar implementation, as in theory 16 floating point operations can be done sequentially. We remind that these results do not make use of any code optimization.

Table 4.7 shows also the performance comparison of two architectures before and after utilizing the code optimizations. Here, we compare the effect of code optimizations on both architectures. The results show that code optimizations improve the performance of the double vector issue slot implementation more than the single vector issue slot implementation. Although it decreases the cycle count by 41% for the single issue slot implementation, it reduces 57% for double vector issue slot implementation. The reason is that the double vector issue slot can exploit more the achieved ILP after the code optimizations, compared to the single vector issue slot implementation. Finally, we observe that the overall gain of the double issue slot architecture with the code optimizations is 30 times better than using only one scalar floating point issue slot. This result is achieved by the VLIW and the SIMD combination in the proposed architecture.

	Cycle Count	**SpeedUp**
Scalar IS	601.0	1.0x
Single Vector IS	77.8	7.7x
1 Vector IS with code optimizations	34.6	17.4x
Double Vector IS	46.12	13.0x
2 Vector IS after code optimizations	20.4	30.0x

Table 4.7: Performance comparisons for single and double vector(SIMD) issue slot implementations.

To have an idea on the achieved SIMD ILP, we have investigated the scheduling results. The Silicon Hive scheduler provides functional unit utilization results. The algorithm has a huge main outer loop where most of the computation takes place. As a result, we focused on that loop. It has been observed that the achieved ILP for SIMD instructions is 1.77. The ideal achievable SIMD ILP is 2, since there are 2 vector issue slots that can run floating point SIMD instructions consequently.

These results prove that the ILP for the ray tracing algorithm can be efficiently exploited. Doubling the vector issue slot resulted in around a 1.8 increases in the performance. However, the deployment of another vector issue slot is costly in terms of area. The area results in Section 3.7, show that 30% of total processor area comes from the vector issue slots. The inclusion of another vector issue slot increases the total area 30% and the logic area by 60%.

4.6.2 Final Configuration

The inclusion of a copy of the vector issue slot results in more than 60% increase in total logic area of the processor. This is very high compared to the 41% improvement in the performance. Anyhow, the question is: do we really need copies of every functional unit in both issue slots? The answer can be found by exploring the resource utilization scheme provided by the compiler. While some functional units can be critical for the performance, some may not be utilized that much. Table 4.8 shows the utilization of each vector functional unit for the main loop of the algorithm, which is the largest portion of the code.

Functional Units	**Utilization % (cycles)**
flgu	11%
fvaru	21%
fvcomp	12%
fvdiv	3%
fvmul	18%
fvsqrt	3%
fvxalu	1%
psu	1%
vlsu	5%

Table 4.8: Vector functional unit utilization in percentage with respect to the number of cycles.

As the table shows, the floating point vector arithmetic unit (*fvaru*) and the multiplier (*fvmul*) unit are the mostly utilized units. Therefore, we copy these functional units into the two issue slots. Additionally, the pass unit (*vpsu*) is copied, as it is needed to pass the data between two issue slots. The rest of the vector functional units is not replicated. Instead, we separated them into two issue slots. This can result in many configurations. However, our experiments have showed that there is limited difference between these configurations, as long as the number of functional units is kept similar for the two vector issue slots. As a result, the configuration in Figure 4.7 has been chosen in our experiments and it is the final architecture used to compare the results with other platforms.

	cycle count	**SpeedUp**
Scalar IS	601	1.0x
1 Vector IS	34.6	17.4x
2 Vector IS	20.4	30.0x
Final Architecture	21.4	28.1x

Table 4.9: The comparison of the final configuration with the previous results.

In this configuration (see Table 4.9), the logic area increases by only 15% compared to single issue slot (it was 60% for double issue slots). However, the total number of cycles is reduced by 38%. If those results are compared with the 2 vector issue slot configuration, we can observe that the overall performance did not drop much. Anyhow, a large area is saved. The average ILP in this configuration is 1.7, which is very close to the previous result. As a matter of fact, a very close performance achievement is realized by a much less area increase.

4.7 Conclusions

In this chapter, the ray tracing algorithm was presented in detail. First, the general concept of ray tracing has been given and the details of the algorithm have been covered. After that, in order to map the algorithm on our processor more efficiently, we presented some code optimizations. Additionally, the issue slot configuration of the processor is refined in order to achieve higher instruction level parallelism and increase the performance. We also present execution time improvements after each step of the algorithm and further processor optimizations. In conclusion, the results show that the ray tracing algorithm is embarrassingly parallel algorithm. Both instruction level and data level parallelism can be efficiently exploited. This is proved by the use of SIMD instructions and by introducing multi issue slots and investigating the resulting ILP.

In the next chapter, we will compare the performance results of the ray tracing algorithm on the proposed architecture with other architectures.

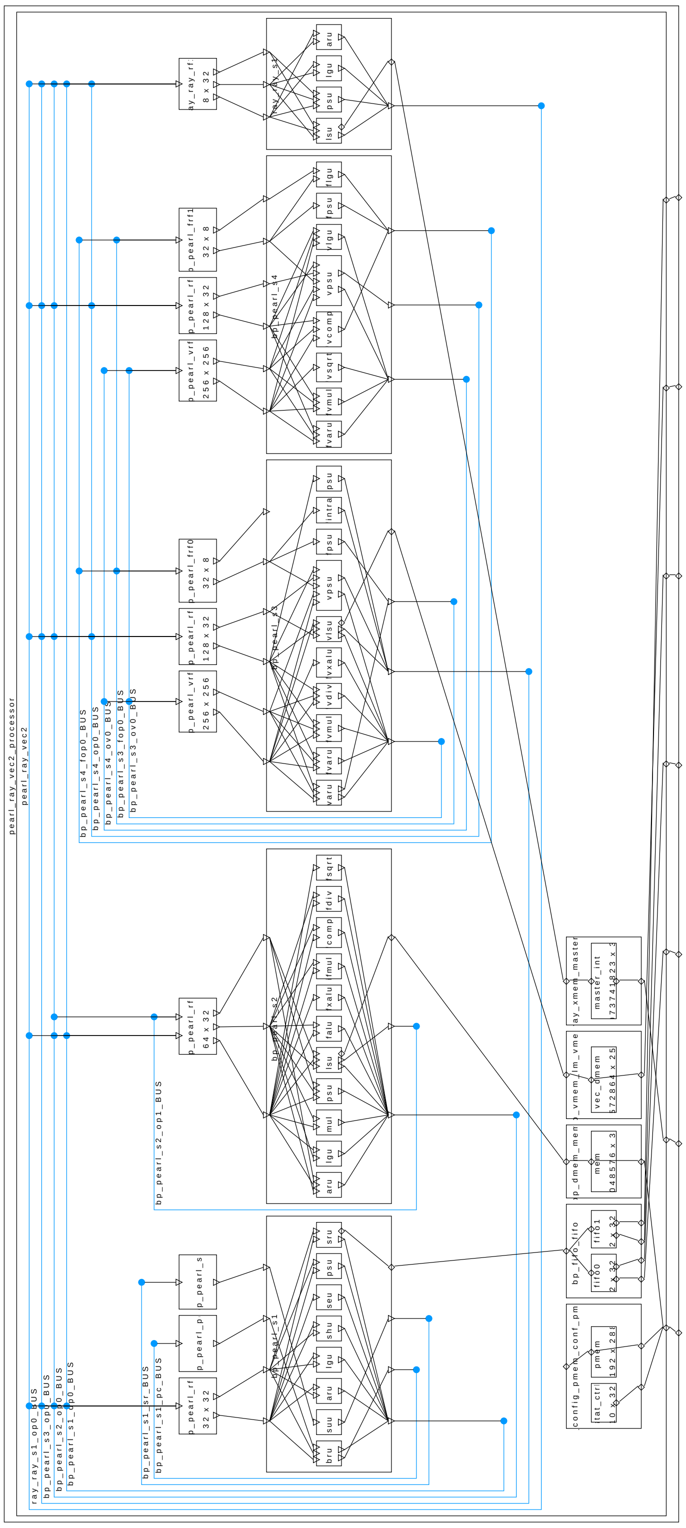

Figure 4.7: The final architecture with 2 vector issue slots.

Experimental Results 5

In this chapter, to have an idea of how well our architecture can perform on the ray tracing algorithm, we compare the performance of the proposed extended SIMD/VLIW floating point processor with a very high-end graphic processing unit and a general purpose processor, in terms of number of cycles, total execution time and power consumption. As our processor is an embedded DSP, a huge difference for power results is expected. GPUs have immense parallel floating point operation capability. Nonetheless, this comparison provides a general idea on the relative performance. At first, we present performance results for the algorithm presentedin the previous chapter on some other architectures. Then, we compare and evaluate the performance of our processor with respect to the other architectures. Finally, we present comparative results based on the C-ray benchmark.

5.1 The GPU Comparison

The goal is to compare the GPU performance and our processor. Therefore, the ray tracing algorithm, presented in Chapter 4, is mapped on a GPU. GPUs are massively parallel processors. At the end, our expectation is to have faster execution time on the GPU.

The implementation of the ray tracing algorithm on the GPU is realized using the CUDA Software Development Kit of Nvidia. CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA. It enables programmers to use GPUs for non-graphics computations. GPUs, with their highly parallel architecture, are capable of processing over more than a thousand threads at high speeds. This is why CUDA and GPU computing have gained interest from the industry and academia in the last years. Many highs speed parallel processing on CUDA platform can be found in the literature, for example [16].

The GPU we evaluated is the Nvidia Quadro 4000. It has 256 CUDA cores that can execute thousands of threads in parallel. Its maximum power consumption is 142 watt according to the data sheet [24]. As a result, compared to our processor, there is a huge floating point compute power in this GPU.

As done for the mapping of the ray tracing algorithm into our platform, it is necessary to separate the code for the host and the code for the device, in order to utilize Recursive Ray-Tracing [32]. The CPU generates the scene and camera variables. Afterwards, this data is passed from the host CPU to the GPU and written to the GPU memory using the *cuda_mem_cpy* function.

In terms of parallelism of the algorithm, a similar approach to the one used in vectorization of ray tracing is followed for the GPU implementation as well. Each pixel, which means each ray, is assigned to one thread of the GPU. In this way, each individual thread calculates the lighting, shading and intersections for all objects. Each thread needs an access to the entire scene. As a result, the scene information and camera variables are stored in ”the shared memory” of the GPU.

The implementation is straightforward: we create a CUDA grid dimensions (dimGrid) and block dimensions dimBlock so that (dimGrid.x*dimGrid.y)*(dimBlock*dimBlock.y) is exactly the number of pixels in the image and the host invokes the CUDA kernel with these device specifications. The CUDA kernel includes all the steps of the ray tracing algorithm starting from the generation of the primary rays. The recursive function calls in the algorithm removed in the GPU kernel too, in similar way to the one explained in the code optimizations section. However, conditional statements and jumps are still present in the GPU version of the code and that cause the GPU to slow down. As this is something related to the GPU architecture itself, we will take this drawback into account.

All threads need to be synchronized so to start simultaneously. Construction and synchronization of the threads take some time. The scheduling of the threads is done automatically. If there are more threads available for the GPU, the compiler decides how to schedule them. The performance comparison between the GPU and our VLIW vector processor can be seen in Table 5.1.

	Execution Time (ms)		**Number of Cycles**	
Number of Objects	**GPU**	**Our processor**	**GPU**	**Our processor**
1	75	82	71.3	28
4	80	117	76	39
8	93	246	88.4	82
40	127	1122	120.7	374
80	170	2097	161.5	698

Table 5.1: Performance comparison between the GPU and the proposed architecture.

Our processor gives better performance results for the smaller scenes that have less number of objects because of the previously mentioned thread construction and synchronization of the GPU. Approximately, 70-75 ms are spent on thread construction initially. When the scene size becomes relatively large, this initialization delay becomes negligible and the performance of the GPU overtakes our processor.

When the total number of objects in the scene is 80, the GPU executes the algorithm 12.3 times faster than the proposed processor. The clock frequency of the GPU is 3 times higher than our processor. So, if we normalize the execution time with respect to corresponding clock frequencies, it can be concluded that the GPU performs 4 times better than our architecture. However, as we have previously mentioned, the power consumption and the area of the GPU should be taken into consideration.

If we compare the power consumption of two devices, the GPU and the implemented processor, the former consumes much more power than the latter. The maximum power consumption of the used GPU is 142 watt, according to the data sheet. We do not know exactly how much percentage of the GPU power is used to execute our algorithm, but we know that total number of pixels in the image is larger than the total number of threads in the GPU. Therefore, the resource utilization of the sources of the GPU must be close to the maximum. The power estimation results for our processor show that the average power consumption is around 0.2-0.3 watt. In conclusion, the GPU performs 4 times faster, although the power consumption difference is notable. Around 700 times more power is consumed by the GPU.

5.2 The CPU Comparison

Another parallel implementation to compare the performance of our ray tracer is the parallelization with OpenMp of the code executed on a 4 core (8 thread) Intel Xeon machine. In fact, the OpenMP implementation is very similar with the CUDA implementation. The scene data structure is kept in the shared memory avoiding the overhead of distributing it to multiple processors [12]. Each thread is assigned to one pixel and the results are taken for different number of threads. The code piece in Figure 5.1 shows how the parallelization is done using OpenMP pragmas.

For this comparison, we used an Intel Xeon X5570 processor. It is a 4 core processor that runs at 2.8 GHz clock frequency. The number of threads is 8, including Intel's hyperthreading technology. It consumes 95 watt power on average, according to the data sheet [17].

```
#pragma omp parallel for
for(x=0; x < width*height; x++)
{
     finalImage[x][y] = trace(x, cam, &data);
}
```

Figure 5.1: The Parallel ray tracing implementation using OpenMP.

	Execution Time (ms)			Number of Cycles		
# of Obj.	**1 thread**	**8 threads**	**Our core**	**1 thread**	**8 threads**	**Our core**
1	740	160	82	2146	464	28
4	990	161	117	2871	466	39
8	1055	210	246	3059	609	82
40	4469	650	1122	12960	1885	374
80	6152	938	2097	17840	2720	698

Table 5.2: CPU comparison with our processor. Intel Xeon processor use single thread and 8 threads implemented on OpenMP.

Table 5.2 shows the comparison between our processor and the Intel Xeon CPU. The results for 8 core CPU was obtained using 1 thread and 8 threads on our OpenMp implementation. We can state that an 8 thread implementation improves the performance by 6.5 times compared to single core implementation (when the number of objects is 80). Linear speed up is observed with respect to the number of threads. However, parallelization is not as effective as in our vectorized implementation. When the algorithm is vectorized and simulated in our 8-way vector architecture, the achieved performance speed up was very close to 8 times. Although the algorithm is parallelized in the same manner for the 8 core machine, 8 times improvement was not feasible on an 8 core machine. However, after utilizing techniques like load balancing or using more effective communication as presented in [18], the performance of the OpenMP implementation might be increased.

It is more convenient to make a comparison when the number of objects is large. As a result, the focus is on the last line of Table 5.2, when the total objects are 80. In terms of execution times, our VLIW/vector processor is 3 times faster compared to the single thread (single core) implementation in the CPU. When the code parallelized in OpenMP is mapped onto 8 cores of Intel Xeon processor, it can execute the algorithm in 938ms, which is 2 times faster than ours. Nevertheless, if the results are normalized with respect to the clock speeds, and we speak about the number of clock cycles, rather than execution time, our processor performs the same algorithm 3.9 times faster than the CPU. It may be concluded that our processor can compete with an high end CPU for the execution of the ray tracing algorithm. Furthermore, if we take a look at the power consumptions of the two, the CPU consumes hundreds times more power than our processor. Power estimation for the former is 80-90 watt, and for the latter is 0.2-0.3 watt.

5.3 C-ray Benchmark

C-ray is a simple open source ray tracing benchmark, which is created to measure floating point CPU performance. The algorithm that C-ray uses and our Ray tracer are not exactly the same. However, they are quite similar. There are some small differences, but most of the key things (intersection test, shading, simple tracing etc.) are the same. The source code and the performance results for many different CPUs on the Cray benchmark are presented in [3]. These results are based on a fixed scene description described by the benchmark. A very similar scene has been created for the ray tracing algorithm we used in this study and the algorithm is simulated on our processor. Figure 5.2 and 5.3 show the resulting images. Figure 5.2 is the original scene rendered using the C-ray tracing algorithm in [3], and Figure 5.3 is rendered using our ray tracing algorithm on our platform. There are small differences in the scene description. However those differences do not affect the overall complexity of the problem. As a result, comparison between the two makes sense. Table 5.3 shows the comparison of our implementation with some CPU results with respect to power and performance analysis.

Again the comparison shows that our processor competes with the presented CPUs in terms of performance. Our processor takes 178 ms to render the scene. Some of the presented CPUs performs better. However, the difference is not much. At most 3 times faster. The power comparison shows the effectiveness of our architecture. The difference in terms of energy consumption is huge as it is depicted in Table 5.3. The MIPS processors date from 1998 to 2002 and were fabricated in 110, 130, and 180 nm processes, which would account for an 8x to 16x relative power difference with the current 40nm processes.

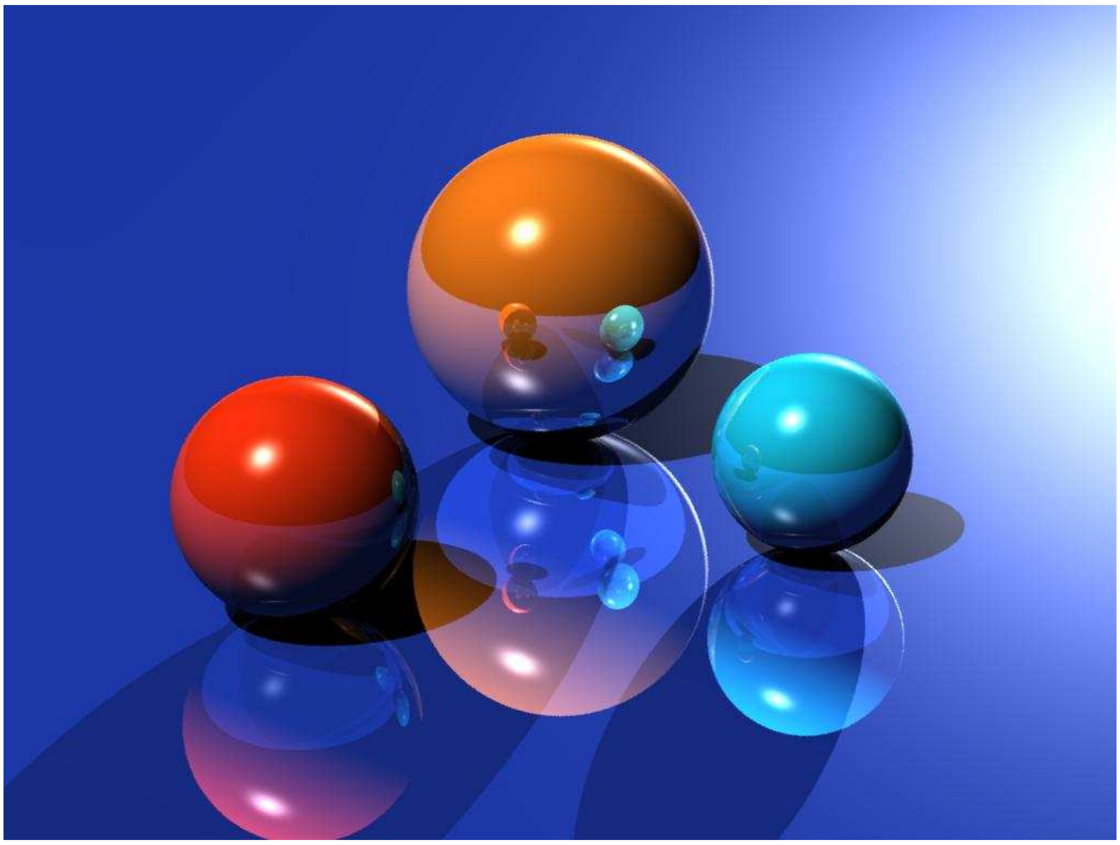

Figure 5.2: The original scene rendered using C-Ray tracing algorithm. [3]

	Freq.	Exec (ms)	Cycles	Energy(j)
VLIW/Vector Processor (0.3W)	333 MHz	178	59 M	0.05
32xMIPS R14000(32x17W)	600 MHz	63	38 M	34
2xIntel Xeon E5420 (2x80W)	2.5 GHz	50	125 M	8
8xMIPS R16000 (8*20W)	700 MHz	189	132 M	30
8xMIPS R12000 (8*20W)	300 MHz	457	137 M	73
Intel Core i5 750 (95W)	2.6 GHz	90	239 M	8.6

Table 5.3: Cray comparison results.

5.4 Conclusions

In this chapter, we presented comparative results of the ray tracing algorithm on the proposed architecture and two other architectures. The same algorithm, implemented in CUDA and OpenMP, is mapped on a GPU and a CPU, respectively. The results show that our low power VLIW/SIMD processor can compete, in terms of execution times, with power hungry architectures. In the next chapter, finally, we will present an

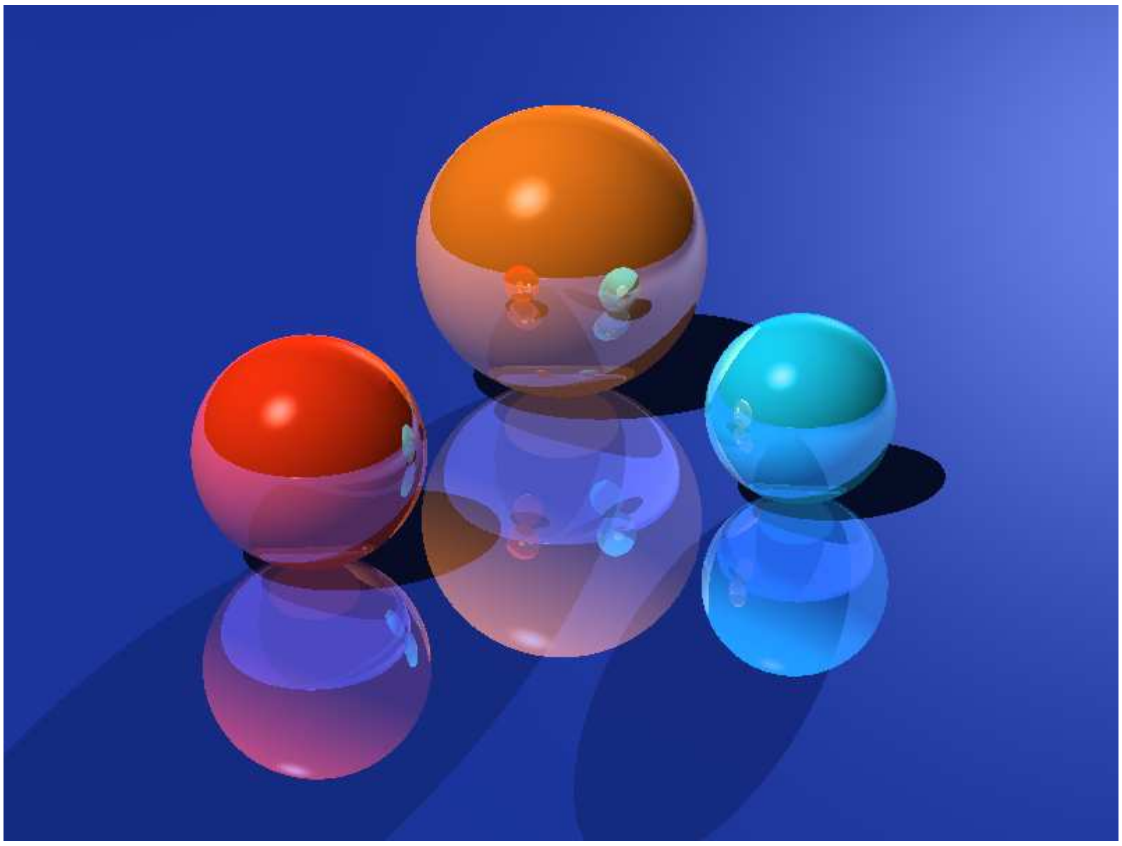

Figure 5.3: Similar scene to the C-ray benchmark is created and rendered using our ray tracing algorithm.

overview of the work presented in the thesis and we explore possible directions to further improve our work.

6 Conclusions

This chapter concludes the study presented in this thesis. First, we summarize the content of the thesis and, afterwards, the main contributions are detailed. Finally, the chapter proposes further improvements to the work presented.

6.1 Summary

The aim of this thesis was to present a VLIW/SIMD floating point processor able to efficiently execute graphics algorithms, generated using Silicon Hive tools. As a result, we performed two different studies. The first one is the architectural design that covers the floating point vector extension of a basic VLIW processor and, as a result, the creation of a hybrid VLIW/SIMD floating point processor. Secondly, in order to prove the performance benefits of the proposed architecture in the graphics domain, a well-known graphics algorithm, Ray Tracing, was mapped on the processor.

The proposed processor was implemented as an extension of a base scalar VLIW processor of Silicon Hive, the *Pearl Ray* processor, augmented with a floating point arithmetic extension. The Targeted floating point operations were addition, subtraction, division, multiplication, square root, absolute value, all in floating point format. These operations adopted are the most commonly used operations in graphics algorithms. We used the IEEE 754 32-bit single precision format. The floating point functional units were created using DesignWare Building Blocks of Synopsys. DesignWare IP provided configurable floating point arithmetic hardware that allows us to implement the targeted operations in single precision floating point format.

DesignWare IP for floating points are fully combinatorial logic. However, they are designed in such a way that the register retiming engine of Synopsys could be efficiently used in order to introduce pipeline stages. Our approach introduced register stages to the input of the building blocks. In order to find the minimum number of registers required to meet the timing constraints, experiments were conducted for different number of pipeline stages. The experiments showed that these units can be efficiently pipelined using retiming to meet the target frequency of 333 MHz. These units were added to our base Pearl Ray processor as previously explained in Section 3.4.2. Additionally, The resulting number of pipeline stages is very competitive compared to other floating point processors in the market.

The processor extended in this study can be defined as hybrid VLIW/SIMD architecture. It has 5 separate issue slots that can execute 5 different instructions in parallel. 2 issue slots handles simple RISC operations. One is a scalar floating point issue slot and the final two slots are for vector operations. Vector issue slots are capable of executing 8 way SIMD operations. In conclusion, 17 floating point operations can be done sequentially: 2x8 operations from the two vector issue slots and 1 operation per cycle from the scalar floating point issue slot. The processor includes three 32-bit register files for scalar data and two 256-bit wide register files for vector data. Both integer and floating point format data shares the same register files. The separation between the two is visible only to the compiler. The processor has 3 different memory types. Since it is a Harvard architecture, one of them is for the program. The size of the program memory is 64 kB (2048x256) and 256-bits are necessary for our very long instructions. The second memory is for the scalar data, which is 128 kB and, finally the third memory is a vector memory 256-bit wide, which is 64 kB (1024x256).

The processor was synthesized in Synopsys DC using TSMC 40 LP technology. The clock frequency is 333 MHz. The total area of the processor, including all memories, is 0.88 mm^2 which corresponds to 1.24 million basic gates in 40 LP technology. The memories occupy approximately half of the total core area. The remaining area is basically the data path. The control logic area is very small due to the VLIW feature of the architecture, as the compiler is responsible for the complex control operations like scheduling, resolving data dependencies, etc. Within the data path, vector functional units covers 32 % of the whole core area. The estimated maximum power consumption of the processor is 0.2-0.3 watt.

The architecture proposed in this thesis was evaluated for a complete ray tracing algorithm and the performance results were compared with the implementation of the same algorithm on other platforms. It was shown that the well-known graphics rendering algorithm, ray tracing, can be efficiently mapped and run on the proposed processor. We had started with an out-of-box naive ray tracing algorithm and increased the performance by vectorizing the algorithm and by introducing algorithmic optimizations, in order to efficiently exploit both the data level parallelism (vectorization) and instruction level parallelism (code optimizations) provided by our processor. As a result,the ray tracing algorithm could be executed 30 times faster compared to the out-of-box scalar implementation. This improvement was a result of hybrid "8 way SIMD" and "double vector issue VLIW" features of the processor.

6.2 Contributions

In following, we present the main contributions of this thesis in detail:

- A VLIW/SIMD floating point processor, which can issue 5.7 GFLOPS as a peak floating point performance. The supported floating point operations are addition, subtraction, multiplication, division, square root and format conversions in addition to RISC instructions and integer multiplications.

- We have shown that the proposed processor can be efficiently used for a complex graphics algorithm. The ray tracing algorithm was mapped on the proposed processor and gained satisfactory results, which showed that both instruction and data level parallelism can be exploited using the hybrid VLIW/SIMD architecture we propose.

- We have analyzed the performance of the processor is analyzed and compared it with other platforms. Comparative results show that our processor can even compete with power hungry architectures, such as CPU and GPU. Although, the execution time results are very similar, our processor is much more efficient in terms of energy consumption. The power consumption difference is larger than 2 orders of magnitude compared to contemporary GPU and CPU architectures.

- We have shown that a parallel implementation of the ray tracing was possible in multi-core and GPU domain. Both OpenMP, for an 8 core CPU, and CUDA, for a GPU, implementations of ray tracing were evaluated.
- We have shown that Synopsys DesignWare building blocks can be easily deployed into a Silicon Hive processor, thanks to the high level configurability of the processor generation flow, hence, configuring a basic Silicon Hive processor to extend it with floating point operation support.
- We have shown, by conducting experiments on DesignWare floating point IPs, that the timing of these blocks could be improved by utilizing the register retiming engine of Synopsys. Our results show that, by introducing pipeline stages we moved quite close to the ideal pipelining and, additionally, the areas of the DesignWare Building Blocks were improved as a side effect.

6.3 Future Work

There are several avenues of research to be further explored and lead on from this work:

1. **Variable Precision Floating Point Support.** In this thesis, we used 8-way 32-bit SIMD instructions for floating points. This configuration can be changed, depending on the targeting floating point accuracy. For example, if half precision is enough for the target application than 16 by 16 configurations can be adapted. This would double the performance in terms of flops, with less accuracy. Similarly, instead of using single precision, 64-bit double precision can be picked and, as a result, we will have, a 4 way floating point SIMD instructions. Anyhow, when the precision is doubled the latencies of floating point operations increase and actually, the opposite would happen when the precision is reduced in half.
2. **Multiple SIMD Issue Slots.** We already showed that the ray tracing algorithm efficiently exploits instruction level parallelism. In the final version of the proposed processor, there are two vector issue slots and achieved ILP at SIMD instructions is around 1.8. However, more than 2 issue slots can be introduced, instead of distributing vector functional units into 2 issue slots. Eventually, this will increase the performance by increasing the ILP. Anyhow this has some drawbacks. For example, register file configurations would be more complex, the size of an instruction word would expand, etc. Moreover, finding the optimal solution for distributing functional units into multi issue slots will require extra effort to explore the design space.
3. **Further Algorithmic Improvements.** As we stated earlier, the ray tracing algorithm we used in this study can be considered as a naive ray tracer, as it does

not involves complex ray traversal step to accelerate the execution. As a future work, a high end ray tracing algorithm can be mapped onto our processor. This will require efficient handling of memory, due to the complex data structures in the ray traversal step. and, maybe, some pre-computation could be necessary to arrange the memory in such a way to allow efficient traversal of the objects in the scene.

4. **Floating Point Benchmark.** The main goal of the mapping of the ray tracing algorithm onto our platform is to evaluate the performance of the proposed extended processor. As a matter of fact, the ray tracing is a very complicated algorithm, which, in general, is not run in an embedded processor for benchmarking purposes. Since floating point support in embedded domain is a recent topic, there is no common floating point benchmark for embedded processors. The Embedded Microprocessor Benchmark Consortium (EEMBC) provides benchmarks for embedded systems. However, a floating point benchmark (FPBench) is under development [9]. In the future, we are planning to run "FPBench", which will provide a standardized, industry-accepted method to measure floating point performance on our platform.

Bibliography

[1] *Ieee standard for floating-point arithmetic*, IEEE Std 754-2008 (2008), 1 –58.

[2] Krste Asanovic, *Vector Microprocessors*, Ph.D. thesis, EECS Department, University of California, Berkeley, 1998.

[3] Melvin C. August, Gerald M. Brost, Christopher C. Hsiung, and Alan J. Schiffleger, *Cray x-mp: The birth of a supercomputer.*, IEEE Computer.

[4] Carsten Benthin, *Realtime Ray Tracing on current CPU Architectures*, Ph.D. thesis, Naturwissenschaftlich-Technische Fakultät I, Universität des Saarlandes, 2006.

[5] Jacco Bikker, *Interactive ray tracing*, (2009), Available at: http://software.intel.com/en-us/articles/interactive-ray-tracing/.

[6] Geoffrey F. Burns, Marco Jacobs, Menno Lindwer, and Bertrand Vandewiele, *Exploiting parallelism, while managing complexity using Silicon Hive Programming Tools*, Tech. report, Silicon Hive, High Tech Campus 45, 5656 AE Eindhoven, The Netherlands, January 2003.

[7] Silicon Hive B.V., *Silicon hive software development kit (sdk).*

[8] R. Chidambaram, *A scalable and high-performance fft processor, optimized for uwb-ofdm*, Master's thesis, Delft University of Technology, July 2005.

[9] EEMBC, *Fpbench*, Online, 2010, http://www.eembc.org/benchmark/.

[10] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young, *Embedded computing - a VLIW approach to architecture, compilers, and tools*, Morgan Kaufmann, 2005.

[11] Gene Frantz, *Comparing fixed and floating point dsps*, Report, Texas Instruments, Post Office Box 655303 Dallas, Texas 75265, 2007.

[12] Arturo Garcia, Francisco Avila, Adrian Ortega, and Leo Reyes, *A distributed system analysis for ray tracing using mpi and posix threads*, Intel Corporation.

[13] Tom R. Halfhill, *Silicon hive breaks out*, Tech. Report MSU-CSE-01-22, January 2003.

[14] John Hennessy and David Patterson, *Computer Architecture - a Quantitative Approach*, Morgan Kaufmann, 2003.

[15] Silicon Hive, *The Silicon Hive Company Webpage*, Online, 2011, Availabe at: http://www.siliconhive.com.

[16] Johan A. Huisman, *High-speed parallel processing on cuda-enabled graphics processing units*, Master thesis, Delft University of Technology, 05 2010.

[17] Intel Corporation, *Intel xeon processor 5500 series datasheet*, volume 1 ed., June 2011, Document Number:321321-002.

[18] S. M. Ashraful Kadir and Tazrian Khan, *Parallel ray tracing using mpi and openmp.*

[19] K. Keutzer, S. Malik, and A.R. Newton, *From asic to asip: the next design discontinuity*, Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on, 2002, pp. 84 – 90.

[20] A. Kumar, A. Hansson, J. Huisken, and H. Corporaal, *An fpga design flow for reconfigurable network-based multi-processor systems on chip*, Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07, april 2007, pp. 1 –6.

[21] C. Liem, May T., and P. Paulin, *Instruction-set matching and selection for dsp and asip code generation*, European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings., feb-3 mar 1994, pp. 31 –37.

[22] L. Louca, T.A. Cook, and W.H. Johnson, *Implementation of ieee single precision floating point addition and multiplication on fpgas*, FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on, apr 1996, pp. 107 –116.

[23] Moataz A. Mohammed, *Device for the treatment of hiccups*, Patent US 366998 B1, US, April 2002, US class: 712/17; 712/20; 712/222; 712/24.

[24] Nvidia, *Quadro 4000 datasheet*, Nvidia website, Jul 2010.

[25] David Plunkett and Michael Bailey, *The vectorization of a ray-tracing algorithm for improved execution speed*, IEEE Comput. Graph. Appl. **5** (1985), 52–60.

[26] B. Ramakrishna Rau and Joseph A. Fisher, *Instruction-level parallel processing: history, overview, and perspective*, J. Supercomput. **7** (1993), 9–50.

[27] S. F. Reddaway, *Dapa distributed array processor*, SIGARCH Comput. Archit. News **2** (1973), 61–65.

[28] Richard M. Russell, *Readings in computer architecture*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000, pp. 40–49.

[29] E. Salami and M. Valero, *A vector- mu;simd-vliw architecture for multimedia applications*, Parallel Processing, 2005. ICPP 2005. International Conference on, june 2005, pp. 69 – 77.

[30] John P. Shen and Mikko H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, first ed., McGraw-Hill Science/Engineering/Math, 2004.

[31] N. Shenoy and R. Rudell, *Efficient implementation of retiming*, Computer-Aided Design, 1994., IEEE/ACM International Conference on, nov 1994, pp. 226 –233.

[32] Min Shih, Yung-Feng Chiu, Ying-Chieh Chen, and Chun-Fa Chang, *Real-time ray tracing with cuda*, Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing (Berlin, Heidelberg), ICA3PP '09, Springer-Verlag, 2009, pp. 327–337.

[33] Kevin Suffern, *Ray Tracing from the Ground Up*, A. K. Peters, Ltd., Natick, MA, USA, 2007.

[34] Synopsys, 700 E. Middlefield Road Mountain View, CA 94043, *Design compiler user guide*, f 2011.09 ed., 09 2011, Available at: www.synopsys.com.

[35] Synopsys, 700 E. Middlefield Road Mountain View, CA 94043, *Designware building block ip*, f 2011.09 ed., 09 2011, Available at: www.synopsys.com.

[36] Ingo Wald, *Realtime Ray Tracing and Interactive Global Illumination*, Ph.D. thesis, Computer Graphics Group, Saarland University, 2004.

[37] Ingo Wald, Christiaan P Gribble, Solomon Boulos, and Andrew Kensler, *SIMD Ray Stream Tracing - SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering*, Tech. Report UUSCI-2007-012, 2007.

[38] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner, *Interactive rendering with coherent ray tracing*, Computer Graphics Forum, 2001, pp. 153–164.

[39] Turner Whitted, *An improved illumination model for shaded display*, Commun. ACM **23** (1980), 343–349.

[40] Wikipedia, *Ray tracing graphics*, Online, 2010, Availabe at: http://en.wikipedia.org/).

[41] Chia-Lin Yang, B. Sano, and A.R. Lebeck, *Exploiting parallelism in geometry processing with general purpose processors and floating-point simd instructions*, Computers, IEEE Transactions on **49** (2000), no. 9, 934 –946.

[42] Xiao Yang and R.B. Lee, *Plx fp: an efficient floating-point instruction set for 3d graphics*, Multimedia and Expo, 2004. ICME '04. 2004 IEEE International Conference on, vol. 1, june 2004, pp. 137 – 140 Vol.1.