Conceptual Design and Realization of a Dynamic Partial Reconfiguration Extension of an Existing Soft-Core Processor

by

Philipp Kerling

A thesis submitted in partial fulfillment for the
degree of Master of Science

at
Technische Universität Ilmenau
Department of Computer Science and Automation
Software Architectures and Product Lines Group

Examiner: Dr.-Ing. Detlef Streitferdt
Supervisor: M.Sc. Michael Kirchhoff

February 12, 2019

DOI: 10.22032/dbt.38246
URN: urn:nbn:de:gbv:ilm1-2019200174

TECHNISCHE UNIVERSITÄT
ILMENAU

Abstract

Technische Universität Ilmenau
Department of Computer Science and Automation
Software Architectures and Product Lines Group

Conceptual Design and Realization of a Dynamic Partial Reconfiguration Extension of an Existing Soft-Core Processor

by Philipp Kerling

Many modern *field-programmable gate arrays* (FPGAs) support partial reconfiguration, which allows to dynamically replace only a part of a design at run time. In this thesis, partial reconfiguration capability is integrated with the *VHDL Integrated Softcore Architecture for Reconfigurable Devices* (ViSARD) developed at Technische Universität Ilmenau and conceived for hard real-time tasks requiring floating-point calculations with high precision. Specifically, its arithmetic logic unit is modified to allow exchanging floating-point arithmetic execution units. Design goals of the partial reconfiguration system are high speed, low latency, low resource overhead, and hard real-time capability. They are reached by implementing a custom partial reconfiguration controller loading partial bitstreams from external RAM over a standard AXI bus and extending the ViSARD appropriately. In a test design that switched between 3 different configurations each containing between 1 and 3 execution units, the proposed partial reconfiguration system achieved the maximum specified bitstream throughput on the target FPGA and allowed for roughly 40 % reduced look-up table usage.

Viele aktuelle Field Programmable Gate Arrays *(FPGAs) unterstützen die Technik der* partiellen Rekonfiguration *(PR), durch die dynamisch zur Laufzeit ein Hardware-Design auch nur teilweise ausgetauscht werden kann. Die vorliegende Arbeit integriert PR-Funktionalität in die an der Technischen Universität Ilmenau für harte Echtzeitaufgaben mit hochpräzisen Fließkommaberechnungen entwickelte* VHDL Integrated Softcore Architecture for Reconfigurable Devices *(ViSARD). Zu diesem Zweck wird die arithmetisch-logische Einheit angepasst, um das Auswechseln von Fließkomma-Ausführungseinheiten zu ermöglichen. Ziele der Entwicklung des PR-Systems sind hohe Geschwindigkeit, niedrige Latenz, niedrige Ressourcenkosten und harte Echtzeitfähigkeit. Erreicht werden diese durch die Umsetzung einer eigenen Steuereinheit (partial reconfiguration controller), die partielle Bitströme aus externem RAM über einen standardmäßigen AXI-Bus lädt sowie die entsprechende Erweiterung der ViSARD. In einem Testdesign, das zwischen drei verschiedenen Konfigurationen mit je zwischen einer und drei Ausführungseinheiten wechselt, hat das entwickelte PR-System den maximal spezifierten Bitstromdurchsatz auf dem Ziel-FPGA erreicht und den Verbrauch an Lookup-Tabellen um etwa 40 % verringert.*

Acknowledgements

First and foremost, I would like to thank Dr. Streitferdt for the opportunity to write my graduation thesis at his chair.

I would furthermore like to extend my earnest appreciation to my advisor, Michael Kirchhoff, who has shaped my understanding of the academic world just as much as this thesis. A lot of very fruitful discussion went into both the theoretical and practical aspects of this work. His input on the meaningful presentation of results was especially valuable.

My friend Kathrin has invested a substantial amount of time into proofreading and giving me precious advice on scientific writing and consistency. I am deeply grateful for her help.

Contents

List of Figures

List of Tables

Abbreviations

ACP accelerator coherency port

ALU arithmetic logic unit

APU application processor unit

ASIC application-specific integrated circuit

ASSP application-specific standard part

AXI Advanced eXtensible Interface

BRAM block random-access memory

CAP configuration access port

CLB configurable logic block

CORDIC coordinate rotational digital computer

CPU central processing unit

DAP debug access port

DDR double data rate

DDR3 SDRAM double data rate type three synchronous dynamic random-access memory

DMA direct memory access

DPR dynamic partial reconfiguration

DPRAM dual-port random-access memory

DPRM dynamic partial reconfiguration manager

DRAM dynamic random-access memory

DSP digital signal processor

ELF Executable and Linkable Format

EMIO extendable multiplexed input/output

eMMC embedded MultiMediaCard

EU execution unit

FAR Frame Address Register

FDRI Frame Data Register, Input

FF flip-flop

FIFO first in, first out

FPGA field-programmable gate array

FPU floating point unit

FSM finite state machine

HDL hardware description language

HWICAP Hardware Internal Configuration Access Port

I/D instruction/data

I/O input/output

IC integrated circuit

ICAP Internal Configuration Access Port

IDE integrated development environment

IEC International Electrotechnical Commission

IEEE Institute of Electrical and Electronics Engineers

ILA Integrated Logic Analyzer

IOB input/output block

IP intellectual property

ISO International Organization for Standardization

LC logic cell

LUT look-up table

MIO multiplexed input/output

MMC MultiMediaCard

MMU memory management unit

NaN not a number

OCM on-chip memory

OS operating system

PCAP Processor Configuration Access Port

PL programmable logic

PLD programmable logic device

PR partial reconfiguration

PRC partial reconfiguration controller

PS processing system

QSPI quad serial peripheral interface

RAM random-access memory

RISC reduced instruction set computer

RM reconfigurable module

ROM read-only memory

RP reconfigurable partition

RTOS real-time operating system

SD Secure Digital

SDK software development kit

SDRAM synchronous dynamic random-access memory

SoC system on a chip

SoM system on a module

SRAM static random-access memory

SSD solid-state drive

TFTP Trivial File Transfer Protocol

Torc Tools for Open Reconfigurable Computing

UART universal asynchronous receiver-transmitter

USB Universal Serial Bus

VHDL Very High Speed Integrated Circuit Hardware Description Language

ViSARD VHDL Integrated Softcore Architecture for Reconfigurable Devices

VLIW very long instruction word

XML Extensible Markup Language

Chapter 1

Introduction

Digital electronic systems have become an integral part of all kinds of devices ranging from small accessories to industrial machines, replacing and surpassing analog components in many domains. Their easier large-scale integration and superior resilience to noise allowed to overcome many challenges that were previously thought impossible with traditional analog electronics. Most devices require computation in some way, which is commonly performed by digital computers. Both analog and digital systems can theoretically offer an unlimited amount of computational precision, but in practice it is much easier to achieve high precision digitally, which is why analog computers are virtually irrelevant today.

Computers consist of one or more programmable processing elements, exemplified by the *central processing unit* (CPU) prevalent in personal and industrial computers. Depending on the application, these elements can vary fundamentally in their characteristics and implementation. Simple control systems use low-end processing elements that are fine-tuned to their requirements, while the CPUs of modern personal computers are very generic and can perform a great range of calculations extremely quickly.

As this example shows, there is plenty possibility for variation among processing elements and looking for the best option to solve a given problem is highly beneficial in terms of power usage, size, weight, and manufacturing cost of a device.

Contrary to big CPUs that are commonly produced as fixed-function chips to be placed on circuit boards (called *hard cores*), *soft-core* processors are primarily designed to be used on another class of integrated circuits called *programmable logic devices* (PLDs). Much like a processing element, PLDs can be programmed to perform any kind of computation, but at the hardware instead of the software level.

The benefits of using a PLD as opposed to a custom chip include increased flexibility and reduced cost when producing a small to medium number of devices only. The flexibility is a result of a characteristic of many PLDs: They can be configured not only once, but reconfigured as many times as needed. This reconfiguration may be limited to a part of the PLD and be performed while other parts continue operating, which is referred to as *dynamic partial reconfiguration* (DPR).

The goal of this thesis is to enable an existing soft-core processor design intended for a specialized set of applications (specifically: the *VHDL Integrated Softcore Architecture for Reconfigurable Devices* (ViSARD) developed at the Computer Architecture and Embedded Systems Group of Technische Universität Ilmenau) to make use of DPR by designing and implementing an extension to the processor that swaps out *execution units* (EUs) which perform specific arithmetic operations such as addition, multiplication, or division. It is conceivable that at least some algorithms may be split (naturally or by reordering independent operations) into stages in which a distinct set of mathematical operations is required. For example, one part of an algorithm may need to do a lot of multiplications, while another part may not require any multiplication at all and instead has to calculate the sine of many values. With the current static design of the ViSARD, all EUs that the program needs at some point in its calculations must be present at all times, meaning that they always occupy space in the PLD the core runs on.

With *partial reconfiguration* (PR), only the EUs that the current part of the program needs must be available. It is unknown whether this optimization of the ViSARD leads to significant resource/area savings in practice, which is why this thesis will evaluate how equivalent PR and non-PR designs compare in terms of device utilization.

In order to maximize the benefit of partial reconfiguration, special weight is put on the efficiency of the proposed solution. Concretely, the PR system design

- must require a minimal amount of additional resources for implementing reconfiguration so they do not cloud the potential savings,
- must complete reconfiguration as fast as possible to allow for rapid switching between sets of EUs,
- must not compromise the hard real-time capability of the ViSARD by the introduction of PR,
- must be able to run on the hardware available for testing (the Trenz Electronic GigaZee containing a Xilinx Zynq-7000 *field-programmable gate array* (FPGA)), but be easily adaptable to other platforms and
- should be reasonably uncomplicated in order to facilitate further research.

These goals will guide the design, the implementation, and the test setups and the interpretation of their results. In detail, the thesis is structured as follows:

Chapter 2 introduces the main building blocks that all further work is built on. These are primarily FPGAs, the ViSARD, and more high-level topics of digital systems design that are relevant to the design and the implementation of the proposed partial reconfiguration system are explained.

Building upon this information, Chapter 3 describes both the final system and how it was reached including the evaluated alternatives. Main subjects are the *partial reconfiguration controller* (PRC) performing the reconfiguration and how PR was integrated into the ViSARD.

In order to consider the practical implementation of the proposed design, Chapter 4 goes on to explain how an actually working system was achieved based on the considerations in the preceding chapter. Besides the hardware components, utility programs developed for use on the computer of the designer that make the realization of PR-based ViSARD designs feasible are introduced.

Chapter 5 presents tests of the PRC and the whole system in simulation and on hardware by describing both their setup and their results. These tests allow conclusions to be drawn about the resource savings achieved by PR and whether the goals as outlined above were reached. Furthermore, different bitstream generation methods (module-based and difference-based) are compared in terms of their advantages and disadvantages and resulting bitstream sizes.

Finally, Chapter 6 sums up the main findings and contributions of the present research and offers some possibilities for further consideration.

Floating-point numbers in this thesis follow the IEEE-754 standard [IEE08] and are always given in double precision (64 bits). The ViSARD itself allows selectable single or double precision, but only the latter is evaluated here as far as PR is concerned.

Multiples of byte units follow the international standard ISO/IEC 80000-13 [Int11] in combination with ISO/IEC 80000-1 [Int08], so for example 1 *kB* (kilobyte) equals 1,000 bytes, while 1 *KiB* (kibibyte) equals 1,024 bytes.

Chapter 2

Fundamentals

The purpose of this chapter is to introduce the devices and techniques used in this thesis, concentrating on the aspects that are fundamentally important: The first section will be about *field-programmable gate arrays* (FPGAs) (the type of device that this thesis is implemented on) and their components. After that, selected topics of digital hardware and systems design will be presented before giving an in-depth explanation of partial reconfiguration and finishing the chapter with a description of the *VHDL Integrated Softcore Architecture for Reconfigurable Devices* (ViSARD). Since it is not possible to describe everything from the ground up within the scope of this master's thesis, the reader is assumed to be familiar with the basic concepts of digital electronics such as Boolean logic.

2.1 FPGAs

Almost all modern electronic devices consist of one or more silicon *integrated circuits* (ICs). These are typically off-the-shelf components that are produced in large quantities in order to be profitable. They can have very generic functions such as in the case of a microprocessor or be designed for a narrower purpose, which makes them *application-specific standard parts* (ASSPs). Additionally, *application-specific integrated circuits* (ASICs) are specially constructed for one specific application and usually manufactured in smaller batches, so the production costs are very high. Prototyping new circuits is especially expensive when multiple iterations of troubleshooting and production are needed. In order to allow for a more flexible approach to development and manufacturing, simple *programmable logic devices* (PLDs) were sold as an alternative in as early as the 1970s. FPGAs are the most advanced type of PLDs and can be summarized as "digital ICs that contain configurable (programmable) blocks of logic along with configurable interconnects between these blocks" [Max08]. [Max08; Tar17]

2.1.1 Overview

At the highest level, FPGAs consist of a "regular array of basic programmable *logic cells* (LCs) and a programmable interconnect matrix surrounding the logic cells" [Gro08]. Additionally, the pins of the device are connected to programmable *input/output* (I/O) cells that allow them to be used with various I/O standards for extra flexibility in communicating with other parts of a hardware system. This basic architecture is shown in Figure 2.1. Many additional components exist that will be explained in detail below.

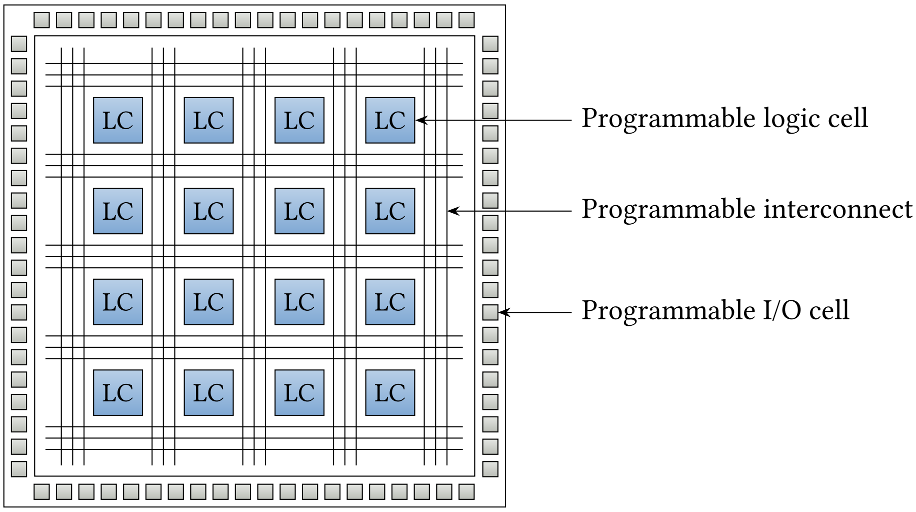


Figure 2.1: Generic FPGA architecture. The programmable interconnect provides flexible connections between the logic cells and to the I/O cells. Additional components omitted for clarity. Adopted from [Gro08, fig. 1.16].

The logic elements can be connected together to form a circuit that realizes arbitrary functions when programmed accordingly. This is conceptually similar to a microprocessor which can likewise perform a wide range of functionality but is limited to sequential execution of a piece of software and a specific set of instructions that can be processed. In contrast, an FPGA operates at a much lower abstraction level with inherent parallelism, leading to better performance. This means that developing designs is vastly different from software programming and in many cases more complicated. In fact, microprocessors themselves can be implemented on FPGAs and it is common to do so in order to combine the benefits of both concepts. [Ker15; Max08]

Various companies are presently manufacturing FPGA devices, with the most relevant of them being Xilinx and Intel (formerly Altera). Naming of components varies between them: Xilinx, for instance, calls their logic cell equivalent *slice*, while Intel calls their equivalent *adaptive logic module* (ALM). Due to the hardware available for implementation, this thesis will focus on Xilinx devices. [Tar17; Xil16a; Max08]

FPGAs are used e.g. for faster and cheaper prototyping of ASICs, but in modern electronics development they are not limited to that. Owing to their flexibility, low cost (compared to ASICs), and short time-to-market, they can be found not only in low-volume order-made devices and systems, but also in end-user products. Typical applications include communication devices, automotive equipment like driver assistance systems, and other systems that require high-performance digital signal processing. Compared to *central processing units* (CPUs), FPGAs are often used to implement algorithms and systems that would be difficult or too expensive to realize in software in an efficient manner. [Max08]

2.1.2 Components

The following subsections will offer a rudimentary introduction to major components of FPGAs. A concise, but comprehensive description of these devices, the associated trade-offs and design flows can be found in the excellent book "FPGAs: Instant Access" by Maxfield [Max08], for example.

Logic cells

The basic building block of any FPGA, the logic cell, consists of at least one multi-input *look-up table* (LUT) that has one *flip-flop* (FF) register connected to it. A LUT is a simple digital logic element that is programmed with the desired output value for every possible combination of input values, so it can be used to realize arbitrary combinatorial binary logic functions. The flip-flop is a storage element that samples its input value at a clock edge and outputs it steadily for the duration of the whole clock cycle, even when the input changes intermittently. Via the routing fabric of the FPGA, the output of the flip-flop can in turn be connected to the input of another LUT. The combination and interconnection of these elements facilitates the building of arbitrary synchronous logic circuits such as state machines. Other architectures (e.g. using multiplexers instead of LUTs) are possible, but not discussed here. [Max08]

FPGA manufacturers usually build a hierarchy using these basic elements. Xilinx slices in 7 series devices, for example, contain 4 6-input LUTs and 8 flip-flops. Two slices are combined to form one *configurable logic block* (CLB). Slices also contain programmable multiplexers to allow for e.g. bypassing the flip-flop in the output path. [Xil16a]

Embedded memory

For storing larger amounts of data that would need too many flip-flops, the FPGA may include embedded *random-access memory* (RAM) modules called *block random-access memory* (BRAM). They can be combined to form larger memories and are typically realized as *static random-access memory* (SRAM), so they do not have any access latency and are very fast. Their number and size varies depending on the manufacturer and architecture of the device. [Max08]

Embedded multipliers/adders

Furthermore, advanced FPGAs will also contain embedded binary multipliers and adders as separate hardwired components. The reason for this is that realizing these functions in programmable logic is inherently slow. These are mainly useful for digital signal processing applications, which is why these blocks are called *digital signal processor* (DSP) slices at least by Xilinx. [Max08; Xil18a]

Embedded processors

Virtually every algorithm can be implemented in hardware (as digital logic), in software (as instructions run on a microprocessor), or using a mixture of both. The decision of which option to use to implement a given task typically requires, inter alia, a trade-off involving criteria such as performance requirements, economical constraints, and developer skills. In many cases, it is beneficial to combine digital logic and processors in order to get the best of both concepts. [Max08]

FPGA suppliers try to make it as easy as possible to take advantage of this combination without increasing system complexity too much by integrating embedded microprocessors in their devices. This is achieved in the following two ways:

- *Hard-core processors* are hardwired CPU cores integrated in FPGAs as separate blocks adjacent to or inside the fabric of programmable logic. They are mostly included in more expensive high-end chips. If the core is not a component of the FPGA, but plays a considerable role in the overall system, the device is referred to as *system on a chip* (SoC). [Max08]
- *Soft-core processors* are instantiated by the designer as entities in the programmable logic design. They get translated to FPGA resources like LCs and interconnect together with the rest of the design. Their performance and efficiency are usually inferior to equivalent hard cores, but they can be used in every project independent of the concrete FPGA chosen as long as it has enough resources. Another advantage is the possibility to adapt the soft core to the specific requirements of the use case, which is not possible with a hardwired CPU. As a consequence, it can be more efficient to use a specially parametrized soft core as opposed to a general-purpose hard core. The cores themselves may be provided by the FPGA supplier or obtained separately. [Max08]

Routing resources

Interconnections between the logic elements are provided by the routing fabric of the FPGA. A typical structure can be seen in Figure 2.2. Routing resources are arranged in horizontal and vertical lines, forming a grid with LCs inside it. The interconnect boxes contain programmable switch matrices that can form connections between the lines entering and exiting on each side with varying degrees of flexibility according to the concrete FPGA architecture. This way, connections across the whole device are possible. [FMM12]

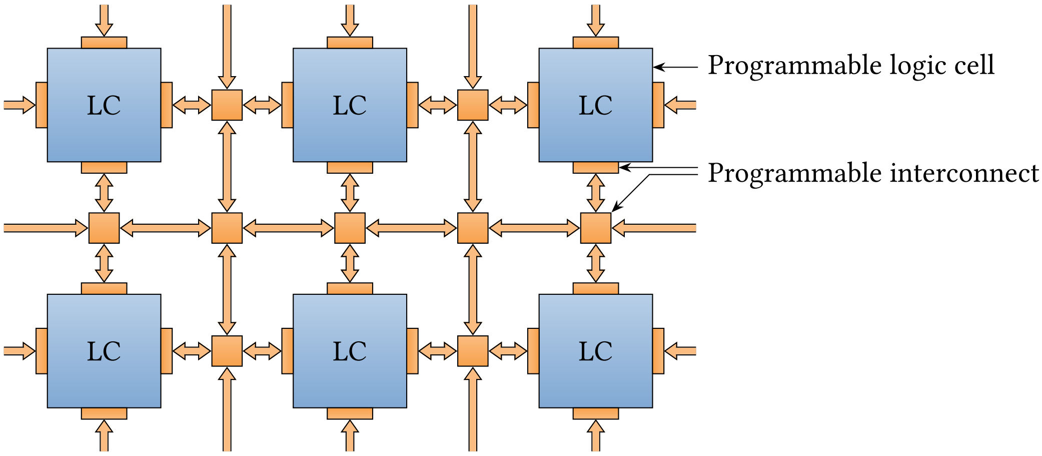


Figure 2.2: FPGA fabric with logic cells and interconnect (zoomed-in version of Figure 2.1). Adopted from [Max08, fig. 2-1].

Utilization

To any designer, the amount of resources used on the FPGA is of great importance. If the device is not utilized to a satisfying degree, it effectively means that money could be saved by choosing a smaller one. The prime measurement for resource utilization is the absolute amount or percentage of component instances used by a given design. The components most interesting to consider for this were outlined above: logic slices (possibly broken down to LUTs and flip-flops), BRAM, and DSP slices. Due to timing restrictions and the limited availability of routing resources, a logic utilization of 100 % is not practical [DeH99].

2.1.3 Configuration

The programmability of FPGAs is realized by configuration cells inside the components that e.g. indicate to a LUT to which inputs it should respond with a logic one or to a routing matrix which lines to connect. This section will introduce both the general concept and specifics of Xilinx 7 series devices, since the FPGA this thesis was implemented on belongs to this family.

Overview

Most commercially available devices use configuration cells based on SRAM technology, allowing for an unlimited number of reconfigurations. The downside is that the configuration is volatile, i.e. it must be programmed into the FPGA after each power cycle. [Max08]

A dedicated *configuration controller* on the device accepts configuration data from inside or outside the chip and feeds it into the SRAM cells inside the device (Figure 2.3). These cells are typically organized in larger registers, so they cannot be accessed individually, but only in chunks. Modern FPGAs contain millions of such cells. Conceptually, an FPGA can be imagined to contain two layers: A functional layer consisting of all logic cells, interconnect, and functional units such as DSPs and below it a configuration layer consisting of 1-bit cells defining the operation and interconnection of the functional layer. The configuration data for SRAM devices is called *configuration bitstream* and is encoded in supplier/device-specific proprietary formats. [Max08; VF18; BLC07]

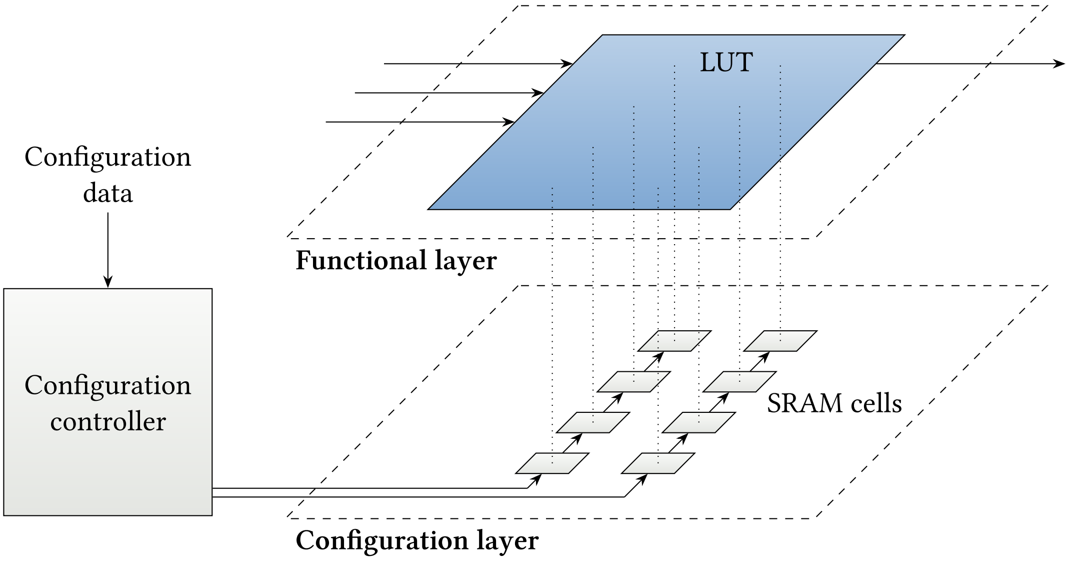


Figure 2.3: Relation of FPGA functional and configuration layer. The exemplary 3-input 1-output LUT on the functional layer requires one configuration cell per possible input combination for a total of $2^3 = 8$ cells on the configuration layer to fully define its behavior. The configuration controller can usually access these cells in chunks, but not individually.

Specifics of Xilinx 7 series devices

Logic elements in Xilinx 7 series FPGAs are tiled into 1 to 24 *clock regions* (depending on the device) that each contain a grid of CLBs 50 units high. Every clock region is interspersed with configuration cell register chains that run vertically through them from one end to the other (the direction may vary). One such register is the smallest addressable unit in the configuration memory of the FPGA and is called a *configuration frame*. All frames have the same length of

3,232 bits (404 bytes), but may contain different information such as LUT configuration, BRAM content, or routing matrix data.

The layout of different types of logic components within a clock region is columnar, with columns of DSP (each 20 units high) and BRAM (each 10 units high) resources inserted into the CLB grid. Additionally, interconnect resources for routing usually exist in-between every pair of columns. The exact configuration differs between clock regions and devices, but columns always run through the whole FPGA in the vertical direction, across clock regions. Figure 2.4 shows an exemplary column layout for illustration. [Xil18b; Xil18c; VF18]

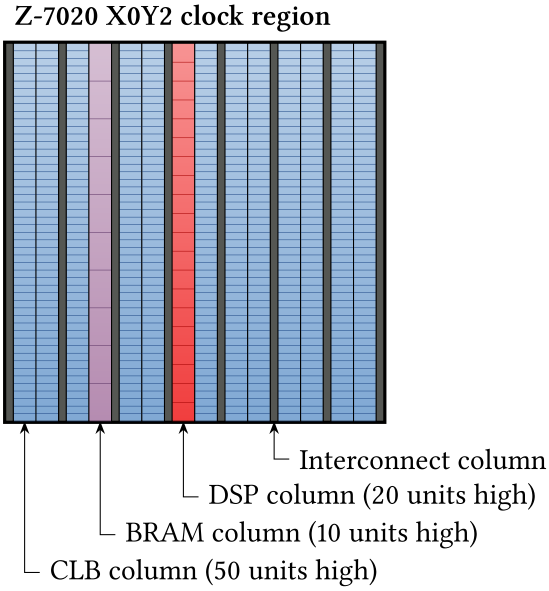

Z-7020 X0Y2 clock region
Interconnect column
DSP column (20 units high)
BRAM column (10 units high)
CLB column (50 units high)

Figure 2.4: Column layout inside the X0Y2 clock region of the Z-7020 FPGA (exemplary).

The format of the bitstream data is rudimentarily documented in [Xil18c]. It consists of 4-byte command words instructing the configuration controller to read or write one of its registers optionally followed by the data to write, again as 4-byte words. To program all configuration cells in the most basic fashion, the *Frame Address Register* (FAR) is set to zero (i.e. first frame in first clock region) and all configuration data is written to the *Frame Data Register, Input* (FDRI) in one batch. The controller will automatically cycle through all configuration cells in an undocumented order without requiring the FAR to be updated. To program a specific part of the device, the FAR register can instead be set to the address of the frame to write.

2.2 Digital hardware and systems design

Since the basic description of the hardware is concluded, the introduction moves one level of abstraction higher to the logic and systems to be implemented on top of FPGAs. This section begins with an outline of important concepts in digital systems design that are required for the understanding of this thesis. Most of these topics are introduced in “FPGAs: Instant Access” [Max08]. More details regarding system level design and related topics are available in “Digital Systems Design with FPGAs and CPLDs” by Grout [Gro08]. Subsequently, the hardware used for implementation is introduced both on the device level (the Xilinx Zynq-7000) and the board level (the Trenz Electronic GigaZee).

2.2.1 Key concepts and principles

Hardware description languages

For developing hardware designs for PLDs, it needs to be possible to describe the functionality of digital electronic circuits using a formalized language – due to the enormous size and complexity of FPGAs, manually configuring every logic slice is not feasible for all but the most simple designs. Every language that is primarily intended not for software development but for designing hardware is called a *hardware description language* (HDL). The two main alternatives in this field are Verilog-HDL and *Very High Speed Integrated Circuit Hardware Description Language* (VHDL), of which only the latter will be used here. The reader is assumed to be familiar with at least the basic concepts of VHDL. A practically oriented introduction to the language offers Kafig, for instance, in [Kaf11], while a thorough discussion of all aspects can be found in “The Designer’s Guide to VHDL” [Ash08] by Ashenden. [Ker15; Gro08]

This thesis makes use of features added in the VHDL-2008 revision of the language standard as they allow for more concise and consistent code.

Design flow

As the HDL code cannot be directly put on the FPGA, the *design flow* when working with systems including FPGAs requires a transformation of this code into valid device-specific configuration data (Section 2.1.3). Major steps of this process (as seen in Figure 2.5) are: *Synthesis* which takes the HDL files and produces a *netlist* describing logic gates and their interconnections implementing the desired functionality, *implementation* which *places* the gates into logic cells and *routes* the required connections, and *bitstream generation* which produces the final configuration to program the actual FPGA device with.

Pipelining

An important concept to understand when working on hardware designs is *pipelining*. As combinatorial functions get more complex, they will span more and more LUTs and slices. Every

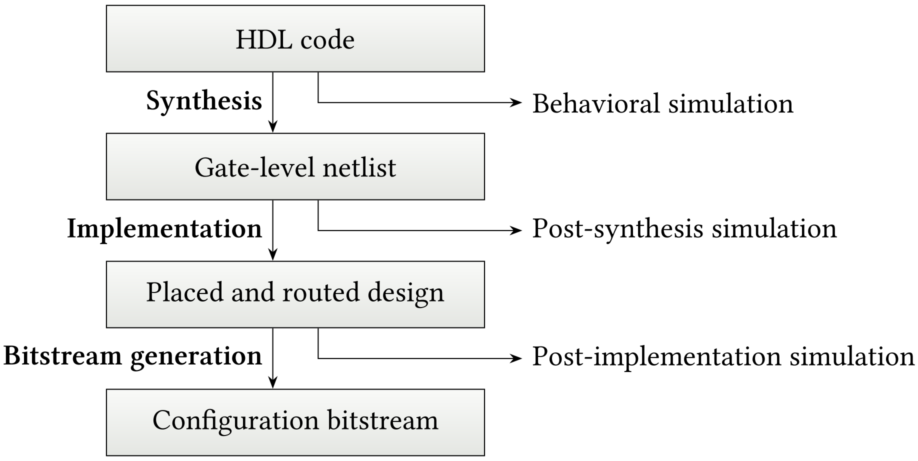


Figure 2.5: Major steps (synthesis, implementation, and bitstream generation) in a typical FPGA design flow, starting with code and ending with a bitstream to program the device. Simulation is possible after each step. Inspired by [Smi10, fig. 3-10; Gro08, fig. 1.20].

connection between those elements and every pass through a basic element such as a LUT takes some time and adds up to the total delay of the circuit, i.e. the time difference between changing an input value and the correct response appearing on the final output. The maximum frequency for which the combinatorial function can be correctly performed is approximately inversely proportional to this delay. [Max08]

Pipelining means that additional register (flip-flop) stages are inserted between stages of combinatorial logic. This breaks up the logic into smaller parts that can operate more efficiently and at higher maximum frequency, but it adds to the latency of the circuit. It will take as many clock cycles as there are stages until the output corresponding to a specific input value is ready to be consumed. Similarly to a car manufacturing line, a new operation can be started in every clock cycle, so the throughput is still high [Max08]. This technique can only be applied to combinatorial functions that have distinct separable stages. FPGA manufacturers such as Xilinx highly recommend pipelining, since it is cheap (their FPGAs include lots of flip-flops in slices anyway) and improves performance [Xil16a].

Intellectual property

As in software programming, it is common to use preexisting functional blocks in hardware designs. These are referred to as *intellectual property* (IP) or IP cores. They can for example be offered by companies for purchase, be included with the design suite of the FPGA supplier, or be available for free on the internet on web pages such as OpenCores [Oli18]. Furthermore, core generators may allow customizing the IP for the respective application by including or excluding certain features or allowing to set performance characteristics such as latency. [Max08]

Bus systems

Connecting functional blocks inside an FPGA or SoC would be very difficult without some standard protocols for communication. This is even more important for IP cores, since they need to be used in a variety of applications and system designers cannot be expected to learn a new method of communication for every core. Standard on-chip bus systems include *Advanced eXtensible Interface* (AXI) [ARM11] and AXI4-Stream [ARM10] by ARM Limited and Wishbone by OpenCores [Ope10]. Xilinx has decided to adopt AXI buses in their products (devices as well as IP) [Xil12].

In AXI terminology, each bus has a *master* and a *slave* side, with the master issuing read and write commands and the slave answering. Reading and writing uses separate channels, and each of these channels is further divided into an address and a data channel, plus one response channel for the writing direction for a total of 5 channels. The reading part of the interface is shown in Figure 2.6. The master initiates read transactions by specifying the initial read address and additional transfer parameters and the slave answers with the requested data (or an error indicator). The width of the address and data signal vectors can be chosen according to the application: Address width is arbitrary, while data width is limited to powers of two between a minimum of 8 and a maximum of 1,024 bits. [ARM11]

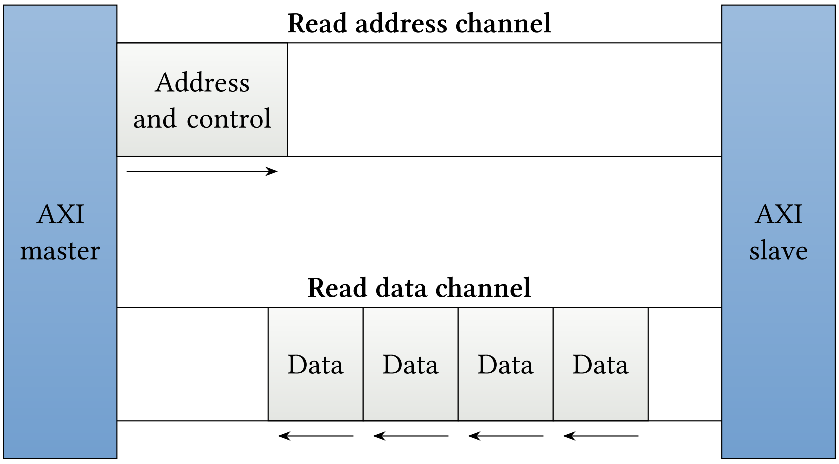


Figure 2.6: AXI read channel architecture. The master sends read requests on the read address channel which are answered by the slave in the form of (potentially multiple) data replies on the read data channel. Adopted from [ARM11, fig. A1-1].

Reads and writes are performed in bursts, i.e. each transaction has a burst length associated with it that specifies how many successive transfers are requested. In many cases, this is more efficient than requesting each transfer separately due to the reduced amount of round-trips and increased opportunity for pipelining and buffering. All channels include handshake signals that indicate when the source is sending valid data and when the sink is ready to accept it. The master does not need to wait for the response to a transaction to arrive to issue the next one, it can immediately signal a new request that the slave will process when ready, possibly in parallel to other ones. [ARM11]

Real-time systems

Practical systems, especially in digital signal processing applications, often have requirements concerning the time the system is allowed to take to react or perform certain operations. A system capable of achieving this is called a *real-time system.* Predictability of all components involved is key to achieve this, but fulfilling this may reduce the performance of the system on average.

Depending on the consequences that missing the deadline for finishing a real-time task would have, they are further categorized. Missing the deadline of a *hard real-time* task is most severe and will cause the system to behave erroneously, while the delayed response of a *firm real-time* task means that the result cannot be used any more. The third and final category of *soft real-time* tasks produces results that would at most cause decreased performance if not available by their deadline. Examples of systems with hard real-time requirements are safety-critical devices such as brake control in automotive vehicles or measurement instruments that need to process incoming sensor data without losing any information. [But11]

2.2.2 The Zynq-7000 SoC

The Xilinx Zynq-7000 family, to which the device used for practical implementation in this thesis belongs, is comprised of SoCs that include an ARM Cortex-A9 based hard-core processor and a Xilinx 7 series based FPGA, with the ARM core being the central component and the FPGA acting as slave. The processor (also referred to as *application processor unit* (APU)), peripherals, and bus interconnect including memory controllers form the *processing system* (PS). The FPGA fabric and all supporting circuitry such as clock managers and the configuration module are in entirety called *programmable logic* (PL). [Xil18r]

Overview

A high-level overview of the SoC is shown in Figure 2.7. The FPGA is programmed by writing bitstream data to the PL configuration module via the *Processor Configuration Access Port* (PCAP) from the PS. Access to the module is also possible from the logic fabric using the *Internal Configuration Access Port* (ICAP). Furthermore, a *dynamic random-access memory* (DRAM) controller is included in the PS to easily access external *double data rate* (DDR) memory from both parts of the SoC. Communication between them is possible using a number of dedicated

AXI buses. Moreover, both the PS and PL include their own separate I/O blocks and associated device pins, called *multiplexed input/output* (MIO) for the PS and SelectIO for the PL. A part of the functions of the MIO pins can be redirected to the PL using the *extendable multiplexed input/output* (EMIO) interface, but the reverse is not possible. [Xil18s]

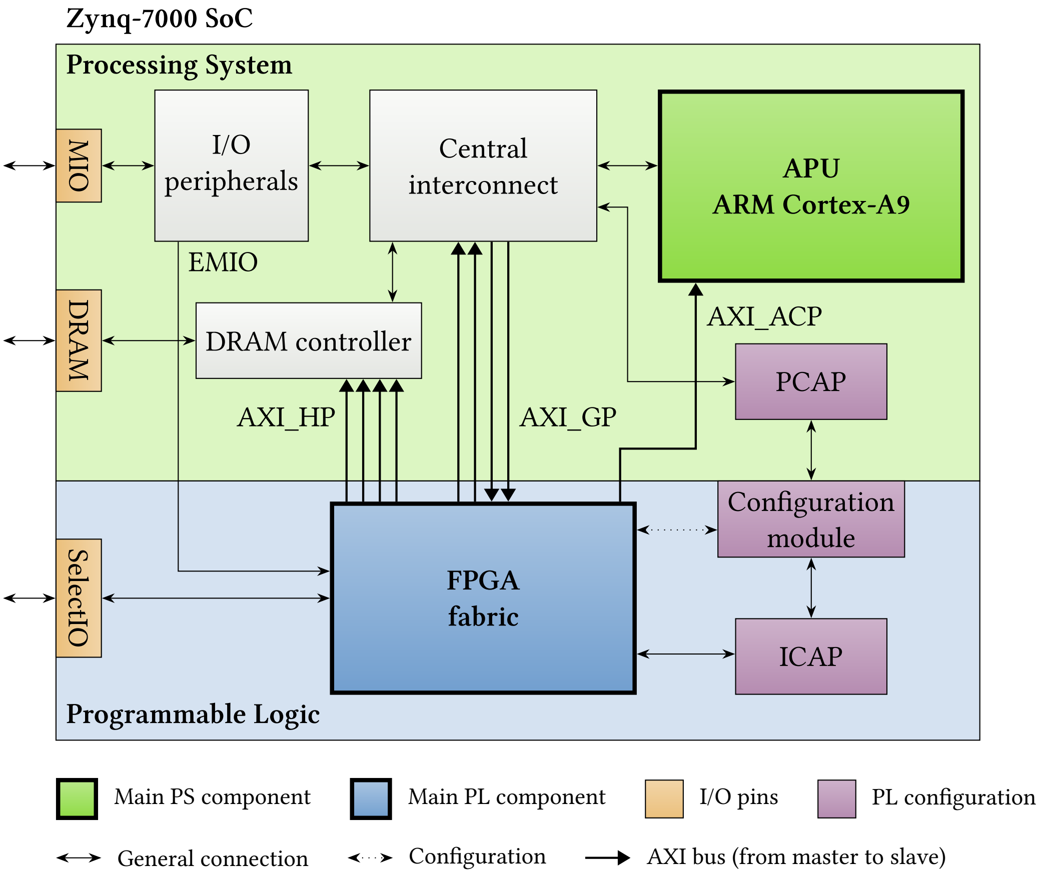


Figure 2.7: Zynq-7000 SoC overview highlighting interconnections between processing system and programmable logic. APU: Application processor unit, EMIO: Extendable multiplexed input/output, ICAP: Internal Configuration Access Port, MIO: Multiplexed input/output, PCAP: Processor Configuration Access Port. Adopted from [Xil18s, fig. 1-1] (reduced to components relevant for this thesis).

In detail, the AXI buses crossing between the PS and PL part of the SoC are (as seen from the PL side, from left to right in Figure 2.7): [Xil18s]

- 4x AXI_HP: 32-bit or 64-bit high performance/bandwidth master ports that can access only the DRAM controller and a small amount of *on-chip memory* (OCM) inside the APU,
- 4x AXI_GP: 32-bit general purpose ports (2x master, 2x slave), and
- 1x AXI_ACP: 64-bit cache coherent master port (*accelerator coherency port* (ACP)).

Xilinx recommends to use the AXI_HP port for "high performance *direct memory access* (DMA) for large datasets", the AXI_ACP for "high performance DMA for smaller, coherent datasets", and the AXI_GP port for "PL to PS control functions [and] PS I/O peripheral access" [Xil18s].

Development

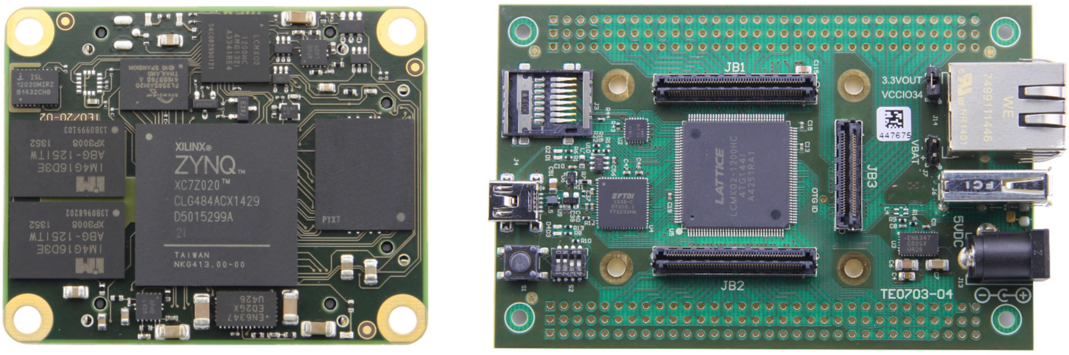
The digital logic development solution offered by Xilinx for all of their recent products starting with 7 series FPGAs and the Zynq-7000 SoC is the Vivado *integrated development environment* (IDE), part of the Vivado Design Suite. In addition to the hardware design, systems including the Zynq-7000 have to consider the software that is to run on the APU. At the very least, the PS is responsible for (initially) programming the PL. The Vivado Design Suite includes a separate *software development kit* (SDK) environment called Xilinx SDK (XSDK) including a complete toolchain (C/C++ compiler, debugger, utilities, and libraries) and an IDE based on the open-source Eclipse C/C++ Development Toolkit. [Xil18n; Xil15b]

One of the first and main points to consider is what kind of *operating system* (OS) to run on the processor. Generally speaking, the options are to run a bare-metal system (i.e. without any OS), a *real-time operating system* (RTOS), or a fully-fledged Linux OS. Xilinx facilitates each of these options by e.g. providing low-level drivers and an SDK for developing bare-metal applications and Linux drivers for easily using a high-level operating system. Using Linux will typically allow for easier development due to the abundance of available libraries and tools, but it does not directly support real-time constraints and increases system complexity. [Xil15b]

Using the Xilinx SDK environment, even applications running without an OS can be created with little effort. If the Zynq-7000 was configured in the hardware design in the Vivado design suite by way of the Processing System IP [Xil17e], the SDK will automatically pick up all important parameters such as the peripherals enabled and their memory mappings, so the designer does not have to set up anything manually. It will also generate boot loader code for initialization of the PS and its controllers and peripherals after power-up. The boot loader can be programmed to the start-up flash memory on the system board. [Xil15b]

Hardware used in this thesis

The hardware available for implementation in the present thesis is the TE-0720 GigaZee *system on a module* (SoM) by Trenz Electronic (Figure 2.8a) in combination with a TE-0703 carrier board (Figure 2.8b). The main module is equipped with a Xilinx Zynq-7000 family Z-7020 SoC, a Gigabit Ethernet transceiver, a 32 MB *quad serial peripheral interface* (QSPI) flash memory for storing the boot loader and system software, a 4 GB *embedded MultiMediaCard* (eMMC) for



(a) TE0720 SoM (top view). Source: [Treb]

(b) TE0703 carrier board (top view). Source: [Trea]

Figure 2.8: Hardware used in this thesis: TE0720, to be mounted on top of TE0703 (scale differs)

additional data storage, and 1 GB of *double data rate type three synchronous dynamic random-access memory* (DDR3 SDRAM). The FPGA inside the Z-7020 is based on the low-end Artix-7 series and provides 53,200 LUTs, 106,400 flip-flops, 4.9 Mb of BRAM, and 220 DSP slices. Power supply to the SoM and connection to Xilinx programming tools via a *Universal Serial Bus* (USB) cable are provided by the carrier board. [Tre17; Tre16; Xil18r]

The Z-7020 PL is divided into 6 clock regions as seen in the device overview in Figure 2.9. The layout of logic columns is not completely homogeneous across the device due to the differing sizes of the clock regions (see Figure 2.4 for the layout of the X0Y2 clock region as an example).

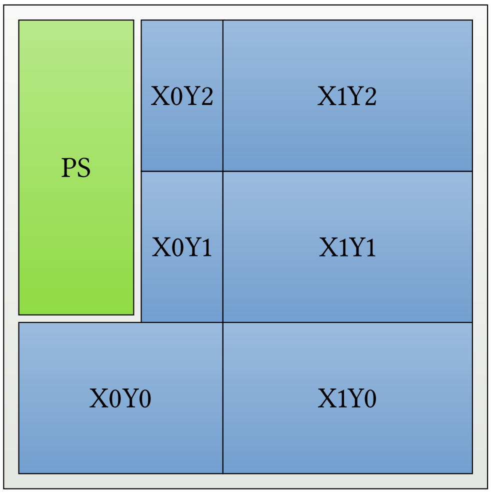


Figure 2.9: Overview of Z-7020 SoC used in this thesis. The programmable logic is divided into 6 clock regions. The upper left corner is taken up by the processing system.

2.3 Partial reconfiguration

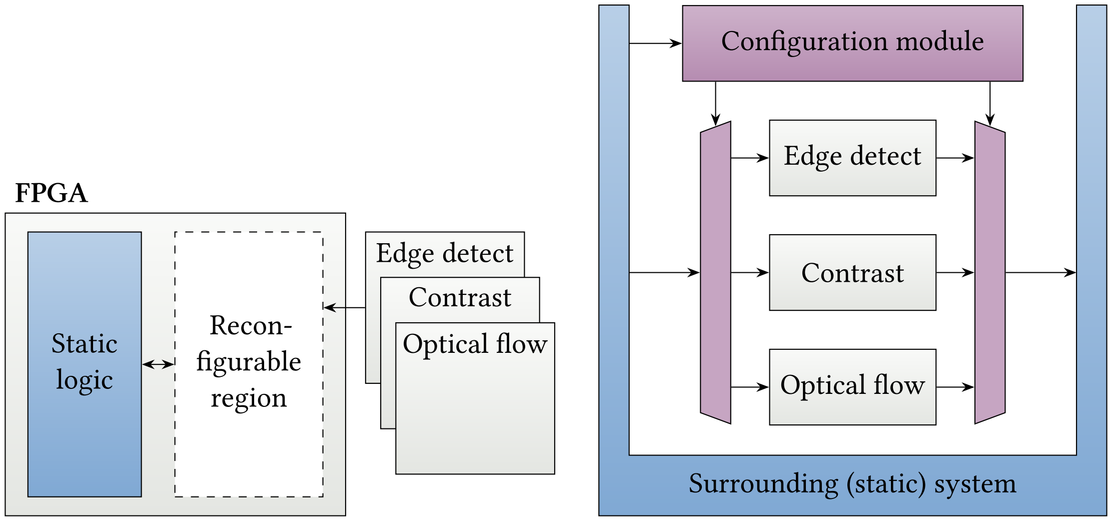
Configuration of FPGAs was already briefly explained in Section 2.1.3 when introducing their components, but there is a more advanced technique called *partial reconfiguration* (PR) that is a fundamental building block of this thesis. This section introduces the principle of reconfiguration including its further classification and focuses on partial reconfiguration, starting with its benefits and use cases. After discussing how this technique applies to Zynq-7000 devices, important considerations for PR designs are listed. For further reading and gaining a deeper understanding of this topic, a comprehensive book on this topic written by Koch in 2013 [Koc13] is recommended. A compact overview of the historical and recent developments in this field is offered in the summary "FPGA Dynamic and Partial Reconfiguration" [VF18] published in 2018.

2.3.1 Principle

First of all, the term *reconfiguration* refers to changing the configuration of an FPGA that is already configured. *Global reconfiguration* exchanges the whole design, while *partial reconfiguration* only exchanges the configuration in some part(s) of the FPGA. Typically, global reconfiguring leads to a complete loss of internal state and necessitates a shutdown/initialization cycle of the logic running on the device. This is not the case with partial reconfiguration, as the part of the FPGA that is not exchanged (referred to as the *static logic*) retains all state. Only the actually reconfigured part (referred to as the *dynamic logic*) needs to be reinitialized if necessary. When the reconfiguration is classified as *dynamic partial reconfiguration* (DPR), the static logic will even continue to run normally and can perform tasks unrelated to the units that are being exchanged. The opposing concept, i.e. stopping the whole FPGA for the duration of the configuration change, is called *static partial reconfiguration.* Alternative terms for these concepts are *active* (i.e. dynamic) and *passive* (i.e. static) reconfiguration. [Di$^+$12; Koc13; VF18]

Partial reconfiguration can be beneficial for a number of reasons: Digital signal processing and cryptography applications may include parameters that can be modeled as connections in digital circuits that do or do not exist, i.e. they are practically equivalent to configuration bits in LCs. Partial reconfiguration can program those bits at run time in order to parametrize the module. This will typically result in lower resource usage than traditional means to achieve the same purpose such as additional input signals to the algorithm. Furthermore, reconfiguration of FPGA-based systems on a module level allows sharing of resources that are never used at the same time.

This basic principle is illustrated in Figure 2.10a. An in-vehicle driver assistance unit will usually have to use image filters in order to process and act upon incoming picture data from various cameras. It might need an edge detection filter, a contrast enhancement filter, and an optical flow analysis filter, for example, as described in [Cla11]. Each is only usable in a specific driving situation, so they are never needed at the same time and having them occupy valuable FPGA resources permanently would be wasteful. An FPGA design with improved resource utilization using PR would be partitioned into static logic that is not reconfigured and dynamic logic in the form of one (or more) reconfigurable regions into which modules can be inserted at run time. When a module is to be exchanged, the surrounding system is tasked with sending a configuration bitstream to the configuration module of the FPGA. The component responsible for this task is typically called *partial reconfiguration controller* (PRC) (not to be confused with the previously introduced configuration controller that is a component of the FPGA hardware). If the PRC is operating from within the FPGA as part of the static logic, the process is classified as *self-reconfiguration*. From a high-level point of view, the reconfigurable region is like a stack of multiplexers for all signals going into and out of it (Figure 2.10b). Exchanging a module is equivalent to changing the selection of these multiplexers all at once.



(a) Principle of partial reconfiguration: A distinct region of the FPGA is designated as a reconfigurable region into which one of multiple implementations (in this example, of an image filter) can be inserted.

(b) Model of partial reconfiguration: The FPGA configuration module controls multiplexers on the reconfigurable region boundary that switch between multiple possible implementations all having the same interface. The whole operation is controlled by the surrounding (static) system.

Figure 2.10: Principle and model of partial reconfiguration. Adopted from [Koc13, fig 1.13].

In order to smoothly exchange these modules, all of them must have a compatible interface. On the HDL level, this typically means that all ports going in and out must be identical in name, type, and width. On the device level, in every module these ports must be routed to the same locations on the boundary of the dynamic logic area. Otherwise, the routes from and to the static design will not match up with the ones inside the dynamic logic and cause severe design malfunction akin to the likely crash of a software application when trying to load a software component that does not implement the expected interface.

Hardware and tools supporting DPR are at present only provided by Xilinx and Intel/Altera. A number of other FPGA manufacturers such as National Semiconductor, Lattice Semiconductor, and Atmel have previously offered this feature, but have since ceased to do so. Of the two remaining options, Xilinx FPGAs remain the most popular choice due to the company's long history of PR support initially offered already in 1995 in its XC6200 series. [VF18; Xil95]

2.3.2 Reconfiguration of Zynq-7000 devices

The FPGA inside the Zynq-7000 SoC is fully equipped for dynamic partial reconfiguration. It offers both external and internal interfaces connected to the PL configuration module that accept partial configuration bitstreams. In the context of this thesis no external components can be added, so only the internal ports are of interest. These are the PCAP for (re)configuring the FPGA from the PS and the ICAP for reconfiguring from within the FPGA design (see also Figure 2.7). They cannot both be used at the same time; which one of them is active is controlled by multiplexers accessible to the APU.

The primitive to instantiate for self-reconfiguration in HDL designs is called ICAPE2 for all Xilinx 7 series devices. Its input and output ports are documented in [Xil18i, p. 333] and are reproduced in Figure 2.11. For self-reconfiguration, the configuration data must be streamed to the 32-bit configuration input port `I` with the `CLK` port acting as clock while both the enable input `CSIB` and the read/write select input `RDWRB` are held low. Although the last part is not specifically documented, it can be inferred from the description of the equivalent `RDWR_B` configuration pin on the device itself [Xil18c, p. 28]. The output port `O` is optionally used when reading data back from the FPGA e.g. for verifying the configuration. There are no handshake signals since the operation of the ICAP is completely deterministic and predictable. One 4-byte unit of data from a configuration bitstream is accepted on the input port in any given clock cycle.

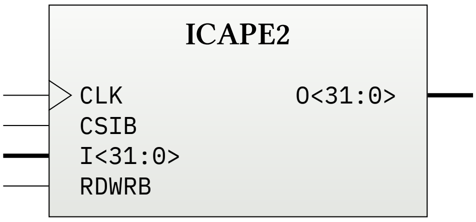


Figure 2.11: ICAPE2 primitive for self-reconfiguration of 7 series devices. Ports are inputs on the left-hand side and outputs on the right-hand side. Angle brackets indicate ranges of vector ports. Adopted from [Xil18i, p. 333].

The Xilinx synthesis and implementation tools and the Vivado IDE support partial reconfiguration in principle when a separate PR license has been purchased. There are, however, a number of limitations which are outlined further below. The workflow is described in detail in the Vivado partial reconfiguration user guide [Xil18m] and a hands-on tutorial [Xil18k] is also offered. The basic idea is that entities in the design hierarchy can be marked as *reconfigurable partitions* (RPs), which turns them into slots to which multiple different implementations (that are referred to as *reconfigurable modules* (RMs)) must be assigned.

The complete tool flow is illustrated in Figure 2.12: The IDE will first run synthesis for every RM declared. After that, it will implement a complete design including one initial (default) RM of choice for each reconfigurable partition and write a full bitstream using the traditional non-PR flow, with the primary difference being that for the RM instances, the gate-level netlists of the preceding synthesis runs are used. After that, for each non-default *configuration* (assignment of RMs to RPs specified by the designer), implementation is run reusing the static part of the initial design results. Finally, a partial bitstream per RM in every configuration (including the default one) is written.

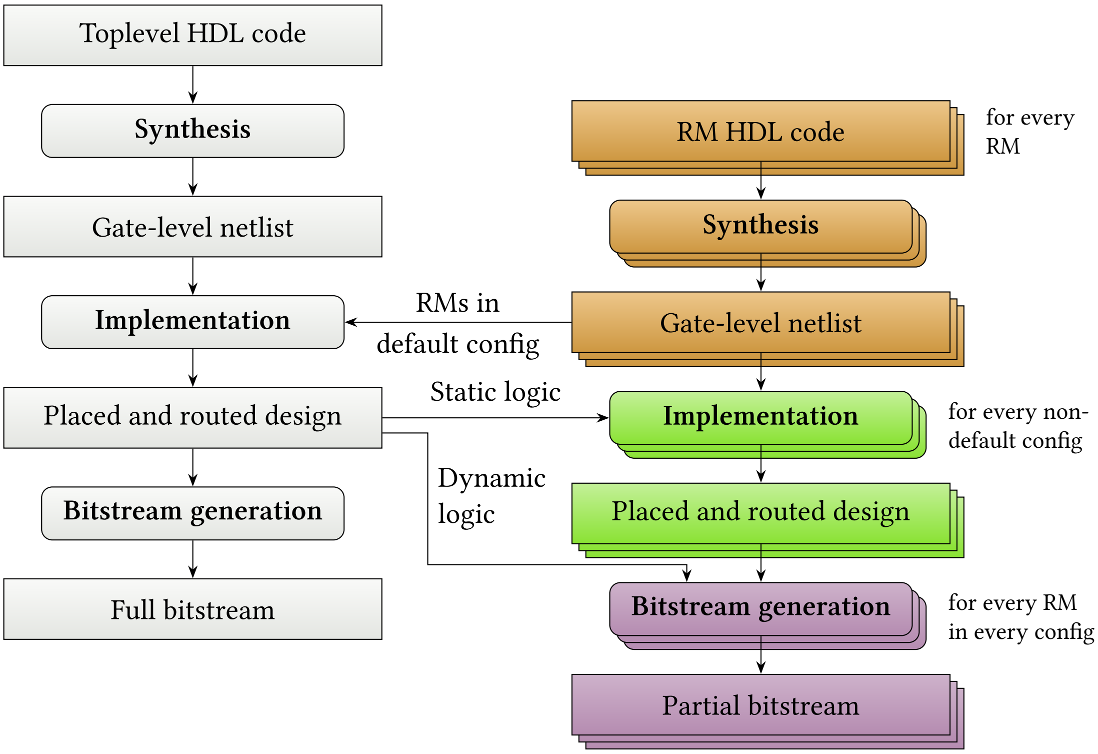


Figure 2.12: Xilinx design flow for partial reconfiguration showing the major steps for the full initial design (left-hand side) and all reconfigurable modules and configurations (right-hand side). The process for the full bitstream is identical to the non-PR case (cf. Figure 2.5). For PR, synthesis is performed for every reconfigurable module, while implementation is performed for every non-default configuration and partial bitstream generation for every RM in every configuration.

The designer manually has to select and specify an area of resources (called *Pblock*) on the FPGA per RP that forms the reconfigurable region on the device. Only resources (concretely: CLBs, DSP slices, and BRAM) contained in the Pblock assigned to a reconfigurable partition are available for implementation of each individual module. Consequentially, the size of the Pblock has to be determined such that for each resource it includes at least the maximum amount used in any of the modules assigned to an RP. In terms of pure logic usage, this will usually be the most complicated (i.e. largest) module. Short of a small number of hardware constraints that may not be violated, the Pblock location within the device is up to the discretion of developer. A possible approach to guide the Pblock layout is to implement the design statically once using the largest module per reconfigurable partition and create Pblocks approximately where the reconfigurable module logic ended up being placed by the tools. [Xil18m]

Each signal crossing a reconfigurable partition boundary from static to dynamic design or vice-versa is referred to as *partition pin* in Xilinx documentation. The locations of partition pins are determined automatically (although they can be specified manually) and are fixed after the initial design run. In earlier PR workflows, it was necessary to insert logic elements such as LUTs or flip-flops into the dynamic logic that would form the endpoint for routes from the static region. These elements were called *proxy logic*. The Vivado Design Suite has overcome this requirement and made these techniques obsolete, so the partition pins supposedly do not generate any logic overhead. [Koc13; Xil18m]

The Xilinx tools ensure adherence to a set of strict rules guaranteeing that the PR design works in practice. For the purposes of this thesis, the following restrictions are especially relevant: [Xil18m]

- Optimization algorithms may not cross the boundaries of reconfigurable partitions and their associated Pblocks.
- Logic elements of the static design may not be placed into the area of a reconfigurable partition to ensure that all resources in the partition are available for implementing the function of the modules and that the static logic is unaffected by switching between modules.
- Similarly to routes to partition pins, routes of the static design that run through a reconfigurable Pblock (called *feed-through routes* [RFG16]) must be fixed after the initial implementation and stay exactly the same in all modules. Otherwise, the static design would cease functioning as soon as an incompatible implementation has been loaded.
- Routes of reconfigurable modules must be contained entirely within their Pblocks.

These rules are followed by the Xilinx tools, but they must in principle be implemented by any synthesis software supporting partial reconfiguration.

2.3.3 Limitations and trade-offs

Partial reconfiguration as a technique is not very widely employed in commercial applications yet [Koc13; VF18]. This section will discuss some possible reasons and which important trade-offs have to be made when considering whether to use PR in a design.

Multiplexing FPGA resources in the time domain on the one hand allows significant area and power savings, reduces cost by potentially allowing a smaller device to be chosen, and increases design flexibility. On the other hand, there are a number of considerations to make:

- Reconfiguring the logic takes a certain amount of time. At the most basic level, the time required for exchanging a module equals its partial bitstream size times the average configuration throughput. This throughput is limited by the lower of two parameters: The maximum throughput allowed by the configuration controller of the device (as indicated in its data sheet) and the maximum rate at which the supporting system can supply configuration data to this controller. Ideally, the bitstreams can be read and buffered at speeds exceeding the limit of the configuration controller to make sure that the system achieves the full potential of partial reconfiguration on the chosen FPGA.

  As far as the device throughput limit is concerned, Figure 2.13 shows the evolution of DPR throughput in commercial FPGAs of the biggest suppliers Xilinx and Intel/Altera during the past 20 years up to the newest devices at the time of the writing of this thesis (Virtex UltraScale+, Intel Stratix-10). It can be seen that speeds of the Virtex family did not improve at all for 10 years between 2004 and 2014.

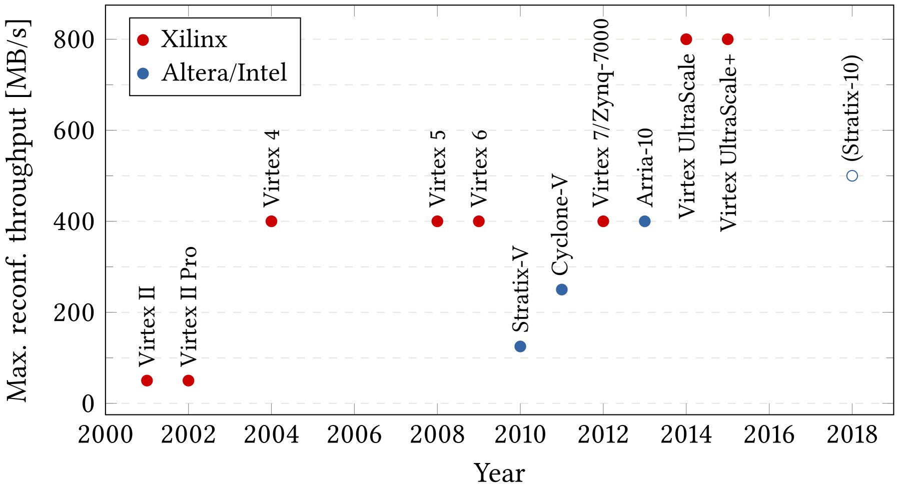


Figure 2.13: Evolution of reconfiguration throughput of Xilinx and Altera/Intel FPGA families supporting dynamic partial reconfiguration. Adopted from [Bio+16, fig. 1]. For Xilinx, speeds are comparable for different families of the same generation. Intel figures are from [Alt18b; Alt18a; Int18b; Int18c; Int18a]. Stratix-10 devices are not generally available yet [VF18], so their reconfiguration throughput must be considered preliminary.

At the same time, FPGAs grew larger in size and consequently needed a greater amount of configuration data. To illustrate: The FPGA part of the Z-7020 used in this thesis has a total bitstream size of 4.05 MB, while it takes up 55.92 MB for the biggest Virtex-7 device. Reconfiguring 10 % of the devices at the maximum throughput of 400 MB/s would take $4.05\text{ MB} : 10 : 400\text{ MB/s} = 1.0\text{ ms}$ and $55.92\text{ MB} : 10 : 400\text{ MB/s} = 14.0\text{ ms}$, respectively. To put this into a larger context, a system that processes a video stream in real time at 50 frames per second must finish each frame in at most 20 ms. Consequentially, a reconfiguration of 10 % of the device resources would be permissible only one time per frame for the big Virtex-7 FPGA. [Xil18q; Xil17f; Xil18c]

The latest UltraScale and UltraScale+ generation devices including the Zynq UltraScale SoCs have achieved significant progress by doubling the previous maximum throughput to 800 MB/s. The largest Virtex UltraScale device uses full bitstreams sized 128.97 MB, so a 10 % reconfiguration in this case would take $128.97\text{ MB} : 10 : 800\text{ MB/s} = 16.1\text{ ms}$. Despite the vastly improved throughput, it still takes longer than for the previous generation due to the even larger increase in configuration data size. It is currently unclear whether reconfiguration throughput will be further improved for upcoming generations or if it will stay the same for a longer period of time again. [Xil18g; Xil18h; Xil18o]

The size of any partial bitstream is approximately proportional to the area the reconfigurable partitions contained in it occupy on the chip. Partitions of increased size and number will naturally lead to longer reconfiguration times. Finally, how often reconfiguration needs to be performed also influences the overall performance. Depending on the concrete usage scenario, a moderate throughput supported by a simple system design might be completely sufficient and not require the developer to reach the maximum speed of the device or buy a newer, more expensive FPGA. [Xil18m]

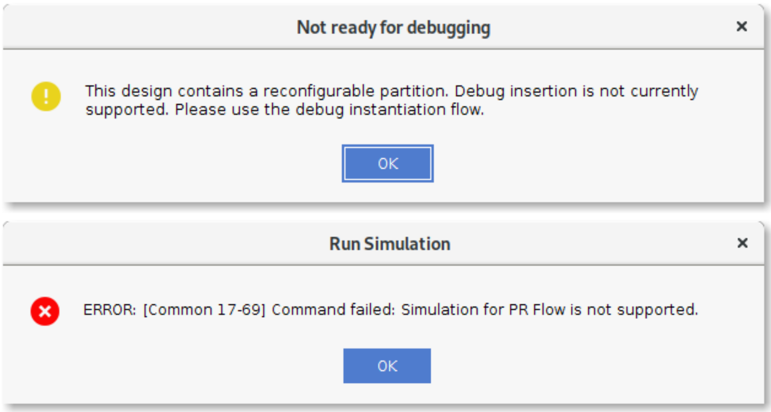
- Reconfiguring the logic requires energy. This aspect will not be considered further in this thesis. For observations on the energy cost of partial reconfiguration, see e.g. [NL16] which claims that the power consumption overhead is very small and [Bec^{+}16] which presents measurement results for a Zynq-7000 SoC.

- Reconfiguration requires supporting components to perform it that themselves take up resources in the system. In the case of self-reconfiguration, resources on the FPGA will be additionally taken up by the partial reconfiguration controller delivering the bitstreams, so it has to be very efficient in order to not spoil the area savings obtained by using PR in the first place. Practical PRCs make sure to have reasonable resource requirements.

- With configuration throughput rising, it gets increasingly challenging to provide the data to the FPGA configuration controller at sufficient speeds; in fact this becomes the key problem to solve for achieving the fastest possible reconfiguration. Simple flash memories and rotational media cannot operate fast enough, while FPGA-internal static memory (BRAM) is quite limited in size. Viable choices are high-speed low-latency external memories such as SRAM, *synchronous dynamic random-access memory* (SDRAM), and *solid-state drives* (SSDs). One of these has to be added to the system. Even if a system already included any of these components as main memory or for data storage, a significant part of the memory bandwidth available might be required for supplying configuration data. This needs to be taken into account when calculating bandwidth allocations and margins. When reconfiguration is not performed very frequently though, the impact on the system should be limited.

- The rules implemented by the design tools that make PR practically feasible cause some amount of degradation in design performance characteristics. For example: Because modules in reconfigurable regions are isolated from both the static design and each other, their placement and routing cannot be optimized globally. Consequently, they will take up more FPGA resources (mainly LCs and interconnect) than they would in an equivalent static design. How big this difference is depends on the concrete design, but for reasonably sized RPs it is extremely unlikely to exceed the resources saved by making the component reconfigurable in the first place. [Xil18m]
- Although suppliers design for PR on a device level and put corresponding options into their tools, they still remain limited. For instance, Xilinx Vivado has a number of limitations when defining reconfigurable partitions in the design: They must not pass parameter and generic values from above, must not contain block diagram sources, must contain only unique IP core instances, must be assigned to exactly one Pblock (that in turn must not contain other RPs), and the same module must not have a static and a dynamic instance [Xil18m]. Additionally, once PR is enabled in a design, important features such as marking signals for debugging in the integrated debug core by HDL attributes and simulation stops working for the whole design, even the static parts (see Figure 2.14). These constraints limit design flexibility and complicate development. For simulation, effectively a second project (with the PR flow disabled) has to be created using the same sources and kept in sync.

This list of drawbacks is not intended to deter the reader from the usage of partial reconfiguration in any way. PR is a very powerful technique that enables designs that would not be possible without it. Realistically, the advantages have to be traded off against the significant increase in system complexity. For most designs that want to reconfigure specific unique modules on an infrequent basis and with a sizable time margin, a lot of the considerations mentioned above will not weigh in very heavily. Nonetheless, how partial bitstreams are stored and delivered are questions that have to be answered for every design. More complicated use



This design contains a reconfigurable partition. Debug insertion is not currently supported. Please use the debug instantiation flow.

ERROR: [Common 17-69] Command failed: Simulation for PR Flow is not supported.

Figure 2.14: Error messages encountered when trying to use standard IDE features of the Vivado design suite in partial reconfiguration-enabled projects.

cases that require very fast reconfiguration like the one presented in this thesis additionally have to make sure that the performance goals can be reached in practice.

2.4 The ViSARD

The ViSARD is a soft-core processor developed at the Computer Architecture and Embedded Systems Group of Technische Universität Ilmenau. It is conceived as a processing element for hard real-time systems requiring floating-point calculations with high precision. One exemplary use case is the standalone white-light interferometry optical measurement device presented in [MMF14] that “allows to map surfaces with nanometer resolution”. The proposed system uses the ViSARD to perform post-processing of the captured image data with minimum latency.

In summary, the main features of the ViSARD are: [Kir14; KF14]

- VHDL implementation configurable with generics:
  - Selectable single (32 bits) or double (64 bits) precision floating-point registers and operations
  - Selectable set of *execution units* (EUs)
  - Configurable instruction encoding (e.g. opcode width)
- Hard real-time capability
- 5-stage pipeline
- Arithmetic calculations operate on floating-point values and include support for special operations relevant to scientific calculations such as the natural exponential function

- Harvard architecture, i.e. program and data memories are accessed separately [HPA07]
- *Reduced instruction set computer* (RISC) load/store architecture, i.e. the operations are rather generic and accessing memory and manipulating data is separate [HPA07]
- 3-operand instruction set
- Power saving by disabling inactive execution and memory units
- Simple data input/output interface

The following subsections will discuss the general structure of the architecture as well as the accompanying tooling, especially the assembly language and assembler. Finally, the *arithmetic logic unit* (ALU) of the ViSARD and its operations will be explained.

2.4.1 Structure

Figure 2.15 shows the structure of the processor. Beginning in the upper left part of the picture, the *program counter* keeps track of which command is being fetched and executed at the moment. It starts at the `Prg_Add` address given to the ViSARD as input when the `MCmd` port is high. When the program is finished, the `StatusBusy` output port is asserted for one clock cycle in order to signal this event to the surrounding system. The *program memory* is directly connected to the program counter and contains the instructions to perform.

Each command read from the program memory is fed to the *instruction decoder.* It extracts the two input operand addresses, the opcode to execute, and the write-back address which is effectively the address of the output operand. All addresses are given to the two *data memories* A and B which form the register file of the processor. They are each implemented as *dual-port random-access memory* (DPRAM) that can be simultaneously read from and written to in the same clock cycle. Their contents are identical so that two arbitrary register values can be read simultaneously in every clock cycle, which is necessary for operations that require two input operands such as addition. Output values are always written to both memories on completion of an operation, making sure that their contents are identical.

All floating-point EUs that perform the actual calculations are contained in the ALU. Available as data to the ALU are the input operands A and B read from memory and, additionally, data from the external `SData` bus used to hand input values to ViSARD programs from outside. Inside the ALU, a multiplexer selects the output of the EU finishing in the current clock cycle as result to write back to the data memories. It is also possible to write results to the external `MData` bus for consumption by the surrounding system. In order to support multiple output values, the `MData_Adr` port uses the write-back address field of the current instruction to indicate which value is being provided.

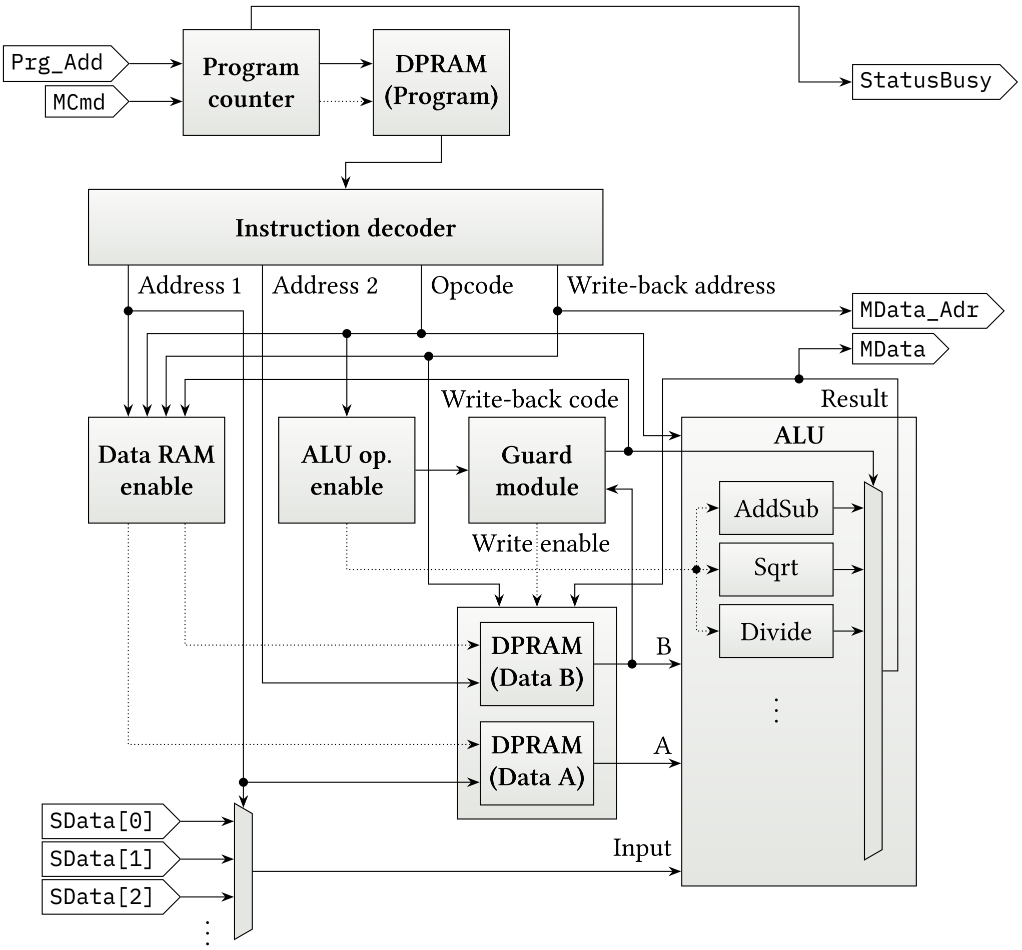


Figure 2.15: Schematic overview of the ViSARD structure. Dotted lines indicate clock enable signals. DPRAM: Dual-port random-access memory. Adopted from [Kir14, fig. 15] (updated to match the current code).

As the ViSARD is fully pipelined and many calculations are potentially running in parallel, the *ALU operator enable* component has to keep track of all of them and identify which EU is finishing its calculation in a given clock cycle. This information is provided to the *guard module*, which forwards it to the ALU and disables writing the current result to the data memories if the input operand B matches certain user-configurable conditions such as equality to zero. Using the guard module, it is possible to perform conditional assignments in programs even though the ViSARD can execute programs only linearly.

For reduced power consumption, the *data RAM enable* and ALU operator enable components control the clock enable inputs of the data memories and execution units, respectively, and assert them if the corresponding component is required to be active.

The hard real-time capability of the ViSARD allows it to be used in scenarios where an algorithm has to be run with strict time constraints and failure to do so would be critical to the operation of the system. It is achieved by a simple instruction set that does not contain any conditional branching instructions and execution units that finish in a fixed number of cycles for every possible input. Additionally, there are no interrupts, no predictive elements, and no caches. In combination, this guarantees that the number of cycles any given program needs for execution is always identical. The instruction set in fact does not contain branching instructions at all, further simplifying the run-time analysis.

2.4.2 Toolchain

Processor architectures themselves have limited practical usefulness due to their very low-level nature. They are generally accompanied by a toolset enabling application development at an appropriate level of abstraction [Kir+17]. The complete toolchain of the ViSARD is shown in Figure 2.16. Part of the project is an assembly language specification to describe programs for the processor and an assembler that converts this language into bytecode [Wag17] as well as a model-based code generator that converts a Matlab/Simulink model into assembly code. More information on model-based development with the ViSARD can be found in [Kir+18].

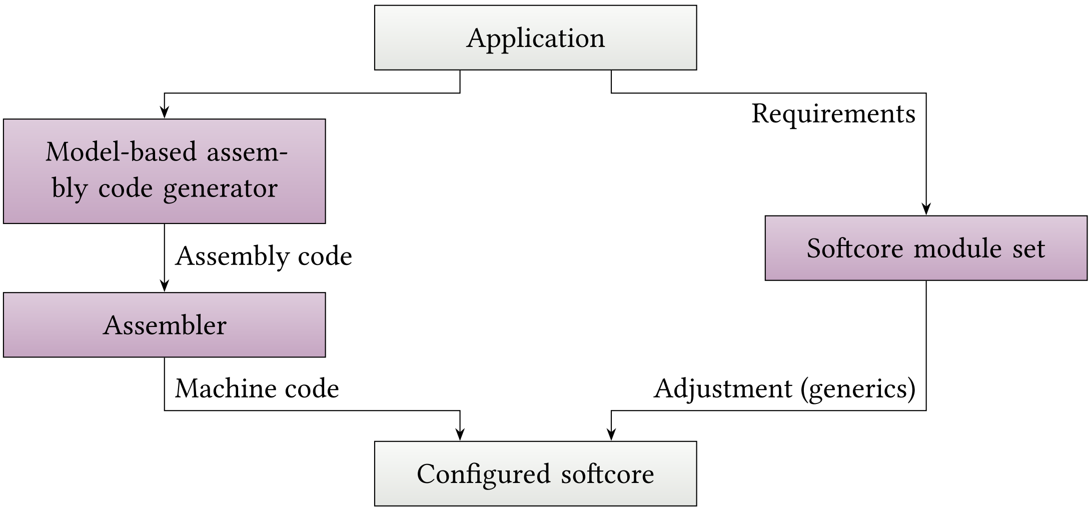


Figure 2.16: The ViSARD toolchain. Parts in purple are part of the project. The configured soft core is the final output that can be put on an FPGA after synthesis. Adopted from [Kir+17, fig. 1].

2.4.3 Assembly language and assembler

Assembly code files for the ViSARD are a strictly sequential list of unscheduled commands, i.e. the code may assume that any operation is complete when the next operation starts. The actual instruction and register scheduling is performed by the assembler according to a configurable optimization algorithm, taking into account the execution time specific to each operation. This frees the programmer from having to consider the details of the microarchitecture when writing code, at least to a certain extent.

Commands are always denoted with the (case-insensitive) instruction mnemonic, two input operands, and one output operand, in this order. If the specific operation does not use an operand, a single question mark "?" is used as placeholder. Apart from commands, the source file may only contain empty lines and comments identified by a leading semicolon. The special `dq` command introduces variables that can be used by later instructions as operands. The arguments of the `dq` command itself are the name of the variable, the placeholder "?", and finally the initial value. Consider the following example program:

```
dq  A ? 0 ; Declare variable A for storing the input value (initially 0)
dq  B ? 5 ; Declare variable B with value 5 (used as constant)
dq  C ? 0 ; Declare variable C for storing the output value (initially 0)

in  0 ? A ; Read input value 0 to variable A
mul A B C ; Multiply input variable A with variable B, save in C
out C ? 0 ; Write C to output value 0
```

It will read in (instruction `in`) one value, multiply it with the constant `B` (5, instruction `mul`), and finally write the result as output (instruction `out`). This example also demonstrates the usage of the `in` and `out` instructions that are provided for communicating non-constant data that the program should use for calculations with the design embedding the processor. The input variables are read from the `SData` port of the processor, while the output variables are written to the `MData` port of the processor (as shown in Figure 2.15). The operands of the instructions denote the variable to be read or written and an index into the corresponding port for addressing multiple in/out variables.

For correct operation, it is important that the assembler and the actual processor agree on parameters such as the instruction encoding and the execution delays of all operations. Due to the configurability and adaptability of the processor, this information typically varies depending on the concrete usage scenario of the processor. The assembler therefore requires a separate configuration file in *Extensible Markup Language* (XML) format that must be kept in-sync with the generic values specified on the ViSARD instance. The file additionally includes assembler-specific information such as the available optimization algorithms.

The output of the assembler is the binary content of the program and data memories in separate text files in hexadecimal notation. These files can be set on the ViSARD instance in the VHDL code in order to fill the memories on the FPGA.

The binary instructions for the processor are structured as seen in Figure 2.17. The operation to perform is specified by the 5-bit opcode field. Additional flags that affect the output of the operation can be given in the 3-bit ALU flags field. This is e.g. used to optionally invert the sign of input operands prior to calculation. The addresses of the input operands and the output write-back operand each have one dedicated field that is currently 8 bits wide. Operand flags

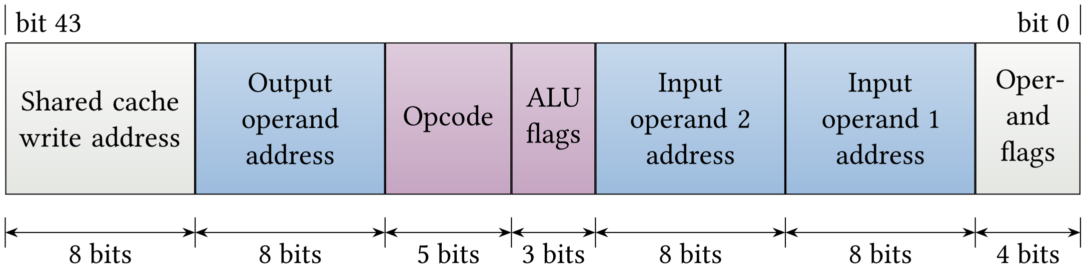


Figure 2.17: ViSARD instruction encoding. Marked in blue: operand addresses. Marked in purple: command (opcode and flags). Adopted from [Wag17, fig. 16].

include the option to bypass operand fetching for one or both input operands. The shared cache field is used for the multi-core extension of the ViSARD described in [Wag17] and will not be discussed further.

An important point to note is that the output write-back address field does not refer to the output of the operation specified in the same binary instruction, but rather to the output of the operation completing in the same clock cycle. This implies that at most one operation can complete per clock cycle. Correctly filling this field is the responsibility of the assembler which delays the output operand given in the original assembly instruction by the number of clock cycles the operation takes to complete. Additionally, the assembler will delay a command if it would complete at the same time as an already issued command. This way, all constraints of the architecture are taken into account. [Wag17]

2.4.4 ALU/FPU operations

The ALU of the processor architecture is the core component performing all computations and exclusively operates on floating-point numbers. Consequentially, it can also be classified as *floating point unit* (FPU). This interpretation is furthermore supported by the fact that there are no logical operations such as bitwise Boolean AND that would be characteristic of an ALU. In this thesis, the terms ALU and FPU are used interchangeably in the context of the ViSARD.

Table 2.1 lists the assembly mnemonics and descriptions for all operations defined for the processor as well as their currently assigned opcodes. Some conversion operators are only available in single or double precision configuration of the processor. The list was initially compiled from the information found in the theses of Ölschlegel [Öls15, table 1] (for an up-to-date list of mnemonics; referred to as "Table A" below) and Kirchhoff [Kir14, table 2] (for opcodes; referred to as "Table B" below), but some deviations were found in the current code that necessitated a revision:

- The input operands in Table A (named Op1, Op2) are mixed up for all conditional assignment operators. They are the other way around. This is correct in Table B.

Table 2.1: ViSARD operations and associated mnemonics. in_1 and in_2 denote the two input operands while *out* denotes the output operand. The special operation sign returns the sign bit of a floating-point number. Furthermore, fsig returns the significand of a floating-point number, while fexp returns the exponent. When flags are not specified for a mnemonic, their value does not affect the operation. Data types are: int8 for 8-bit integers, int16 for 16-bit integers, int32 for 32-bit integers, and single/double for single and double precision floating-point numbers, respectively. `SData` and `MData` are special buses for communication with the embedding design.

Mnemonic	Prec.	Opcode	Flags	Effect
`in`	both	`00000`		$out \leftarrow \mathtt{SData}[in_1]$
`out`	both	`00001`		$\mathtt{MData}[out] \leftarrow in_1$
`mov`	both	`00010`	`000`	$out \leftarrow in_1$
`abs`	both	`00010`	`001`	$out \leftarrow \lvert in_1 \rvert$
`add`	both	`00011`	`000`	$out \leftarrow in_1 + in_2$
`sub`	both	`00011`	`100`	$out \leftarrow in_1 - in_2$
`abssub`	both	`00011`	`111`	$out \leftarrow \lvert in_1 \rvert - \lvert in_2 \rvert$
`mul`	both	`00101`		$out \leftarrow in_1 \cdot in_2$
`div`	both	`00110`		$out \leftarrow in_1 : in_2$
`wbnone`	both	`00111`		No operation
`sqrt`	both	`01000`		$out \leftarrow \sqrt{in_1}$
`sin`	both	`01001`		$out \leftarrow \sin in_1$ (input in radians)
`cos`	both	`01010`		$out \leftarrow \cos in_1$ (input in radians)
`sgl_to_i16`	single	`01011`		$out \leftarrow$ convert in_1: single to int16
`sgl_to_dbl`	double	`01011`		$out \leftarrow$ convert in_1: single to double
`sgl_to_i32`	single	`01100`		$out \leftarrow$ convert in_1: single to int32
`dbl_to_i32`	double	`01100`		$out \leftarrow$ convert in_1: double to int32
`i32_to_sgl`	single	`01101`		$out \leftarrow$ convert in_1: int32 to single
`i32_to_dbl`	double	`01101`		$out \leftarrow$ convert in_1: int32 to double
`i16_to_sgl`	single	`01111`		$out \leftarrow$ convert in_1: int16 to single
`dbl_to_sgl`	double	`01111`		$out \leftarrow$ convert in_1: double to single
`i16_to_dbl`	double	`11000`		$out \leftarrow$ convert in_1: int16 to double
`dbl_to_i16`	double	`11001`		$out \leftarrow$ convert in_1: double to int16
`i8_to_dbl`	double	`11011`		$out \leftarrow$ convert in_1: int8 to double
`mov_ispositive`	both	`10000`		$out \leftarrow in_1$ **if** $\mathrm{sign}(in_2) = 0$
`mov_isnegative`	both	`10001`		$out \leftarrow in_1$ **if** $\mathrm{sign}(in_2) = 1$
`mov_isnull`	both	`10010`		$out \leftarrow in_1$ **if** $\mathrm{fsig}(in_2) = 0$ **and** $\mathrm{fexp}(in_2) = 0$
`mov_isnotnull`	both	`10011`		$out \leftarrow in_1$ **if** $\mathrm{fsig}(in_2) \neq 0$ **or** $\mathrm{fexp}(in_2) \neq 0$
`mov_isnan`	both	`10100`		$out \leftarrow in_1$ **if** $\mathrm{fsig}(in_2) \neq 0$ **and** $\mathrm{fexp}(in_2)$ is all `1`s
`mov_isnotnan`	both	`10101`		$out \leftarrow in_1$ **if** $\mathrm{fsig}(in_2) = 0$ **or** $\mathrm{fexp}(in_2)$ is not all `1`s
`mov_isinf`	both	`10110`		$out \leftarrow in_1$ **if** $\mathrm{fsig}(in_2) = 0$ **and** $\mathrm{fexp}(in_2)$ is all `1`s
`mov_isnotinf`	both	`10111`		$out \leftarrow in_1$ **if** $\mathrm{fsig}(in_2) \neq 0$ **or** $\mathrm{fexp}(in_2)$ is not all `1`s
`natexp`	both	`11010`		$out \leftarrow e^{in_1}$

- Conditional assignments were represented inaccurately in both tables. On the one hand, Table A oversimplifies by abstracting away the floating-point representation in hardware. For example, the `mov_ispositive` operation is specified as "if Op1 >= 0.0 [Target] := Op2" in Table A. However, the processor only checks the sign bit of the floating-point number. This difference becomes relevant when handling special values such as *not a number* (NaN). By definition, comparing NaN with any value for (in)equality should always yield a false result [IEE08], meaning that `mov_ispositive` would not perform the assignment when the input operand is NaN. In reality, the processor will perform the assignment if it is a positive NaN (i.e. the sign bit is not set).

  On the other hand, Table B correctly uses special operators to represent the sign, significand, and exponent parts of floating-point numbers without further simplification, but has wrong formulas for checking for unequal-to-zero (logical operator must be "or", not "and", and the second condition must be inverted) and NaN (exponent must be all 1s, not all 0s). These have been corrected here.

- The instructions `movconst` and `mlookup` appear in Table B, but not in Table A, and do not currently have an implementation since they were added only for compatibility with an earlier project. Consequently, they were omitted here.

- The instruction `u8_to_dbl` newly introduced in Table A was renamed to `i8_to_dbl` since the execution unit can only accept signed integers, contrary to what the mnemonic would indicate.

- ALU flags and opcodes on the `abs`/`mov`/`add`/`sub`/`abssub` family of instructions as indicated in Table B did not match the implementation in all cases and were not consistent. This was resolved by giving each of the ALU flags a distinct meaning and matching the opcodes to the EUs. Flag bits 0 and 1 control whether to unset the sign bit of the first and second input operand, respectively. Bit 2 decides whether the resulting sign bit of the second input operand should in turn be inverted. One opcode each was dedicated to passing the input value on without further operation (`00010`) and floating-point addition (`00011`). This way, all mentioned mnemonics can be translated to a suitable combination of opcode and flags, as seen in the table.

- The `abssub` instruction is defined in both table A and table B in the same way in which it is reprinted here, but the actual implementation did not match up. Instead of $|in_1| - |in_2|$, it calculated $|in_1| - in_2$. This has been corrected in the code as part of the ALU flag rework.

Nearly every mathematical operation has exactly one dedicated EU that exclusively handles that operation. The exceptions are addition/subtraction and sine/cosine, which each share one EU at the moment due to the similarity of the calculations involved. All units are fully pipelined, i.e. a new operation can be started in every cycle. The processor implementation makes sure that results are written to the output operand location when they are ready, while the assembler makes sure that data dependencies between commands are met.

This chapter has introduced FPGAs, important aspects of digital systems design, partial reconfiguration, and the ViSARD processor. These are the technologies that the research presented in this thesis is built upon. In the following chapters describing the design and implementation of the PR extension of the ViSARD, the information presented here will often be referred to and support the understanding of the topic.

Chapter 3

Design

As mentioned in the introduction, the primary goal of this thesis is to develop a system allowing the ViSARD to benefit from partial reconfiguration of the underlying FPGA fabric. This system will need at least the following components (Figure 3.1):

- One or more ViSARD cores with modified ALUs that are prepared for swapping out one or more execution units with partial reconfiguration,
- a PRC for delivering partial bitstreams to the FPGA when needed,
- a memory that holds the partial bitstreams for access by the PRC, and
- an application-specific top-level control module observing the ViSARD cores and initiating program execution and partial reconfiguration as appropriate.

Some variations concerning the exact borderlines and connections of the components are possible and will be discussed further down in this chapter. In general, however, these are the building blocks required to enable dynamic partial reconfiguration of the ViSARD and consequently form the set of subtasks that have to be achieved. Below, the first component to be described is the PRC and where partial bitstreams are stored, followed by the modifications to the ViSARD.

3.1 Partial reconfiguration controller

As a first step, the requirements for the partial reconfiguration controller need to be devised. In order to be practically useful, these are (in no particular order):

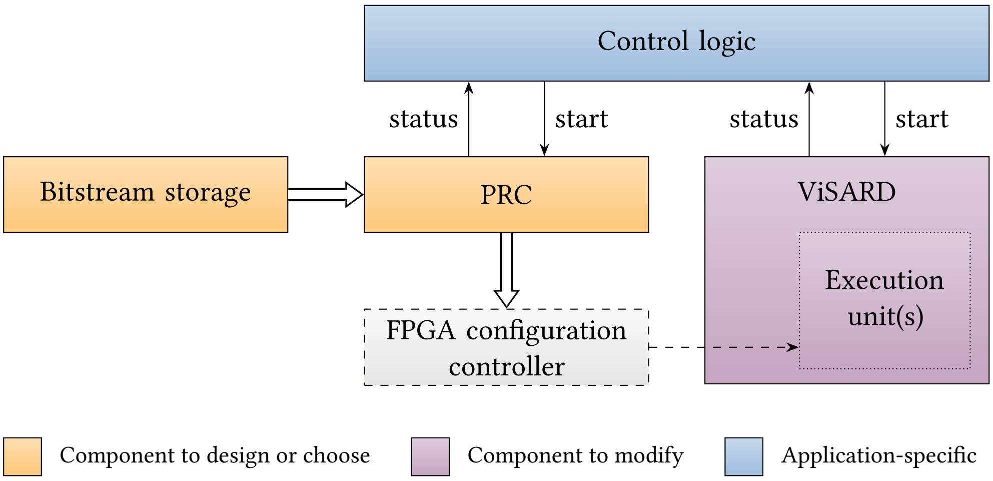


Figure 3.1: Essential components of a partially reconfigurable ViSARD-based system. Arrows indicate conceptual flow of information and do not necessarily match the actual port directions. The configuration controller (dashed) is a fixed component of the FPGA that will swap out the execution unit(s) inside the ViSARD based on the bitstream given to it by the PRC. There may be multiple ViSARD cores.

1. It must run on the hardware available for research, specifically the Trenz Electronic GigaZee SoM introduced in Section 2.2.2. This implies that it must be designed to support at least the Zynq-7000 FPGA and take the limitations of the GigaZee board design into account. Furthermore, it should also run on FPGAs that do not include a hard CPU core but are otherwise similar such as the Xilinx Virtex-7 series or be easy to adapt.

2. It must be implemented in an uncomplicated fashion such that future research can readily perform modifications as seen fit.

3. Its timing must be predictable so that the hard real-time capability of the ViSARD is not compromised in the complete system.

4. Its FPGA resource requirement should be low, since the PRC constitutes overhead introduced by partial reconfiguration. Having it take up a lot of space would severely affect its usefulness [Guo^{+}17].

5. It should be fast so that the potential of partial reconfiguration can be evaluated. More concretely, the bitstream delivery should be the limiting factor for the reconfiguration throughput since its maximum speed is a device characteristic (400 MB/s in case of the Zynq-7000 SoC) and cannot be influenced. The configuration access port of the FPGA should be active for at least 95 % of the time between the request and the completion of a PR cycle.

With these requirements in mind, several possible solutions were evaluated.

3.1.1 Available options

PRCs using the PCAP Firstly, requirement 1 effectively precludes a whole class of reconfiguration controllers: All PRCs that use the PCAP for bitstream delivery cannot be used due to its exclusivity to the Zynq series. Adapting such a controller to usage of the ICAP would warrant a complete re-design. Furthermore, Xilinx specifies a typical throughput of the PCAP of around 145 MB/s on the Zynq-7000 [Xil18s, p. 221] compared to a maximum of 400 MB/s for the ICAP. Experiments suggest that the realistically achievable PCAP data rates are a bit lower in practice at around 128 MB/s [VF14; RA14]. This is not enough to satisfy requirement 5. Therefore, only solutions involving the ICAP were evaluated further.

PRCs with CPU-tailored interfaces Secondly, another class of reconfiguration controllers is designed specifically for usage in combination with a CPU that provides commands and/or bitstreams. For this purpose, soft-core processors such as the MicroBlaze [Xil17d] included in the Vivado design suite can be used in combination with a peripheral on the AXI bus that enables access to the ICAP. The most basic design will use the AXI *Hardware Internal Configuration Access Port* (HWICAP) IP core by Xilinx [Xil16b] (or similar) that is essentially a low-level bridge between the AXI bus and the ICAP. All control logic is implemented in software (cf. Figure 3.2).

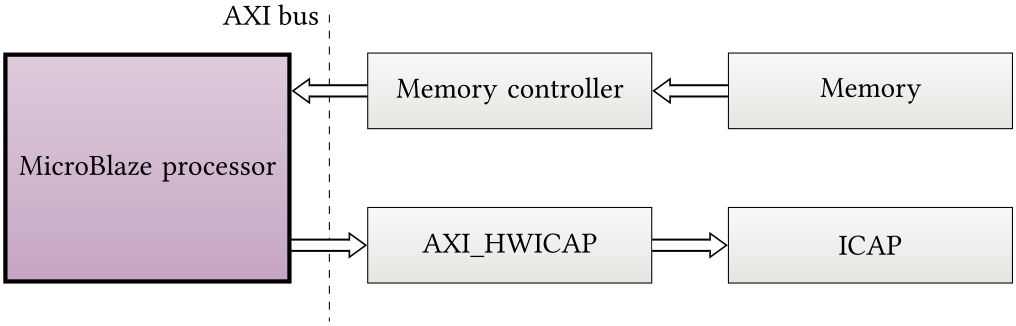


Figure 3.2: Exemplary structure of a microprocessor-based partial reconfiguration system using the HWICAP solution by Xilinx. The MicroBlaze processor is in complete control of the reconfiguration process and responsible for fetching bitstream data from the memory and forwarding it to the ICAP. Arrows indicate conceptual flow of information and do not necessarily match the actual port directions.

More advanced approaches like the RT-ICAP [PSS17] move some aspects of the process out of the CPU and provide additional features such as bitstream decompression, but still offer an interface that is best suited for interoperating with a processor. There are a number of similar designs along these lines; see for example [Hüb+10; Guo+17; CF15]. Furthermore, it is possible to use the hard CPU core inside the Zynq for controlling the reconfiguration process, but send the bitstream to the ICAP instead of the PCAP for performance reasons. An example of this is the ZyCAP [VF14]. There is even a proposal by Becher et al. that “includes a hybrid reconfiguration approach utilizing both the [PCAP] and the [ICAP] in order to minimize reconfiguration time and system energy consumption" [Bec+16]. Systems using the ViSARD, however, might or might not include other CPUs. Requiring the inclusion of one only to facilitate DPR would run counter to requirement 4 due to the increased resource usage.

Xilinx PRC Thirdly, Xilinx itself offers a partial reconfiguration controller as part of their IP core library that can be used without an additional fee [Xil18f]. It supports up to 32 reconfigurable regions that can each contain up to 128 different configurations. The basic operation is simple: When given the corresponding trigger signal to load a specific configuration into a reconfigurable region, it fetches the bitstream from a user-configurable memory location on the AXI bus and pipes it to the ICAP. It offers several more advanced features such as an AXI control interface and module shutdown/startup management that are not required for the ViSARD and therefore not considered here.

The number of reconfigurable regions and the memory locations and sizes of the partial bitstreams have to be specified in the parameters of the PRC IP core at design time, but they can be modified after the design has been implemented by executing special commands in the Vivado design environment. Since the size of partial bitstreams is typically determined after implementation, without this feature the whole process would have to be rerun from the top after changing a value in the configuration of the IP core.

As far as the requirements for this thesis are concerned, the Xilinx PRC directly violates constraint 3 because its timing and speed characteristics are not specified at all. Xilinx does neither guarantee that the IP core will start and/or finish reconfiguration within a defined amount of clock cycles nor make any statement about the typically achievable reconfiguration throughput. Furthermore, the controller is essentially a black-box, contradicting requirement 2.

PRCs in the literature Fourthly, there are other existing PRCs proposed in the literature that are designed for stand-alone usage in hardware designs without any supporting processor. This includes the older ICAP-I by Lai and Diessel [LD09] as well as the newer *dynamic partial reconfiguration manager* (DPRM) by Tarrillo et al. [Tar+14] and a DPR scheme by Beckhoff, Koch, and Torresen [BKT14] additionally offering relocation-aware compression. Their design files are not publicly available, so they cannot be used in the system implemented in this thesis. Additionally, none of those takes real-time constraints into account, violating requirement 3, and they do not come close to the device limits in terms of configuration speed. The scheme proposed by Beckhoff et al. for example is reported to reach a configuration throughput of 73 to 97 MB/s, while Lai and Diessel achieve 180 MB/s and Tarrillo et al. get up to 253 MB/s in measurements while theoretically reaching the device maximum of 400 MB/s when storing the bitstream completely in BRAM. The same is true for the Speed Efficient Dynamic Partial Reconfiguration Controller (SEDPRC) presented in [Bha+12] that is designed to exclusively use BRAM.

Custom PRC Finally, it is possible to implement a custom reconfiguration controller that is tailored to the requirements of the work in this thesis as outlined above. This is the option that was chosen considering the significant disadvantages of the alternatives. The requirements now become the design goals of the custom PRC. As there are no goals that conflict with each other, it should be possible to fulfill all of them. The resulting VHDL entity will be called simply `prc`.

3.1.2 Custom controller

The basic function of the PRC is to load a partial bitstream from memory and stream it to the ICAP when instructed to do so by a trigger signal that includes the information of which specific one of a number of bitstreams to activate. To achieve satisfying performance when considering the whole system, the primary questions are where (i.e. in what kind of memory) the bitstreams are stored, how that location is accessed, and how the data is forwarded to the FPGA. These questions will be answered below.

Bitstream storage and access

Many options can be considered for the storage location of the bitstream. The first categorization is whether the memory is internal or external to the FPGA. Internal memory is available as BRAMs distributed in the device. They can be used as *read-only memorys* (ROMs) by not writing to them from the hardware design, but instead putting their contents into the full bitstream initially loaded onto the device. The block RAM is the fastest option since it is usually located directly adjacent to the logic cells requesting its contents and is implemented as SRAM, i.e. it has no access latency. It does, however, come with considerable downsides.

Block RAM is a scarce resource in most FPGAs compared to the abundance of logic cells. The device used for implementation in this thesis, the Z-7020, is based on the low-cost Xilinx Artix-7 series and contains 0.6125 MB of BRAM [Xil18r]. Considering that a full bitstream for this device is about 4.05 MB, reconfiguring 10 % with two interchangeable implementations will need about $4.05\text{ MB} : 10 \cdot 2 = 0.81\text{ MB}$. This already exceeds the amount of available memory. Furthermore, even when reconfiguring portions smaller than 10 %, the bitstream storage occupies valuable resources on the FPGA that might be required for the actual application. This severely impedes goal 4. It is “not acceptable to expect so much resources to be free in the device” [Siv10]. Compressing the bitstream would allow for a reduction in data size of about 50 %

according to [Koc13], but it comes at the cost of increased logic resource usage and possibly reconfiguration time[1], running counter to goal 5.

Another issue is that the partial bitstreams have to be embedded in the full initial bitstream as BRAM initialization data in order to be accessible in the design. However, the contents are known only after the static design has been compiled once and all partial bitstreams have been written. So the design must be compiled again, this time using the results of the previous run in the BRAM initialization memory files. This incurs a considerable amount of overhead both in design implementation time and in the complexity of the whole process. An alternative would be to update the memory contents in the final bitstream without compiling the design again, but Xilinx currently does not offer a viable tool to achieve this task. There is an utility called UpdateMEM that can be used for changing the initialization data of integrated Xilinx soft-core processors in bitstreams, but it does not support general custom BRAM configurations [Xil18l].

For the reasons mentioned above, using internal memory is deemed unfeasible. When considering external memory as an alternative, the constraints of the Trenz Electronic board available for this thesis must be kept in mind. Only solutions involving the components that are already on the board can be practically implemented.

The components of the board were previously introduced in Section 2.2.2. Relevant memory parts are the 32 MB QSPI flash memory, the 4 GB eMMC, and the 1 GB DDR3 SDRAM chip. All of them are connected to the PS part of the Zynq-7000, making any direct access from the PL impossible. The flash memory and eMMC are too slow for directly streaming bitstreams. The maximum clock of the QSPI peripheral of the Z-7020 is 100 MHz [Xil18q], yielding a theoretical maximum transfer rate of 50 MB/s at 4 bits per clock cycle. For the *Secure Digital* (SD) peripheral (accessing the eMMC), the maximum clock frequency is 50 MHz in high speed mode [Xil18q]. This leads to the theoretical maximum transfer rate of 25 MB/s at 4 bits per clock cycle. Both are way below the 400 MB/s needed to saturate the ICAP. Again, bitstream compression could alleviate this problem by reducing the amount of data to be transferred, but the downsides of this approach (increased resource usage, inability to reach maximum speed) also apply here.

The DDR3 SDRAM is specified as having a maximum read bandwidth of 4,264 MB/s with DDR3-1066 memory [Xil18s, p. 659]. To access it from the PL, one of the AXI buses of the processor as introduced in Section 2.2.2 has to be used. When using the high performance AXI_HP bus, the maximum read bandwidth is 1,200 MB/s at an AXI clock of 150 MHz. This is more than sufficient for the intended purpose of loading bitstreams. Timing-wise, a maximum latency for memory accesses can be guaranteed by making sure that no other components in the SoC

[1] Depending on the algorithm and parameters, the decompression logic might not be able to run fast enough in order to support the targeted configuration data throughput or require excessive amounts of logic resources to do so. Additionally, the decompressor might be stalled on memory reads and thus not be able to deliver optimal speed. This is e.g. true for the implementations compared by Koch in [Koc13].

(such as the CPU or the DMA controller) use the DDR3 SDRAM or the AXI_HP access is given highest priority [Xil18s, p. 128].

Due to the volatile nature of the RAM, the bitstreams cannot be saved there permanently. Instead, they have to be copied from some other persistent storage device on system startup. The eMMC is considered most appropriate for this purpose, since it it does not contain any other data required for the system to function as opposed to the QSPI flash which contains the Zynq boot loader. Still, the eMMC controller is not directly accessible to the logic on the FPGA. In order to copy the bitstream data to the DDR3 SDRAM, a piece of software has to be written that runs once on the PS when the system is initialized and after that turns inert. This component will be referred to as `bin_provider`.

For connecting the PRC and the bitstream storage over AXI, it was briefly considered whether usage of an AXI4-Stream connection instead of a fully-fledged standard AXI bus would be beneficial. AXI4-Stream [ARM10] is used for streaming data into and out of a module without using distinct memory addresses and accesses. On the one hand, it uses considerably less signals and fits the intent of the PRC which is in itself mostly a stream forwarder. On the other hand, accessing memory content needs an intermediary instance like the Xilinx AXI DMA core [Xil18e] for handling data transfer requests and turning them into a stream. This entails a performance penalty and increased resource usage as well as overall system complexity. It was therefore decided to interface with the standard AXI bus directly in line with goal 2.

Figure 3.3 shows the structure of the partial bitstream storage and delivery solution proposed in this thesis containing the components as named and outlined above. Additionally, the control logic that monitors the execution of the ViSARD and triggers partial reconfiguration when appropriate is called `execution_control`. Although it is generally specific to the application, it needs to be implemented at least in a simple variant in order to verify and demonstrate the operation of the whole system.

Assuming a minimal `execution_control` entity that runs a set of programs in an endless loop, the flow of operation beginning with the initialization of the system is as follows:

1. `bin_provider` starts executing on the APU, copies the raw bitstreams from the eMMC to the DDR3 SDRAM 1:1, and puts the APU to sleep.
2. `execution_control` starts executing a program on the ViSARD core.
3. `ViSARD` indicates to `execution_control` that the program has finished execution or has reached a specific point instruction (depending on the application).
4. `execution_control` sends a trigger to start reconfiguration to `prc`.

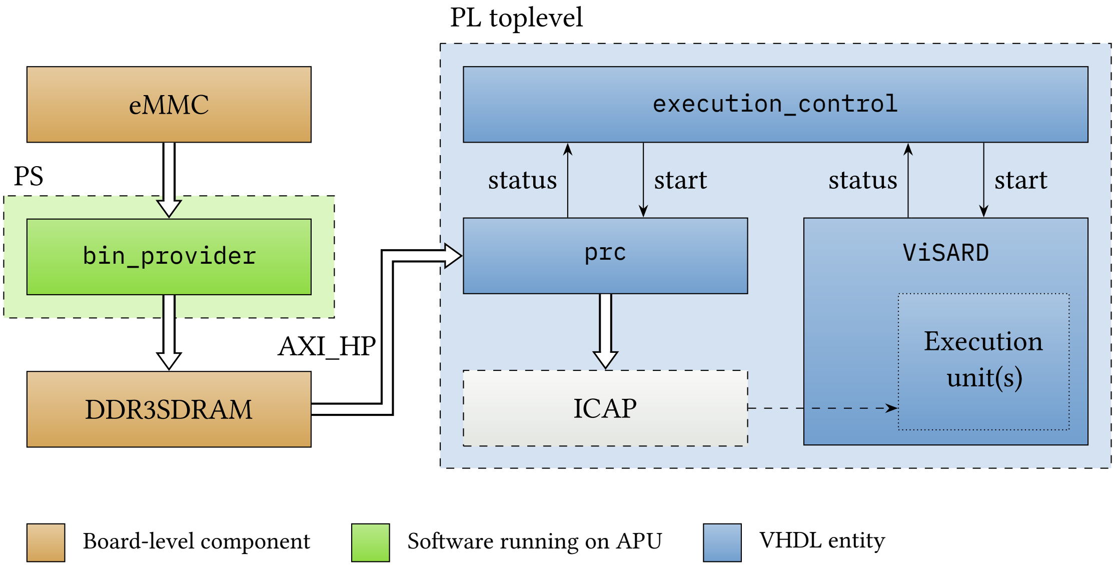


Figure 3.3: Structure of the custom partial bitstream storage and delivery solution. Arrows indicate conceptual flow of information and do not necessarily match the actual port directions.

5. `prc` begins to buffer the partial bitstream by requesting it from the DDR3 SDRAM via the AXI_HP bus of the PS memory controller.
6. `prc` forwards the partial bitstream to the ICAP inside the FPGA.
7. The FPGA configuration controller processes the bitstream and swaps in a different execution unit into the ViSARD.
8. `prc` indicates to `execution_control` when the whole bitstream has been processed, i.e. when reconfiguration has finished.
9. `execution_control` continues at its discretion, for example starting a different program on the ViSARD core that uses the newly inserted EU.
10. Repeat from step 3.

This now might seem like it does not fulfill criterion 1 in the sense that the design is specific to the Zynq SoC, but in fact care has been taken to make it as easy as possible to adapt this design to an FPGA that does not include a hardwired CPU: The proposed PRC loads bitstreams from an AXI bus. Where and how these bitstreams are stored is irrelevant to its operation. Due to the limitations of the available hardware, the Zynq-7000 AXI_HP bus was chosen to deliver the data, but that is not the only option when also considering other devices. Any other storage location that offers access over AXI can be used instead. The bus cleanly separates storage and delivery of the bitstream, so it is easy to replace the storage component only.

A possibility for Virtex 7-based system designs, for example, would be to instantiate an AXI-compatible eMMC and DDR3 SDRAM controller (the Xilinx memory interface generator directly offers AXI access [Xil18p]) in the FPGA design and use an AXI DMA controller such as [Xil18d] to copy the data over. This can be achieved by connecting ready-made components that likely will exist in an advanced design anyway. Using an AXI BRAM controller (e.g. [Xil17a]), it would even be possible to use block RAM for storage without much effort. Furthermore, it is conceivable to keep the current system structure with minimal modifications by running the `bin_provider` software on a soft-core processor (see Section 2.1.2 for an introduction of the principle) instead. Optimally, this solution is used when the soft core is part of the design anyway so as to not waste resources needlessly.

Bitstream delivery

After making sure that the bitstream can be accessed sufficiently quickly not to become a bottleneck, the actual delivery of the data to the FPGA has to be designed so that it can keep up with the flow and stream it to the ICAP at maximum speed.

The ICAP takes 32 bits of data each clock cycle at a maximum frequency of 100 MHz. For simplicity, the AXI bus is assumed to likewise use a 100 MHz clock. The AXI_HP bus of the Zynq-7000 uses a data width of 64 bits, so for each cycle twice the amount of data needed for the reconfiguration is read. This is actually good, since it means that there is a considerable amount of leeway for delivering the bitstream to the ICAP. Nevertheless, some sort of width adaption/buffering between the AXI and ICAP is necessary. Depending on the concrete bitstream provider on the AXI bus, there might be a few clock cycles of inactivity between successive read transactions. Reading faster than strictly necessary allows to use this period of inactivity for bitstream delivery still. Xilinx, for instance, lists the maximum efficiency of the Zynq-7000 DDR memory controller as 97 % for sequential read accesses [Xil18s, p. 660].

For stream buffering, using a *first in, first out* (FIFO) buffer is a very common choice. In this case, the write port is 64 bits wide and the read port 32 bits. Theoretically, the FIFO also enables clock domain crossing to have the AXI reader and the ICAP writer run in two different clock domains at possibly different speeds, but the current design does not make use of this.

Some features that are present in other PRCs are deliberately not implemented since they are not relevant for the purposes of this thesis. This includes bitstream compression (needs more resources and might decrease speed, but cannot increase configuration throughput beyond the device maximum), checksumming (unnecessary since memory used is assumed to be sufficiently reliable), and bitstream/design compatibility checks (only a safety feature, but complicates code).

The overall system layout is most similar to the ZyCAP [VF14]. The ZyCAP likewise caches bitstreams in DDR memory and forwards data to the ICAP by reading from the Zynq AXI_HP bus with a special DMA controller. However, it is designed to be operated from the PS via a separate AXI control port. For the work presented here, it is the other way around. The PS is mainly used as memory controller due to the constraints of the hardware available for implementation. It is not involved in any way in controlling the partial reconfiguration process. The idea of using external RAM to cache the bitstreams for very fast access was also used in other past system designs such as [Cla+10; Liu+09].

Compared with other controllers designed for systems without supporting processor, separation of bitstream storage and delivery modules is a common theme, but the usage of the modern and standard AXI bus to achieve this particularly stands out. The DPRM [Tar+14] explicitly does not use any standard data bus, but requires custom memory controllers to be written. The situation is similar for the ICAP-I [LD09]. Using AXI, the PRC developed in this thesis is readily adaptable to differing systems by replacing the storage module with another ready-made one. During research, only one other standalone partial reconfiguration controller was found which offered AXI for bitstream fetching, which is the one developed by Xilinx itself [Xil18f]. Overall, it is very similar in operation: It waits for a hardware trigger and then streams a partial bitstream from an AXI bus to the ICAP. With all optional features turned off, it still needs a sizable amount of resources (around 1000 LUTs is the reported minimum in a standard configuration [Xil15a]).

In summary, the proposed partial reconfiguration system structure should allow for almost 100 % utilization of the ICAP (goal 5) on the target hardware (goal 1) and be implementable with few resources (goal 4) since the PRC implements the required functionality only. Real-time capability (goal 3) can be included in the implementation that will also focus on concise code (goal 2). Therefore, all design goals can be reached.

The design of fast and reliable partial reconfiguration was one part of the thesis, but the task of integrating it into the ViSARD remains. The next section will explain how this was achieved.

3.2 ViSARD

The goal of this thesis is to reconfigure ViSARD cores on the EU level, i.e. exchange one or more EUs in the core for different ones. A range of considerations surrounding the ALU and the management of reconfigurable partitions have to be made in order to achieve this in an efficient and practically useful fashion. This section describes how exchangeable execution units were integrated into the ViSARD ALU, what further modifications were necessary to the

ViSARD overall, which EUs it makes sense to exchange, and how they could be mapped to RPs and Pblocks.

3.2.1 Related work

Designing soft-core (co-)processors with PR capabilities is a concept that has been described in the literature in the 1990s already. A 1995 paper by Wirthlin and Hutchings [WH95] proposes the Dynamic Instruction Set Computer that can load instruction modules dynamically when they are actually needed by an instruction to be executed in a just-in-time fashion. If all slots are occupied, previously loaded modules are removed starting with the ones whose last usage was the most time ago. One variation of this scheme is the ConCISe [KBH99] that reduces the reconfiguration overhead by compile-time analysis of application code, allowing to combine custom instructions in one configuration. However, the dynamic nature of the approach makes the run time of programs hard to estimate and therefore unfit for hard real-time systems. The ViSARD is designed to be fully predictable at all times.

On a lower level, the Common Minimal Processor Architecture with Reconfigurable Extension (CoMPARE) [SGS98] implements a RISC processor that includes reconfigurable resources (LUTs and switch matrices) in the ALU that can be configured by the program itself. This is not strictly speaking a PR design, but the basic concept is the same. An advantage of this approach is that the compiler can choose to implement operations either with the basic ALU operations or on the reconfigurable unit depending on performance considerations in an highly integrated fashion.

Other works deal with the dynamic partitioning of issue slots between cores in a *very long instruction word* (VLIW) architecture [ANW10] and the dynamic adjustment of the size of the register file [WAN10].

A whole chapter is dedicated to soft cores in the partial reconfiguration book by Koch [Koc13], but the focus is clearly on providing application-specific extension instructions to general-purpose processors although the general idea is similar.

Most similarly to the research at hand, exchanging integer EUs in the open-source LEON3 processor is discussed in [Zai$^+$08]. This specific paper mainly presents an area, energy, and power analysis showing that savings in these aspects can be achieved by PR in practice, further motivating the work in this thesis. The LEON3 processor itself is a 32-bit microprocessor more inclined towards general-purpose computing applications, so its goals vastly differ from those of the ViSARD.

A prior diploma thesis completed in 2010 [Sch10] included a case study in which reconfigurable single-precision FPUs were connected to a general-purpose hard-core processor that was also responsible for managing the PR process. Contrary to the modifications done to the ViSARD in this thesis, the floating-point EUs were accessible over a processor-local bus and not directly part of the ALU, making them inherently less efficient. The experiment was successful and showed the general feasibility of the partial reconfiguration approach for co-processors. Resource usage or performance improvements were not evaluated.

3.2.2 Integration

The ALU in its current form primarily consists of a number of execution units and a big multiplexer to select the result to write back in a given clock cycle. It therefore seems straightforward to put one or more of the EUs into RPs (Figure 3.4), and in principle it is when paying attention to tool limitations. For the designer, selecting which concrete EUs these are would ideally be a simple matter of e.g. adjusting a few generics on the ViSARD instantiation, but the limitations require a more complicated process to be followed. The text below will describe which modifications were made to the code in order to make partial reconfiguration possible and how designers can utilize those to have their custom ViSARD design use the right EUs.

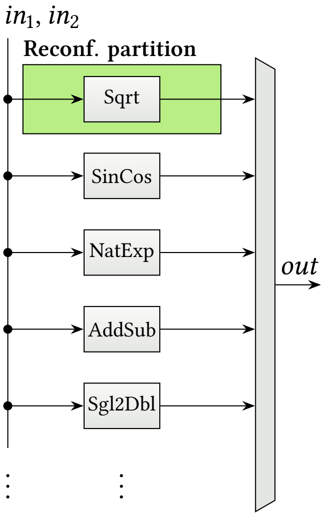


Figure 3.4: Structure of the ViSARD ALU with one exemplary execution unit enclosed in a reconfigurable partition. in_1 and in_2 are input operands. *out* is the multiplexed output operand.

Specifying exchangeable execution units

First and foremost, the EUs must still be sub-components of the ALU so the design hierarchy is kept intact. As the reconfigurable modules have to be defined in the Vivado project, it is not possible to set up a partial reconfiguration project just by editing the VHDL code. The ViSARD in its preexisting static form uses generics (user-defined parameters in VHDL that can influence the function of a component) set on the top instantiation to enable the designer to choose which EUs should be included and available to the code. One Boolean generic is available for each EU (except AddSub) that either enables or disables it. If it is disabled, it does not take up any resources inside the ALU and corresponding instructions do not yield a valid result.

Recalling the Xilinx tool limitations described in Section 2.3.3, it is not currently possible to have any generics defined on a reconfigurable module. This prohibits handling the exchangeable EUs in a universal fashion similar to non-reconfigurable ones. In this thesis, the approach taken instead is that the reconfigurable units and their connections are entered directly into the ALU component according to a process outlined further below in Section 4.6.1. Alternatively, a VHDL code generator could be developed that takes the generics and bakes them into the code (or uses any other feasible generation method, e.g. template-based). This is desirable in the long run, but time-intensive to write and test. More importantly, FPGA manufacturers might include something similar as a feature in their products in the future.

To allow for easy reuse of the existing assembler and infrastructure, all EUs in reconfigurable partitions retain their exact opcode and behavior. Dedicating opcodes to reconfigurable units is an approach sometimes taken when extending general-purpose processors with custom instruction set extensions (described e.g. in [Koc13]). As in the case of the ViSARD the basic instructions are made reconfigurable, the only difference to programs should be whether a mnemonic is available (i.e. will yield a valid result when invoked). Having different opcodes depending on which EUs were chosen for PR would result in a lot of complication both in the assembly process and the VHDL code. Keeping them the same, however, allows to use the same assembler configuration as when compiling for a non-PR-enabled ViSARD instance – the partial reconfiguration process is completely transparent to the program and assembler. Furthermore, ViSARD components depending on concrete opcodes (except for the ALU) such as the operator enable module can stay as-is and need not be modified for PR.

Program and data memories

Exchanging the EUs at run time requires running different programs that only use the features that are available in the current configuration. The ViSARD was not designed for multiple executables: It has one program and one data memory in BRAM that is initialized once when the FPGA is programmed. Three fundamental approaches are considered for enabling multiple programs: Adding memories for each program, rewriting the content of the existing memories during PR, and extending the ViSARD to support different programs contained in the existing memories.

Firstly, if one set of program and data memories is not sufficient, an obvious idea is to add more of them; one per program to execute (and by extension, per EU configuration). This could even be implemented in a generic way: As the count of programs is a parameter of the top ViSARD component and not of any reconfigurable module, a VHDL generic can be supplied to specify the number of memories to create and their respective contents with an accompanying port to select the current program at run time. Still, there is the downside of multiple memories increasing the resource footprint of partial reconfiguration. The program and data memory instances are currently limited to powers of two in depth. Instantiating more of them will lead to increased waste in the form of unused but occupied memory. Even if they could have arbitrary size, BRAM elements on the FPGA come in a set of predefined size configurations, so there will be more unused memory in either case. As the goal in this thesis is to have reconfiguration with minimal overhead, this option was not chosen.

Secondly, as each configuration of execution units typically has exactly one accompanying program/algorithm that makes use of it, the program and data memories could be reprogrammed as part of the partial reconfiguration process. Although this adds additional steps to a PR cycle, it would not necessarily lead to an increase in reconfiguration time. The memory contents can be modified in parallel to reconfiguring the logic if storage memory bandwidth allows. Another positive effect is that the BRAM required by the ViSARD is less than in a non-PR design since the internal memories must only be large enough to fit the biggest program part instead of one program that does everything. Nonetheless, this approach complicates the overall system by making it necessary to make the raw address and data signals of the ViSARD-internal memories available to the PRC, violating design hierarchy. Furthermore, adding another data stream to the reconfiguration process makes the PRC more complex and will require additional logic resources there.

Finally, all programs can share the same program and data memories, but have the ViSARD operate on a different region of them depending on which program is being executed. Since this does not significantly change the resource usage compared to non-PR designs and is easy to implement given the current structure of the ViSARD, this is the approach that was chosen for this thesis.

As a feature for algorithms requiring multiple iterations, it is already possible to supply a non-zero program start address in order to jump directly to a desired location when starting execution. This port can be reused in order to start a different program altogether when all instructions are stored in the same memory.

Missing for this principle to work in practice is the possibility to supply a program end address at run time. At present, it is specified using a VHDL generic at design time. Instead, a port on the ViSARD top entity must be used for setting the end address, identically to what is already being done for the start address.

As the ViSARD does not have any branching or similar instructions, the machine code does not include any program memory addresses that would have to be adjusted for multiple programs. This is different for the data memory for which operand addresses are a part of the program code. Ideally, the assembler would get all program parts as input and automatically make sure that each part gets its own distinct set of addresses so that there are no conflicts. Practical tests have shown that this is not currently possible as the assembler will overwrite constants declared in earlier programs even when all input code is marked `loopable` (i.e. all variables not explicitly written to should retain their original value).

As alternative, all programs can be given to the assembler separately and a port must be added to the ViSARD that acts as an offset added to all data memory addresses encountered in the machine code. This way, each program is under the impression of having its own data memory space starting at address zero.

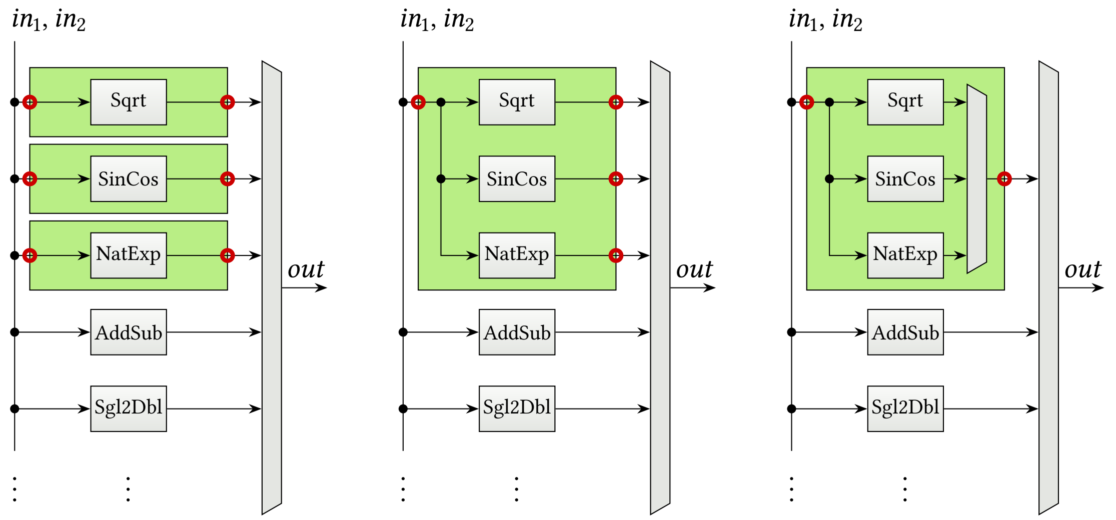
Due to its simplicity and non-invasive nature, the last option of extending the capabilities already present in the ViSARD was chosen for the actual implementation of multi-program support.

3.2.3 Assignment of execution units to reconfigurable partitions

If there is more than one execution unit to reconfigure inside the ALU, the EUs can be put into reconfigurable partitions in a number of ways differing in the amount of Pblocks and partition pins needed. The details of this options are discussed in the following paragraphs.

The most straightforward layout is to create one reconfigurable partition for each EU. As shown in Figure 3.5a, this constitutes a **1:1** mapping of execution units to partitions: Each RP contains strictly one EU, receives the common input values and produces exactly one output value that is given to the ALU result multiplexer just like the static EUs. The similarity to the preexisting static design makes this approach uncomplicated to integrate, but it comes with considerable drawbacks.

Separate RPs per EU imply that a distinct Pblock has to be defined for every EU slot. The Xilinx tools currently do not allow putting multiple RPs into one Pblock, which would in theory be possible and alleviate much of the problems described here. Separate Pblocks intensifies the problem of finding appropriate EUs that have similar resource requirements as each one has to be viewed in isolation. It is not possible to combine multiple EUs in a Pblock and sum up their resource requirements in order to find suitable combinations. This will invariably lead to a higher surplus of resources ending up dormant. Furthermore, the number of partition pins is as high as it can be and there is no meaningful way to have a different number of EUs per configuration because the resources must stay reserved even if an RP is unused.



(a) 1:1: One separate partition per EU with separate input and output operands

(b) 1:n: Multiple EUs combined in one RP with shared input and separate output operands

(c) 1:n with mux: Multiple EUs combined in one RP with shared input operands and one output operand multiplexed inside the partition

Figure 3.5: Options of assigning execution units (gray boxes, selection exemplary) to reconfigurable partitions (green rectangles). Partition pins are marked with red circles. On the input side, each line represents a 128-bit bus containing the combined two input operands. On the output side, each line represents a 64-bit bus containing the output operand. Clock, reset, and clock enable signals omitted.

Each execution unit in this basic layout needs 194 partition pins: EUs typically require two input operands[2] and produce one output value. Each operand is an IEEE-754 double precision floating point number represented by 64 bits. Therefore, 64 pins are needed to transfer the information of one operand. As control information, EUs are handed a clock enable and a reset signal. The additional clock signal is not included in this calculation since it is routed separately and is common to all elements anyway. Given the count of exchangeable execution units n, the total number of partition pins in the design is thus

$$n \cdot (2 \cdot 64 + 64 + 1 + 1) = 194n.$$

A reduction of this is possible by putting multiple EUs into one reconfigurable partition for a **1:n** mapping (Figure 3.5b). Since only one Pblock must be defined in this case, most of the problems of the 1:1 approach disappear and it becomes possible to have a differing number of EUs in each configuration. These can all share the input operands and reset signal, but require

[2] A number of execution units such as the square root operation require only one input operand. If this is the case for all EUs used as reconfigurable modules for a given reconfigurable partition, it is possible to eliminate the port reserved for the second input operand. The calculations in this section will change accordingly, but the general implications stay valid.

separate output operands and clock enables, so the number of partition pins drops to exactly

$$2 \cdot 64 + 1 + n \cdot (64 + 1) = 65n + 129.$$

Finally, the result multiplexer that is part of the ALU can be moved in part to the reconfigurable partition in order to save pins used by the output operands. Only one execution can finish in any clock cycle, so it is sufficient to have this output leave the RP. Figure 3.5c shows that this significantly reduces the number of partition pins. For this scheme to work, a signal to select between the EUs needs to be added. As the clock enable still needs to be separate, the number of partition pins with this solution is

$$2 \cdot 64 + 64 + 1 + n \cdot 1 + \lceil \log_2 n \rceil = n + 193 + \lceil \log_2 n \rceil .$$

This is still proportional to the number of EUs, but almost unnoticeably so (cf. Figure 3.6).

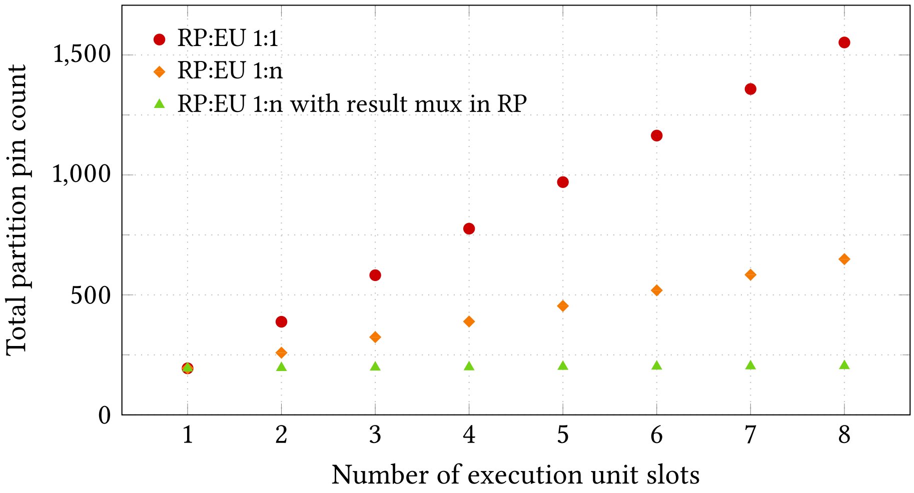


Figure 3.6: Plot of total amount of partition pins in the design with increasing number of execution units for different options of assigning them to reconfigurable partitions. All of the graphs are increasing proportionally to the number of EUs, but the slope is very different.

While both Xilinx [Xil18m] and Koch [Koc13] recommend to minimize the number of partition pins, it is unclear how much of an impact this will have in practice with the modern PR flow. It is clear that fewer pins reduced LUT usage back when LUTs were needed as part of the proxy logic inferred for each pin, but as previously mentioned this is not the case any more and partition pins do not by themselves require any logic resources. Apart from this, possible advantages of fewer partition pins may be improved timing and routing results. One contribution of the present thesis will be to practically evaluate the two 1:n styles mentioned above and verify whether an impact on the resource usage can be registered. The 1:1 style is skipped since its large waste of resources make it entirely impractical for the purposes of the ViSARD.

When multiple ViSARD cores are involved, each of the options for a single core outlined above can be implemented accordingly by replicating the structure of the reconfigurable partition(s) for each core. Alternatively, when multiple EUs are combined in a partition, they can be combined again to form a big Pblock for all processor cores. This will, however, make it impossible to reconfigure cores individually, and may lead to routing issues when a large number of cores is implemented that all need to access a central block of EUs. The approach chosen in this thesis is therefore not to share Pblocks between cores.

3.2.4 Pblock locations and shapes

As explained in Section 2.3.2, the designer has a certain amount of freedom in choosing where on the chip to place the reconfigurable regions. However, there are recommendations that should be followed on Xilinx 7 series devices in order to get the most out of partial reconfiguration and a small number of constraints that have to be followed due to the layout of the device.

The height of clock regions align with configuration frames (see Section 2.1.3). Consequentially, it is usually beneficial to align the upper and lower boundaries of all reconfigurable Pblocks with clock region boundaries. On the one hand, drawing the boundaries freely will allow for a more selective inclusion of resources and avoid having too much dead logic elements in the Pblock, but on the other hand, two potential problems arise: Firstly, static logic that is part of the same configuration frame will get rewritten. In most cases, this is not a problem since Xilinx guarantees glitchless reconfiguration when a configuration cell is programmed with the value it already contained. However, this is limited to static logic that does not contain any memory elements such as BRAMs or LUTs used as shift registers [Xil18m]. Such elements would lose their value. Secondly, the aforementioned static logic needs to be part of the bitstream, even though it is not exchanged. This increases the bitstream size unnecessarily. The clock region alignment constraint only concerns the vertical boundaries of Pblocks. The horizontal boundaries are not affected and it is permissible to span multiple clock regions (in both directions).

Additionally, Pblocks should be strictly rectangular if possible. It is permitted to form irregularly shaped Pblocks (such as U or T shapes), but discouraged by Xilinx due to the associated routing difficulties. The same is true for Pblocks formed by multiple rectangles with gaps between them. [Xil18m]

The columnar layout of the logic resources inside a clock region further necessitates that depending on the logic resource usage of a reconfigurable module, it might not be possible to include only logic (CLB) slices in the reconfigurable resources. If the Pblock needs to cross

BRAM or DSP columns in order to accumulate enough LUTs and flip-flops, all associated resources have to be included and therefore will not be available to the static logic any more. It is generally a good idea to make use of these resources in the dynamic logic as far as possible so they do not lie dormant.

On a lower level, interconnect columns (as previously included in Figure 2.4) internally consist of a pair of adjacent columns of switchboxes. It is forbidden to split them by creating a Pblock that contains only one half, which is called a *back-to-back violation* (Figure 3.7). Effectively, this constraint means that columns can be included in pairs only under most circumstances. The Vivado design suite makes sure that this is followed automatically by excluding interconnect columns on the sides if necessary if the `SNAPPING_MODE` property of the Pblock is configured (recommended). [Xil18m]

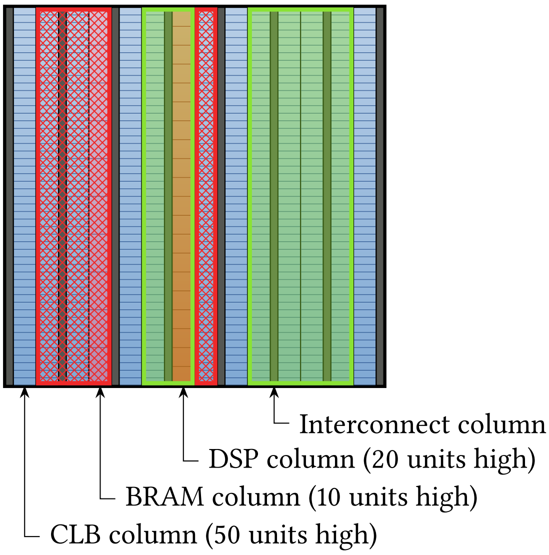

Interconnect column
DSP column (20 units high)
BRAM column (10 units high)
CLB column (50 units high)

Figure 3.7: Valid (green) and invalid (red) exemplary Pblocks with regard to back-to-back violations. The black interconnect columns may not appear at the edge of a Pblock except for a number of corner cases.

When combining the clock region alignment with the avoidance of back-to-back violations, the consequence is that the resource granularity for reconfiguration of logic is 2 CLB columns or 100 CLBs or 800 LUTs and 1,600 flip-flops. It is possible to include a single CLB column only if it is directly adjacent to a DSP or BRAM column (which is also included). For DSP and BRAM, the granularity is 20 DSPs or 10 BRAM tiles since columns of these types are always next to a CLB column.

3.2.5 Examination of execution units for reconfiguration

Which execution units are made dynamically exchangeable and which stay part of the static logic is a choice that the designer can make freely depending on the concrete algorithm or algorithms being implemented on the processor. Nevertheless, it makes sense to discuss this question in a generic sense as a starting point and to illustrate which kind of algorithms can benefit the most from partial reconfiguration. This section will therefore present a number of general observations.

All of the assembly mnemonics supported by the ViSARD were listed in Table 2.1 on page 33. Most of these directly correspond to EUs inside the ALU, but some EUs cover multiple mnemonics and some mnemonics are evaluated directly without involving separate calculations. Table 3.1 lists all execution units available in double precision (single precision is not evaluated in this thesis). There are two distinct groups: Calculation operations and conversion operations.

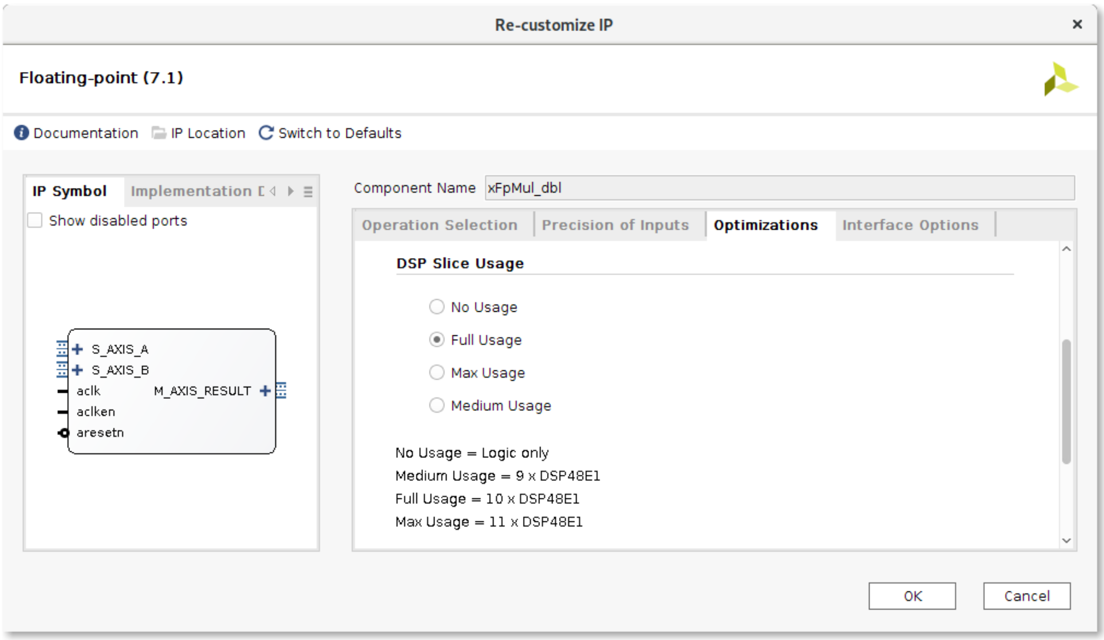
Table 3.1: ViSARD execution units available in double precision and the mnemonics they handle. Data types are: int8 for 8-bit integers, int16 for 16-bit integers, int32 for 32-bit integers, and single/double for single and double precision floating-point numbers, respectively. Abbreviations indicate the name of the component instance in the ALU.

Abbrev.	Operation	Mnemonic(s) handled
AddSub	Addition/Subtraction	`add`, `sub`, `abssub`
Multiply	Multiplication	`mul`
Divide	Division	`div`
Sqrt	Square root	`sqrt`
SinCos	Trigonometric operations	`sin`, `cos`
NatExp	Exponential function	`natexp`
Sgl2Dbl	Conversion: single to double	`sgl_to_dbl`
Dbl2Int	Conversion: double to int32	`dbl_to_i32`
Int2Dbl	Conversion: int32 to double	`i32_to_dbl`
Dbl2Sgl	Conversion: double to single	`dbl_to_sgl`
I162Dbl	Conversion: int16 to double	`i16_to_dbl`
Dbl2I16	Conversion: double to int16	`dbl_to_i16`
I82Dbl	Conversion: int8 to double	`i8_to_dbl`

Conversion operations are general-purpose in nature, require a very small amount of logic slices, and do not make use of DSP and BRAM components at all. It is therefore improbable that any gains can be achieved by exchanging them as required for algorithms or algorithm parts. Furthermore, out of the calculation operations, addition/subtraction is so fundamental that more or less all practically relevant programs will need it. The operations left for further inspection are therefore: Multiplication, division, square root, trigonometric, and exponential function.

For minimizing resource losses incurred by putting logic into exclusive Pblocks, it is important that all of the EUs assigned as modules to a reconfigurable partition have a roughly compatible resource footprint. If, for example, the first EU needs a lot of DSP slices, but the second one that is later exchanged for the first does not use any at all, these resources will lay dormant and are effectively lost to the design as long as the second EU is active. A certain amount of mismatch cannot usually be prevented, but making sure it is sufficiently small is key to a practically useful partial reconfiguration of the ViSARD core.

In order to obtain some numbers to base judgment of this on, all EUs were embedded in a design for the Z-7020 FPGA at a target frequency of 100 MHz and their resource utilization numbers extracted from the Vivado design suite. All EUs are implemented by configuring the Xilinx floating-point operator IP core [Xil17c] with the appropriate options. Besides setting the calculation and operand precision, some operations such as multiplication and the natural exponential function allow additional configuration of the FPGA resource usage: DSPs and/or block RAM can be used instead of regular logic resources for improved device utilization (Figure 3.8).



□ Show disabled ports

Component Name xFpMul_dbl

DSP Slice Usage

○ No Usage
● Full Usage
○ Max Usage
○ Medium Usage

No Usage = Logic only
Medium Usage = 9 x DSP48E1
Full Usage = 10 x DSP48E1
Max Usage = 11 x DSP48E1

Figure 3.8: Floating-point IP configuration example: DSP slice usage setting for the double-precision division operation.

Furthermore, the latency (i.e. the execution time) of the operations is configurable from 0 up to a maximum number of cycles that depends on the other configuration variables. This setting offers a trade-off between latency and maximum operating frequency of the circuit. On the one hand, setting it to zero will yield an EU that in theory computes instantly, but will usually fail timing and not work in practice. On the other hand, setting it to the maximum value will surely achieve timing closure, but with increased latency and at a potentially much higher frequency than actually necessary. The resource usage might also be higher since the higher latency implies that the data has to be stored for more clock cycles, requiring more flip-flops to hold the information. Consequently, the latency should be set to the lowest value possible that is still operable at the desired frequency.

For the sine/cosine operation that is mainly implemented by a different IP core, no BRAM or DSP resources can be used and the latency of the main operation is fixed. However, the precision (in bits) with which the calculation is performed is configurable up to 48 bits and the mode of operation can be selected as serial or parallel. In serial mode, a new calculation can be started only after the previous one has finished, while in parallel mode the same is possible in every clock cycle. More information on the SinCos EU is provided in Section 4.2.

The mentioned configuration options were varied so their influence on the resource trade-off could be evaluated more closely. The results are listed in Table 3.2. The sine/cosine operation presented the problem that at default tool settings, the Vivado design suite was not able to achieve timing closure for 40 bits of precision in parallel mode of operation. This was rectified by switching to the performance-optimized tool preset, but for even higher precision this did not help any more. Since the maximum of 48 bits is desirable for the kind of high-precision calculations that the ViSARD was designed for, serial mode of operation was selected instead for that configuration.

The main conclusions that can be drawn from this information are:

- Using the maximum latency possible indeed does not yield significant (or any) improvements in resource usage (see AddSub and Sqrt as examples).
- Conversion operations are indeed quite small and thus unlikely to benefit from reconfiguration.
- Only the exponential function can make use of BRAM resources.
- Using the exponential function without any DSP and BRAM resources requires roughly 6,000 LUTs, which is about 11 % of all LUTs available in the device. With full DSP usage, this drops to a tenth, but at the same time a large amount of DSP resources become occupied. Considering that in a typical ViSARD core only the EUs can make use of DSPs but every component requires logic slices, it is therefore recommended to enable full DSPs resources in the exponential function IP core configuration. As far as BRAM is concerned, the recommendation is to enable it depending on the design and the other execution units that share the reconfigurable partition: On the one hand, if enough logic slices are available anyway, it would not make sense to force the inclusion of a BRAM column for this one EU. On the other hand, if the design does not otherwise require the BRAM tiles anyway, allocating them to the exponential function might make it possible to free up enough logic resources inside a reconfigurable Pblock to insert another execution unit into it.

Table 3.2: Resource footprint of all double precision execution units extracted from multiple full designs containing all EUs at once. The configuration column indicates the values of key parameters influencing resource usage chosen in the IP configurator. Common settings for all IP: Optimize for speed, no optional output signals. Latencies that are underlined conform to the maximum value possible. Target frequency $f = 100$ MHz. Target device: xc7z020clg484-1. Resource values in parentheses indicate the total amount available on this device. Default tool flow preset unless noted otherwise. EU abbreviations are explained in Table 3.1.

EU	**Configuration**	**Latency**	**LUTs**	**FFs**	**BRAMs**	**DSPs**
			(53,200)	(106,400)	(140)	(220)
AddSub	DSP: None	4	638	350	0.0	0
AddSub	DSP: None	12	650	1,063	0.0	0
AddSub	DSP: Full	4	614	948	0.0	3
Multiply	DSP: None	4	2,657	892	0.0	0
Multiply	DSP: None	9	2,241	2,419	0.0	0
Multiply	DSP: Med.	15	273	559	0.0	9
Multiply	DSP: Full	15	220	498	0.0	10
Multiply	DSP: Max	16	202	490	0.0	11
Divide		30	3,196	3,018	0.0	0
Divide		40	3,251	3,026	0.0	0
Sqrt		28	1,713	1,654	0.0	0
Sqrt		57	1,730	3,230	0.0	0
SinCos	Serial, 32 bits prec.⋆	41	928	548	0.0	0
SinCos	Serial, 40 bits prec.⋆	50	1,159	682	0.0	0
SinCos	Serial, 48 bits prec.⋆	57	1,359	795	0.0	0
SinCos	Serial, 48 bits prec.⋆	64	1,352	1,181	0.0	0
SinCos	Parallel, 40 bits prec.⋆	47		*(no timing closure)*		
SinCos	Parallel, 40 bits prec.⋆†	47	6,183	5,822	0.0	0
SinCos	Parallel, 48 bits prec.⋆†	55		*(no timing closure)*		
NatExp	BRAM: None, DSP: None	20	6,061	3,979	0.0	0
NatExp	BRAM: None, DSP: Med.	20	2,049	1,112	0.0	15
NatExp	BRAM: None, DSP: Full	20	1,853	1,022	0.0	26
NatExp	BRAM: Full, DSP: None	20	5,209	3,702	2.5	0
NatExp	BRAM: Full, DSP: Med.	20	1,122	1,029	2.5	15
NatExp	BRAM: Full, DSP: Full	20	828	919	2.5	26
NatExp	BRAM: Full, DSP: Full	57	831	1,953	2.5	26
Sgl2Dbl		2	17	68	0.0	0
Dbl2Int		3	217	139	0.0	0
Int2Dbl		3	169	112	0.0	0
Dbl2Sgl		2	35	72	0.0	0
I162Dbl		3	70	63	0.0	0
Dbl2I16		3	133	89	0.0	0
I82Dbl		2	39	31	0.0	0

⋆ Optimal pipelining, radian phase format, no coarse rotation (input limited to first quadrant), round to nearest even

† Flow presets: Flow_PerfOptimized_high for synthesis, Performance_ExploreWithRemap for implementation

- Division and parallel sine/cosine are by far the largest operations in terms of logic usage because they are complicated to implement and cannot be compacted by off-loading calculations to hard BRAM and DSP resources.
- Varying the DSP usage level for the Multiply operation has minimal effect as long as DSPs are used in some fashion.
- Multiplication with DSP usage enabled is small (one CLB and one DSP column), so making it reconfigurable by itself needs to be considered carefully.
- Including additional useful operations in the ViSARD such as the natural logarithm, the reciprocal, or the reciprocal square root would enable more diverse options of combining EUs for partial reconfiguration.
- Examples of EU combinations with sufficiently compatible resource footprints are:
  - NatExp (no BRAM, medium DSPs) vs. Sqrt,
  - NatExp (full BRAM, medium DSPs) vs. SinCos (serial, 40 bits prec.), and
  - Divide vs. Sqrt and SinCos (serial, 48 bits prec.) vs. Sqrt, NatExp (full BRAM, medium DSPs), and Multiply (full DSPs).

The list above contained general hints and conclusions that can be drawn from comparing the currently existing execution unit implementations. In summary, the resource usage as listed in Table 3.2 in combination with the aforementioned recommendations should allow designers to make a reasonable decision on what EUs to include in partial reconfiguration and how to configure the corresponding IP cores.

3.2.6 Impact on resource usage

In Xilinx 7 series FPGAs, each logic slice contains 4 LUTs and 8 flip-flops for a LUT:FF ratio of 1:2. However, they can only be interconnected in a restricted fashion. Every LUT is on its output side associated with the input sides of 2 flip-flop storage elements[3] [Xil16a]. Effectively, this means that even when one part of a design does not make full use of the FFs in the logic slices it occupies, they are nevertheless not practically available to other parts as well.

As seen in the directly preceding section in Table 3.2, the resource requirements of the execution units of the ViSARD rarely come close to the optimum LUT:FF ratio. In fact, they often require fewer FFs than LUTs, leaving many FFs unoccupied. A preliminary investigation of the ViSARD resource usage indicated that this also holds for a complete, configured core with the ratio being approximately 1:0.75.

[3] The LUT outputs can also bypass the FFs and feed directly into the routing fabric.

All ViSARD resource results measured in implemented designs will therefore report two times the LUT requirement as FF usage when the LUT:FF ratio is below 1:2. This more realistically represents the amount of flip-flops that are really occupied and unavailable to other components, enabling the designer to gauge what influence partial reconfiguration will have on design resource utilization more fairly.

It is possible in principle to bypass the LUT and feed one signal directly into either or both of the two associated FFs. All design FFs placed into the same slice must, however, always have completely identical clock, clock enable, and set/reset signals [Xil16a]. This restriction and routing locality considerations will usually prohibit different design parts sharing flip-flops within a slice. The ViSARD makes liberal use of local clock enable signals for power saving and typically has a separate core clock and reset line, so it is highly unlikely that significant amounts of unrelated FFs can be put into slices occupied by it.

The design of the partial reconfiguration system including the custom PRC and why it was deemed necessary to develop it were presented in this chapter along with considerations on how to enable the ViSARD to have execution units exchanged at run time. Chapter 4 will continue in that direction by talking about the practical implementation of the aforementioned components and additional utilities.

Chapter 4

Implementation

As it is now clear how the proposed system is going to be structured and what components need to be created and modified, the implementation chapter will describe the practical aspects of this process. It starts with preparatory work on the ViSARD including the addition of the SinCos execution unit and continues with the partial reconfiguration controller, its interface, and its storage format. Moreover, the related software utilities `bin_packer` and `bin_preloader` will be introduced followed by a description of how to practically leverage the new PR capabilities aimed at designers wanting to use the ViSARD in their project. Finally, there will be a brief discussion of different partial bitstream generation methods that will later be compared in the tests.

4.1 Preparations

In preparation for implementing the PRC from scratch and modifying the ViSARD, an example Vivado project containing several ViSARD cores and supporting code was copied and updated to the Vivado version 2018.2 used in this thesis. After that, all IP cores were similarly updated to their newest version. The only obstacle encountered was that the maximum latency of a floating point operation had changed, making the currently configured value invalid. This was easily resolved by setting a different value and reconfiguring the ViSARD and assembler accordingly. The updated IP cores were used as basis for inclusion in all newly created Vivado projects.

Furthermore, all floating point IP cores were changed from blocking to non-blocking mode. Blocking in this context means that the core will wait for both input operands to be available before starting the calculation, allowing them to arrive in different clock cycles. Inside the ViSARD ALU, the operands are guaranteed to always be available at the same time, so this feature is not used. Since blocking mode consumes more resources than non-blocking mode [Xil17c], applying this setting allows to reduce resource usage[1] without any negative effect.

Another set of modifications around the ALU was to actually implement the changes in opcodes and flags as presented in the ViSARD ALU/FPU operations section (page 34) and add the implementation of the `sin` and `cos` instructions.

4.2 Sine/cosine execution unit

Although they were already defined as operations, the processor was not able to perform trigonometric calculations because no EU had been included to handle them. The new SinCos execution unit was built to fill this gap. As the floating point IP core used for all other operations [Xil17c] did not include any trigonometric ones, the *coordinate rotational digital computer* (CORDIC) IP core [Xil17b] was chosen to fulfill this rule. It allows calculating the sine and cosine using an iterative algorithm with a very high amount of precision of up to 48 bits (configurable).

The CORDIC uses fixed-point numbers for input and output, so conversions from floating to fixed point on the input side and from fixed to floating point on the output side were added to the EU (shown in Figure 4.1 for the maximum precision). These conversions were implemented using the standard floating point IP core [Xil17c]. Fixed-point numbers are divided into an integer part and a fractional part that can have their widths tuned to the range of numbers they should be able to hold. For the CORDIC, the integer part is always 3 bits wide on the input and 2 bits wide on the output regardless of the precision of the calculation. Both sides include a sign bit as for the sine and cosine operations both the input and output can be negative.

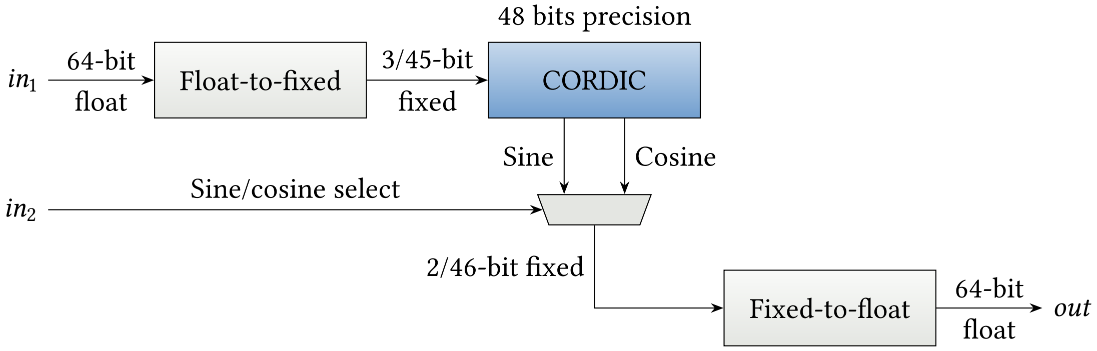


Figure 4.1: Data flow inside the SinCos execution unit highlighting the data type conversions. Fixed number precision is indicated by the width of the integer part (including a sign bit) and the width of the fractional part separated by a slash. This illustration assumes the maximum of 48 bits of precision for the CORDIC.

[1] The resource usage figures in Table 3.2 presented previously already include this reduction.

This structure is the characteristic difference to floating-point numbers that do not store an integer part at all, but instead include an exponent field that specifies to which power of two the fractional part should be taken. Their precision is fixed and equal to the width of the fractional part plus 1 bit (i.e. 53 bits for double precision). Although this does allow them to represent a much larger range of values without sacrificing precision, calculations with floating-point numbers are much more time- and resource-intensive[2], which is why the CORDIC algorithm is usually implemented with fixed-point calculations [Ort+03].

Both sine and cosine are calculated at the same time by the algorithm. However, the ViSARD can currently only write back one result per clock cycle. Which value should be considered the output of the EU is given in the first bit of the second input operand that would otherwise be unused. As this information is relevant in the clock cycle the CORDIC produces the output to a given input value and not when calculation is started, the select input of the multiplexer is delayed by the combined latency of the float-to-fixed conversion and the CORDIC IP core.

Prior practical tests (Section 3.2.5) have indicated that the IP core cannot always be operated in a fully-pipelined parallel computation mode like all other current EUs are, but for highest precision has to be switched to serial mode instead due to timing issues. This is unproblematic as long as only one operation is started for testing purposes or optimization is disabled in the assembler, but for real-world usage the assembler would have to be modified to honor the restriction that only one trigonometric calculation can be in progress at all times.

The last consideration concerning the SinCos EU is whether the achieved precision of 48 bits in calculation is appropriate for the purposes of the ViSARD which is conceived as co-processor for highly precise calculations. As the ViSARD works with 64-bit numbers when double precision is selected, it appears like 16 more bits of information would be needed. This can, however, not be compared easily since the IP operates on fixed-point numbers which are represented differently as explained above. The smallest difference expressible by the 46-bit fraction of the CORDIC output is fixed at $2^{-46} \approx 1.4 \cdot 10^{-14}$, which is also the smallest non-zero number representable. The smallest non-zero double precision floating point number is $2^{-1022} \approx 2.2 \cdot 10^{-308}$.

As the CORDIC algorithm requires one more iteration per additional bit of calculatory precision, hundreds of iterations would be necessary to reach a comparable level for all possible input values. Table 3.2 already showed that resource usage of the IP core in parallel mode of operation grows rapidly. Having even more iterations is not realistic at least with the implementation offered by Xilinx. It is worth mentioning that the lack of range is most problematic for the sine operation when numbers are extremely small[3]. When looking at input numbers

[2] Many operations allow to treat fixed-point numbers the same as or very similarly to integers.

[3] This applies likewise to both input and output numbers, since the sine operation on very small numbers is approximately equal to the identity function. With the cosine operation, small inputs produce values around 1, for which floating-point numbers do not offer a significant range advantage.

between 0.1 and 1, the fixed-point representation offers between 42 and 45 bits of precision. This is a loss of only 8 to 11 bits compared to the 53 bits of a double floating-point number.

Alternative approaches use e.g. precomputed LUTs or Taylor series approximations, but they similarly cannot reach the full range of precision of floating point numbers. It is therefore the opinion of the author that the CORDIC module represents an appropriate compromise between precision, speed, and resource utilization fit for the ViSARD. [NDB05; Ort^{+}03; DdD07]

4.3 Partial reconfiguration controller

The custom PRC as introduced in Section 3.1.2 is responsible for loading partial bitstreams from an AXI bus (described in Section 2.2.1) and piping them to the FPGA Internal Configuration Access Port with maximum speed, low resource usage, and predictable timing characteristics. All these goals will have consequences for the implementation described here.

4.3.1 Interface

The PRC was hand-coded in VHDL (like the ViSARD). Appendix A.1 contains the source code of the main `prc` entity as well as the packages `types` and `utility`. Seen from the outside, the PRC interface consists of three generics and a number of ports. In detail, the generics specify parameters of the AXI and how many bitstreams can be addressed. All of them are listed in Table 4.1. The ports (Figure 4.2) fall into four categories: Clock/reset (`clk_i`, `reset_i`), partial reconfiguration control (`bitstream_index_i`, `req_i`, `ready_o`, and `done_o`), AXI (master interface; only read address and read data channels as the PRC never writes to memory), and ICAP (only clock enable and 4-byte data word). All control ports are active high.

Table 4.1: Custom PRC generics and their functions

Generic name	Function
`AXI_ADDRESS`	Base address where partial bitstreams are located on the AXI bus
`AXI_ADDRESS_WIDTH`	Bit width of addresses on the AXI bus and the corresponding `axi_araddr_o` output port (default 32)
`BITSTREAM_INDEX_BITS`	Bit width of the `bitstream_index_i` input port that specifies which bitstream to load (this generic limits the number of addressable bitstreams to 2 to the power of its value)

The AXI ports nominally follow the AXI3 version of the protocol because that is the standard provided by the Zynq-7000 on its internal buses. The changes in the newer AXI4 are sufficiently minor, so that the PRC could be directly connected to an AXI4 slave, too.

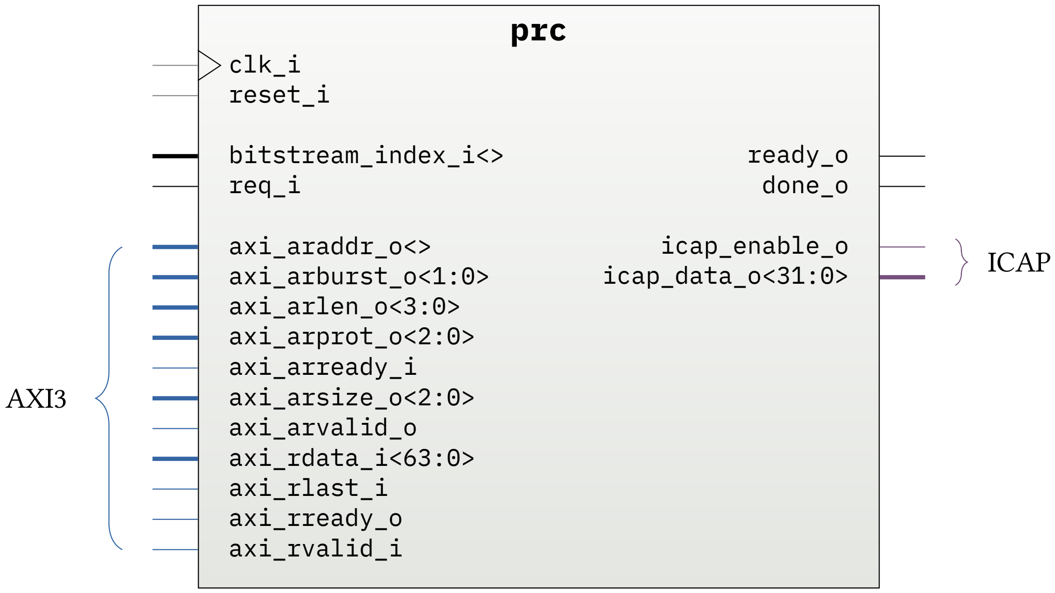


Figure 4.2: Custom PRC ports. Ports on the left-hand side supply information to the PRC, while ports on the right-hand side supply information to the embedding system. The actual VHDL port direction may differ and is indicated by an `_o` suffix for outputs and an `_i` suffix for inputs. Angle brackets indicate ranges of vector ports. If they are empty, the range is user-specified using the associated generic (Table 4.1).

The ICAP primitive of the target device (see Section 2.3.2) has to be instantiated by the user and connected appropriately, i.e. it must use the same clock as the PRC and have the `I` data port connected to `icap_data_o` and the active-low `CSIB` clock enable port to the inverse of `icap_enable_o`. Some prior theses that used the Xilinx ICAP primitive reported difficulty in achieving working partial reconfiguration in practice due to timing issues [Siv10, p. 12; Seg13, p. 36]. Those could not be observed during the work performed for the present thesis. It is unclear whether this is by chance or due to improved tool support.

The user-facing control interface is very simple to use (Figure 4.3): When the PRC is ready to accept a request for reconfiguration, it asserts `ready_o`. Requests are set by asserting the corresponding input `req_i`. It is possible to keep this port high when the PRC is not ready yet. It will ignore the request until it is, or put differently: Reconfiguration will begin in any clock cycle in which both `ready_o` and `req_i` are high. `ready_o` will typically become low immediately after that as the PRC needs to process an incoming request first before accepting another one, but this behavior is not mandated by the interface and cannot be relied on.

Which partial bitstream to load is specified using the `bitstream_index_i` input accepting an unsigned zero-based number. When reconfiguration has completed, `done_o` becomes high for the duration of one clock cycle. This is guaranteed to happen only after all data was written to the ICAP, which in turn guarantees that the reconfigured region is ready to be used when the event is received. `ready_o` is asserted again whenever the PRC is able to begin processing another request. Specifically, this might be the case when a reconfiguration that is currently

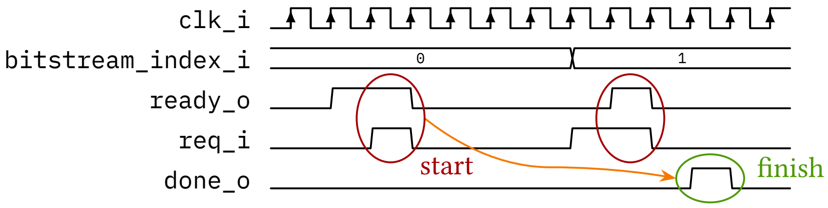


Figure 4.3: Exemplary timing diagram of requesting partial bitstream transfers from the custom PRC. As indicated, the PRC may be ready again even if the last requested bitstream transfer was not finished yet. A transfer is started as soon as the PRC is ready and `req_i` is asserted.

underway is nearing its end, so that another partial bitstream can already begin preloading for reduced latency between successive reconfiguration cycles. How this was achieved is explained further below.

4.3.2 Bitstream storage format

Partial bitstreams are raw streams of commands and data for the FPGA configuration controller (cf. Section 2.1.3). They are generated as one file per reconfigurable module/configuration (shown in Figure 2.12) and depending on the options used to create them and the resources the logic occupies, they can differ in size. For the PRC to be able to quickly access and load bitstreams, all of them must be available in a contiguous region of memory and additionally have to be indexed to allow locating the start address of an arbitrary bitstream without performing multiple look-ups or skipping through other bitstreams in order to find the correct one.

The index format used in this thesis is a plain table of all available bitstreams (Figure 4.4) that starts at offset 0 in the memory allocated to the partial bitstream storage (given with the `AXI_ADDRESS` generic, see Table 4.1). Each bitstream gets one 8-byte entry consisting of two 4-byte unsigned integer values: The address of the first byte of the bitstream as offset to the base address (0) and its size in bytes. All entries are stored successively, enabling instant lookup of the values of one bitstream by loading 8 bytes of data from the base address plus 8 times the index of the desired bitstream. The actual bitstreams are stored without any post-processing and likewise in succession, but aligned to 4 KiB so AXI transactions are guaranteed never to cross a 4 KiB boundary which is forbidden by the specification. Contrary to the Xilinx PRC [Xil18f], this technique makes it possible to test bitstreams of different sizes (e.g. generated with different methods and/or settings) without having to perform any modifications to the PRC configuration or the implemented netlist.

4.3.3 Data transfer architecture

The PRC contains three state machines, each responsible for one channel of information: the AXI address channel, the AXI data channel, and the ICAP channel. The address channel *finite*

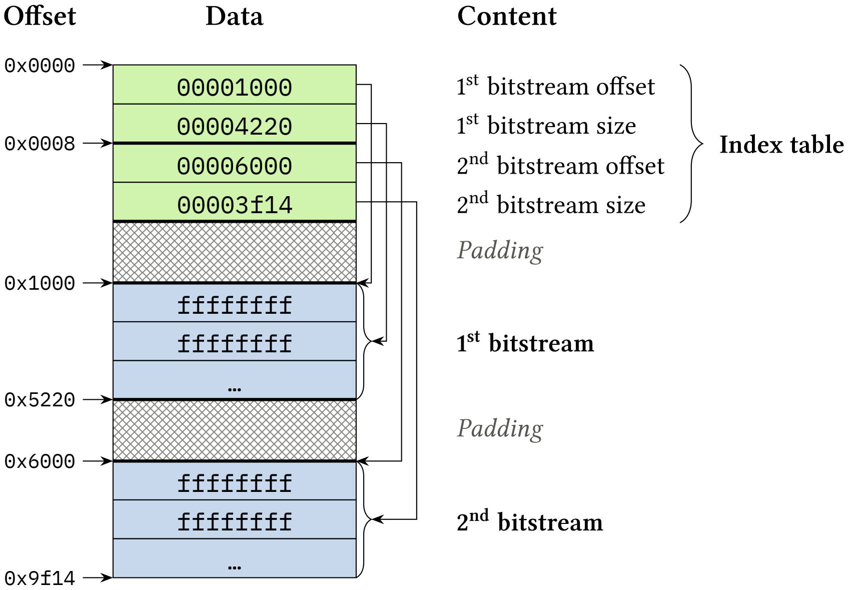


Figure 4.4: Packed bitstream structure consisting of 2 partial bitstreams. The index table grows as more bitstreams are added. All numbers are in hexadecimal notation. Data is organized in 4-byte words both in the index table and the bitstream. The content of the padding is undefined, but usually zero.

state machine (FSM) is the first to act when a bitstream transfer is requested: It instructs the AXI slave to supply the respective entry (8 bytes) in the bitstream index table on the AXI data channel. After that, it is idle until this information has arrived. The data channel FSM will remember both the offset and the size of the bitstream, reset the internal counters, and write the size to the FIFO in order to send it to the ICAP state machine. It is needed there so that the end of the bitstream is recognized and `done_o` can be asserted once all bytes were transferred to the FPGA configuration controller.

As soon as the start address of the bitstream to load is known, the address channel state machine continuously issues read requests for bitstream data independently of the data channel. Each time the AXI slave acknowledges that it has received one such request, the desired address is advanced by the size of the transfer (64 bytes total: burst size of 8 times 8 bytes each) until the end of the bitstream is reached. The AXI data channel FSM forwards the received data to the FIFO where it is consumed by the ICAP FSM and in turn streamed to the ICAP.

This approach ensures that the queue of operations the AXI slave has to perform is always full, which is critical for maximum performance because it allows the slave to effectively pipeline operations and start fetching more data while writing out the response to a request. The contrary approach of always finishing one data transfer before requesting the next bytes in the bitstream would potentially stall the whole reconfiguration process while waiting for the AXI slave to provide more data.

As soon as the address channel FSM is in an idle state when all memory read requests have been issued (or after reset), the `ready_o` port goes high to indicate that another entry can be fetched from the bitstream index table, starting the next partial reconfiguration process. Beginning the next reconfiguration is possible even when the ICAP is still busy and `done_o` was not asserted yet. In that case, the desired bitstream is already requested from the AXI bus and put into the FIFO buffer when space is available such that PR can start as soon as possible. This technique minimizes the time between reconfigurations under certain circumstances (the next bitstream to load is already known and the reconfiguration can begin immediately after the preceding one without affecting the running design) and will later be referred to as *bitstream preloading*.

To reduce implementation complexity, the sizes of the bitstreams and their contents are sent to the same FIFO. The ICAP FSM reads from this FIFO whenever there is data to retrieve. When a new bitstream is started, it must first read two units of data (the bitstream size and a dummy word) because the FIFO write width is fixed to 64 bits while the read width is fixed to 32 bits, i.e. it is impossible to put one isolated 32-bit unit of data into the FIFO. In practice this means that even if the FIFO is completely filled with data to send to the ICAP, there will be a pause of exactly 2 clock cycles between successive bitstreams in which no reconfiguration takes place.

A different architecture such as using two separate FIFOs for sizes and data or handing the bitstream size to the FSM in some other way would theoretically allow for continuous delivery of bitstream data without any pause, but this was deliberately avoided. The current solution is easy to understand and review and when considering the time and clock cycles typically needed for one reconfiguration cycle, the two cycles of forced pause between reconfigurations is negligible. The impact of this decision is verified in practice later on (Section 5.3.2).

For low resource usage, the depth of the FIFO buffer has to be small. The writer is significantly faster than the reader and a great depth would waste resources. Throughout this thesis, a value of 16 64-bit units is used, which is the minimum the Xilinx FIFO generator supports.

4.3.4 Real-time capability

The implementation as outlined above has predictable timing characteristics and also satisfies the hard real-time requirement. Given an AXI slave that answers requests in continuous bursts in at most d clock cycles, the PRC needs a maximum of $2 \cdot d + 4$ clock cycles from `req_i` being asserted to the first data word being sent to the ICAP:

1. The request for the bitstream table index entry is put on the AXI bus.
2. The AXI slave receives the request and needs d cycles to answer.

3. In the cycle in which the answer arrives, the first chunk of bitstream data is requested and the bitstream size written to the FIFO.

4. The AXI slave receives the request and again needs *d* cycles to answer, during which the ICAP FSM reads the bitstream size from the FIFO.

5. In the cycle in which the answer arrives, the first 8 bytes of bitstream data are written to the FIFO.

6. The ICAP FSM reads 4 bytes from the FIFO and puts them on the ICAP data bus.

Assuming that the AXI slave can sustain the required throughput and given a bitstream size of *n* bytes, the total time needed for a reconfiguration process until `done_o` is asserted is at most $2 \cdot d + 3 + \frac{n}{4}$ clock cycles. As far as DRAM controllers are concerned as AXI slaves, they typically have a high *d* for individual requests, but achieve sufficient transfer rates far exceeding the required 400 MB/s when they have many operations to process in their queue. The PRC ensures that the queue is always full, so the upper bound stays valid.

4.4 Bitstream packer

The `bin_packer` utility merges multiple partial bitstream files into one indexed file according to the requirements of the PRC as described in Section 4.3.2. It was implemented as command-line utility in the Python programming language because the ViSARD project already includes some Python-based tools such as a processor simulator, so it is not an entirely foreign technology in the ViSARD world. Appendix A.2 lists the full source code.

When run with the `--help` option, the software prints information about its usage:

```
$ bin_packer.py --help
usage: bin_packer.py [-h] output bitstream [bitstream ...]

Pack multiple partial bitstreams into one indexed file

positional arguments:
  output      output file
  bitstream   partial bitstream in binary format to include in package

optional arguments:
  -h, --help  show this help message and exit
```

It indicates that the invocation on the command line requires the names of the output file and at least one input file. Usually, two or more inputs will be given since DPR with only one partial bitstream is a rare edge-case that is not considered in this thesis.

When run with appropriate arguments, `bin_packer` will combine the files and output useful debug information such as the offsets and sizes of the bitstreams included:

```
$ bin_packer.py bitstreams bitstream1 bitstream2
Packing 2 bitstreams to bitstreams...
0: @0x001000 size 0x020000: bitstream1
1: @0x021000 size 0x020000: bitstream2
Bitstreams successfully packed
Final size: 266240 (0x041000) bytes
```

4.5 Bitstream preloader

The software `bin_provider` running on the Zynq APU has the task of copying the partial bitstreams from the on-board eMMC to the DDR memory for easy and fast access from the programmable logic (see Section 3.1.2). It was developed using the Xilinx SDK for bare metal, i.e. without supporting OS. This choice is motivated by the fact that the functionality of the program is very basic and it would not make use of any advanced features offered by e.g. Linux. Moreover, operating systems typically include a technique called virtual memory in order to isolate processes from each other. This is unneeded in this case and makes it harder to access designated physical memory locations from normal applications, which is necessary in order to make the bitstreams available on the AXI_HP bus.

In detail, `bin_provider` performs the following operations when executed (observable in `main` function of software, see code in Appendix A.4):

1. Initialize the *universal asynchronous receiver-transmitter* (UART) peripheral for printing debug messages on the serial connection.

2. Initialize the SD/*MultiMediaCard* (MMC) controller.

3. Read a fixed number of blocks from eMMC starting at offset zero to the target DDR memory location specified as constant in the code (currently `0x10000000`). This location has to match the one that the PRC is set to use for reading the data later on and the amount copied has to be large enough to encompass all bitstreams.

4. Flush APU data caches. Some data may otherwise stay in the caches of the APU and never be written to the external memory, rendering it inaccessible over the AXI_HP bus and thus to the PL.

5. Set the multiplexer guarding access to the PL configuration module to ICAP so the FPGA design can perform self-reconfiguration (see Section 2.3.2).

6. Mark all DDR memory as non-secure in the `TZ_DDR_RAM` register so it can be accessed from the PL over the AXI bus with regular transactions (see [Xil14, pp. 13, 21]).
7. Put APU to sleep (see [Xil18s, p. 683]).

The SDK compiler produces a standard *Executable and Linkable Format* (ELF) file that can be run on the Zynq either directly from the IDE via USB connection to the GigaZee test board or programmed to the on-board QSPI flash. The bitstreams have to be written to the eMMC by some other means, e.g. special commands for the U-Boot boot loader present on the board. Practical tests will use this approach (experimental setup described in detail in Section 5.2.1).

4.6 ViSARD integration

The general process of how the ViSARD has to be modified for DPR was laid out in Section 3.2.2. The considerations here will focus on the practical aspects of how users can adapt their ViSARD-based design for making use of partial reconfiguration.

4.6.1 Reconfigurable partition inside the ALU

In order to make execution units exchangeable with each other, first of all every EU involved in PR must be disabled by setting its corresponding generic to `false` in the ViSARD instantiation. This makes sure that the static version of the EU is not included and does not take up extra resources.

After that, one VHDL entity per RM has to be prepared containing the associated EUs. An example incorporating the Sqrt, NatExp, and Multiply EUs is shown in Figure 4.5. Although the specific EUs will be different, the general layout stays the same. All arithmetic EUs are available as entities with a unified interface wrapping the IP they require to perform their calculation.

The interface ports of the RP should be as described in Section 3.2.3: An obligatory clock pin `clk`, two 64-bit input operands `data_a` and `data_b`, one `enable` pin per EU, and a shared `reset` pin on the input side and a 64-bit output operand per EU on the output side. As a requirement for DPR, the ports of all RMs must be completely identical. Consequentially, there may be unused inputs and/or outputs when a particular set of EUs does not include any that need a second input operand or when RMs contain a differing number of EUs.

At first, it is recommended to only add the specific RM to the static design that is expected to be the most challenging in implementation as it will form the baseline for the others. This RM

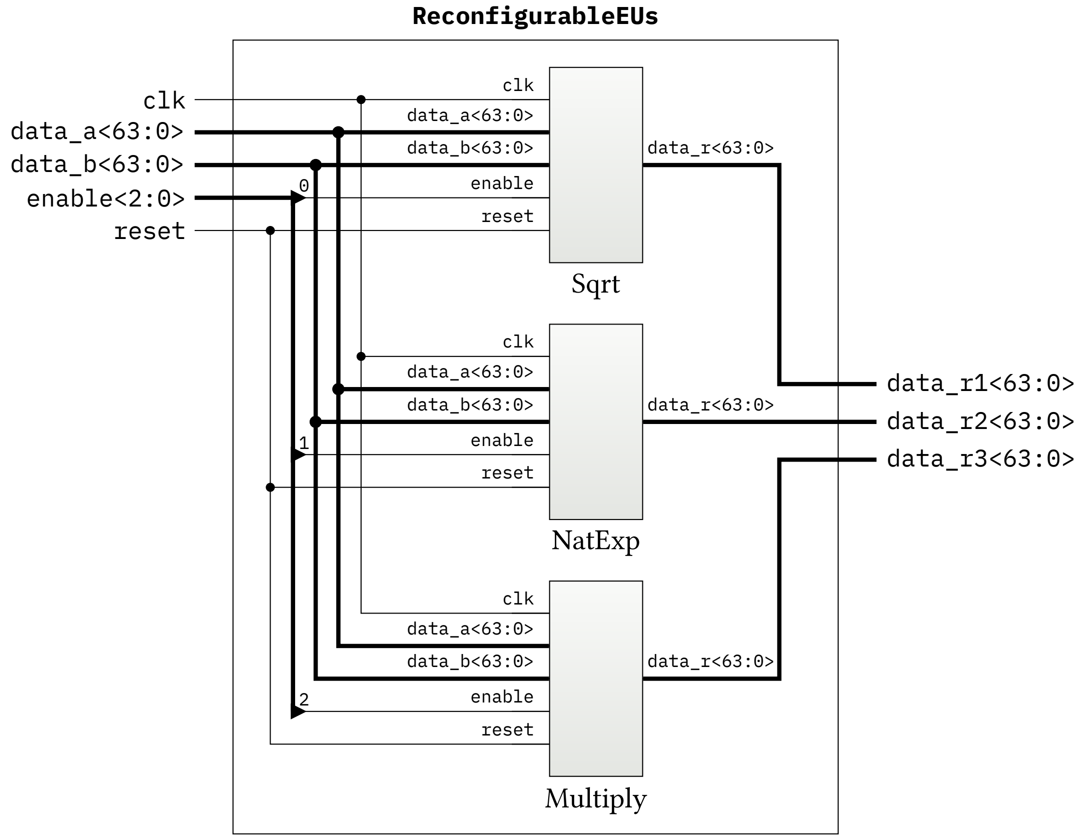


Figure 4.5: Exemplary reconfigurable module entity containing the Sqrt, NatExp, and Multiply execution units. Ports on the left-hand side are inputs, while ports on the right-hand side are outputs. Angle brackets indicate ranges of vector ports.

must be instantiated in the ALU pretending that it acts as placeholder for the union of all EUs per slot. Consequentially, the clock enable signals must each be a Boolean OR-combination of all clock enables relevant to their respective EU slot. As an example: If there are two RMs of which the first one contains the Sqrt EU in the first slot (as is the case in Figure 4.5) and the second one contains the SinCos EU in the same place, the first enable signal given to the RP should be `sqrt_en OR sincos_en` (in VHDL syntax). This OR operation does not enable the EU more than necessary (which would increase power consumption) because when the ViSARD is processing a program with a given EU configuration loaded, instructions for non-present EUs cannot work and consequentially their enable signal will never be asserted.

Furthermore, each output of the reconfigurable partition has to be stored in a separate signal reserved for this purpose. The last modification of the ViSARD itself is to replace the result inputs to the ALU with the corresponding outputs of the RP for the affected opcodes.

After that, the project can be converted to the PR flow and the instantiation of the reconfigurable EU component in the ALU converted to a reconfigurable partition (see [Xil18m] for details). After that, all further reconfigurable modules can be added and the PR wizard run in order to generate the PR configurations and implementation runs.

If one EU is used in multiple RMs, the IP cores it might use have to be copied and renamed for every instance due to the inability of the Vivado IDE to use the same IP in more than one RM. As the instantiation of the IP in the EU entity is bound to the name of the core, that entity likewise has to be copied and adjusted. This cannot presently be avoided.

During the practical work on this thesis, difficulties with IP in reconfigurable modules were encountered. The solution to these problems was to remove the files in question from the project and re-add them directly to the RM by editing the sources in the RM definition window. Furthermore, Vivado often reported that "IP was locked by [the] user" and that that may cause problems although no lock was placed intentionally. This prevented any changes to the settings of the affected cores. This situation could be successfully resolved by executing the command `set_property USER_LOCKED 0 [get_ips <ip name>]` on the Tcl console.

4.6.2 Memory concatenator

The ViSARD received additional ports for the end address of the program (`Prg_End_Add`) and the offset to use in the data RAM (`Data_Add`) according to the previous considerations for combining multiple programs to execute in one set of program and data memories (see Section 3.2.2). For the user, the problem remains that the ViSARD assembler produces one pair of output files (memory contents) per program. Combining these manually every time the code or assembler configuration changes would be tedious. As solution, another small utility program named `mem_concatenator` was implemented, again in Python for the same reason as `bin_packer` (Section 4.4). Appendix A.3 lists the full source code.

When run with the `--help` option, the software prints information about its usage:

```
$ mem_concatenator.py --help
usage: mem_concatenator.py [-h] output input [input ...]

Concatenate multiple ViSARD memory initialization files

positional arguments:
  output      output file
  input       ViSARD memory initialization file

optional arguments:
  -h, --help  show this help message and exit
```

Very similarly to `bin_packer`, it requires an output file as well as at least one input file as parameters. The memory contents of the files are concatenated in the order they are given and written to the output. The header of the file that indicates the number of words is updated accordingly to reflect the increased size. If only one input file is given, the output and input is identical.

When run with appropriate arguments, `mem_concatenator` will combine the files and output useful information such as the offsets and sizes of the memory files included:

```
$ mem_concatenator.py t-prog-c0.txt natexp-prog-c0.txt sqrt-prog-c0.txt
Concatenating 2 memory files to t-prog-c0.txt...
@    0:    36 lines from natexp-prog-c0.txt
@   36:    44 lines from sqrt-prog-c0.txt
@   80: --end--
Concatenation successful
```

These numbers can be used to derive the program start and end addresses for the ViSARD. In the example above, the first program would run from address 0 to 35, while the second one would start at address 36 and end at 79. The resulting program and data memory files must be set as memory initialization files on the ViSARD instance in the `g_ProgMem_Init` and `g_DataMem_Init` generics.

4.7 Partial bitstream generation

Partial reconfiguration requires partial bitstreams, which can be generated in a number of different ways despite the result having the same function and effect. Differences exist in the methodology used to derive the bitstream and in their number and size. This section introduces the module-based and difference-based approaches as well as compression performed by Xilinx tools and finally torCombitgen, which tries to achieve a compromise of both general approaches.

4.7.1 Module-based generation

The most basic option that Vivado will perform when using the PR flow is a *module-based* approach that writes one bitstream per reconfigurable module that contains the data for all configuration frames encompassed by the Pblock associated with the reconfigurable partition and ignores everything else in the design. As all frames are included unconditionally, this type of bitstream is largest in size.

4.7.2 Xilinx compression

A moderate size reduction is possible by enabling a type of compression in the Vivado IDE that makes use of a special feature of the configuration processor of Xilinx FPGAs. It is possible to use the special and largely undocumented (cf. [Xil18c]) command Multiple Frame Write to write identical data to multiple frame addresses. This command takes up a lot less space compared to spelling out identical content each time. As this approach (hereafter referred to as *Xilinx compression*) operates on the configuration processor level, it does not require any decompression before handing the bitstream to the ICAP and reduces reconfiguration time by the same factor it reduces the bitstream size (assuming constant throughput of configuration data). It is essentially free in terms of resource usage, effort, and drawbacks to activate the compression option in the Vivado IDE, so this is always recommended.

How much of an impact this bitstream alteration will have is hard to predict and depends on the contents of the RMs. The highest amount of savings can be expected when logic slices are unused (empty) and when BRAM tiles use identical initialization data; this also applies to unused and therefore zero-initialized BRAM.

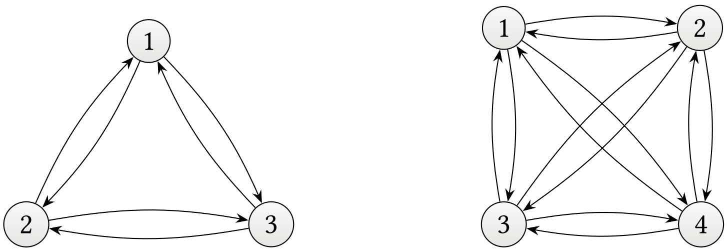
4.7.3 Difference-based generation

A completely different approach to generating partial bitstreams is the *difference-based* one that is largely oblivious of individual RMs. It takes two full top-level bitstreams including the static design and one or more RPs as input and compares all configuration frames to find which ones differ from each other. Only those are written to the partial bitstream file. Compared to the module-based method, this has the advantage that potentially fewer frames have to be written in case there are frames that are identical between the two analyzed designs. Of course, all frames containing only static logic will be the same and therefore excluded, so that the bitstream is never going to be larger than an equivalent one generated by the default module-based approach. [CMS06]

Similarly to the Xilinx compression option, identical frames in dynamic logic are most likely to be found when logic is unused. Additionally, smaller parts of logic that are the same in multiple RMs might be removed, but it is unclear if this is going to have a measurable impact in practice. Slight differences in any part of an RM might cause the placer to put all instances at completely different spots, yielding vastly different configuration data also in unrelated areas.

A disadvantage of this approach is that only differences between exactly two designs can be processed. When three or more RMs have to be considered, a bitstream has to be generated for every pair of top-level bitstreams so that it is possible to switch from any RM to every other one. The number of partial bitstreams necessary for n top-level ones (illustrated in Figure 4.6) is $n(n-1) = n^2 - n$, i.e. it is quadratic in n.

Vivado supports generating bitstreams with this method by using the `-reference_bitfile` option of the `write_bitstream` Tcl command documented in [Xil18j].



(a) $n = 3$ top-level bitstreams requiring $3 \cdot 2 = 6$ partial bitstreams

(b) $n = 4$ top-level bitstreams requiring $4 \cdot 3 = 12$ partial bitstreams

Figure 4.6: Relation of full designs (nodes) containing different RMs and partial bitstreams (edges) necessary for switching between all possible configurations with difference-based bitstream generation. Full and partial bitstreams form a complete graph. Adopted from [CMS06, fig. 7]

4.7.4 torCombitgen

In order to eliminate the need for a partial bitstream per RM pair, the tool Combitgen proposed in [CMS06] takes all relevant full bitstreams as input and identifies frames that are identical in each of them. These are removed to produce one partial bitstream per top-level one. Again, the static logic is guaranteed to disappear as it must be the same in every file. This approach represents a middle ground between plain difference-based generation that requires lots of bitstreams and module-based generation that does not take potential size savings by identical frames into account. As such, the size of the bitstreams (or number of configuration frames they contain) should lie between those two extremes. Contrary to the pure difference between two top-levels, all bitstreams produced by Combitgen will have the same size.

The expectable savings are not accurately predictable, but the previous considerations stay valid. An additional point is that as the number of RMs increases, the bitstreams will quickly become closer in size to the module-based approach as it grows more and more unlikely that frames identical in every single RM can be found at all. The paper [CMS06] reports an experimentally identified time saving of 3 to 4 % in an exemplary design that consisted of a clock divider with three different division factor options as RMs. This seems to be mostly owing to an additional optimization that Combitgen performs on the bitstream by converting regular frame writes into multiple frame writes because that allowed removing one frame of padding per write on the target platform.

Combitgen was developed for the now obsolete Virtex-II series of FPGAs, but the general idea stays valid even with contemporary devices [Cla11]. As part of a previous advanced seminar carried out by the author, this idea was reimplemented using the open-source framework *Tools for Open Reconfigurable Computing* (Torc) [Inf] for reading, manipulating, and writing Xilinx bitstreams. This way, all FPGAs supported by Torc are supported in this tool (called torCombitgen) with minimal effort. Torc supports most Xilinx FPGAs up to 7 series devices, including the one used in this thesis. However, the multi-frame write compacting capabilities of Combitgen are not reproduced as it is unclear whether this kind of transformation would be valid on all target devices, especially given the lack of clear documentation.

Practical tests will be performed with all of the generation methods mentioned above including torCombitgen in order to assess if size savings are possible with the current generation of design implementation tools, devices, and their slice layout. torCombitgen has not previously been tested with anything except a proof-of-concept design similar to the clock divider in [CMS06], so it will be interesting to see how the principle holds up in the practical tests in Chapter 5.

This concludes the explanation of how the partial reconfigurability of the ViSARD was implemented. After describing preparatory tasks and the newly introduced SinCos execution unit, all important aspects of the PR system were shown: The PRC, its interface especially towards the execution control unit, and the accompanying utility `bin_packer` were presented. Next, the `bin_provider` software to run on the Zynq APU was introduced, followed by information for designers using the ViSARD on how to leverage the new PR capability in their designs including invocation of the `mem_concatenator` utility. The last section discussed different bitstream generation methods that will also be featured in the next chapter, which shows how and with what results the work performed for this thesis was tested.

Chapter 5

Test and results

This thesis consists of two major components to be programmed or modified: The custom PRC and the ViSARD. Both must function well on their own and in combination. This chapter presents the tests that were performed in order to prove the practical feasibility of partial reconfiguration of the ViSARD. For each test, its specific goals and setup as well as the results and their interpretation are presented.

The ViSARD has a well-established codebase that is proven to work in practice. For the PRC, this is not true however. Thus, the first step was to test the PRC in isolation to verify it is working before continuing with the ViSARD integration. Three integration tests were performed that combine both components in different scenarios: The first test is conceived to show that the basic functionality of replacing one execution unit by another one works. The second test adds an additional ViSARD core and activates bitstream preloading to show that very high reconfiguration throughput can be reached. The third and final test goes back to one core, but has a more diverse set of EUs to exchange in order to show the actual potential and resource savings of partial reconfiguration.

5.1 Simulation of the partial reconfiguration controller

A simulation of the PRC was conducted firstly in order to verify its behavior. This was performed in a simulation environment as opposed to real hardware because it allows for a drastically reduced iteration time and enables viewing the waveforms of all design-internal signals, which is not as easily possible on hardware.

5.1.1 Test setup

By running this test, it should be possible to show the following:

- The bitstreams are combined into one file including a size and offset table by the software `bin_packer` running on the designer's PC.
- The PRC is parsing and using this data to load and stream partial bitstreams.
- The FIFO and state machines integrating the AXI and ICAP sides inside the PRC are working.
- The PRC follows the AXI3 protocol specification.
- The principle of loading data via the Zynq AXI_HP interface works.

Behavioral simulation is the most basic type of simulation that is closest to the original HDL source code and enables the best debugging capabilities. It is chosen for this test because all goals can be reached without going further down into more involved types of simulation that work on the gate level such as post-synthesis (see Figure 2.5). No actual reconfiguration takes place as simulating partial reconfiguration of logic cells is not currently supported by the tools due to the complexity involved. In a behavioral simulation, accurately modeling PR behavior is impossible in principle because the simulator operates on the register transfer level. This level of abstraction is not concerned with hardware internals such as LUTs and flip-flops. Statements as to the practical functioning of partial reconfiguration will therefore be left for further tests. The Vivado project used may in fact not even enable the PR flow as that would instantly disable all simulation functionality (see Section 2.3.3).

As the PRC cannot be tested at all without an AXI bitstream provider, using the Zynq AXI_HP bus like in the real system allows to observe behavior that is close to what can be expected on hardware. A behavioral model of the Zynq PS that acts as AXI slave and can provide arbitrary user data on request is required for this. This exists in the form of the Zynq-7000 Verification IP provided by Xilinx [Xil17g] that is automatically used in Vivado for simulations of designs that include the PS as functional block. The contents of the memory readable over the AXI buses of the Zynq must be written to simulated memory inside the verification IP by calling special functions (such as `write_mem`) on the IP in the SystemVerilog language before they can be read by AXI masters. An added benefit of this approach is that every AXI bus present on the IP includes a protocol checker that will abort the simulation should any violation be detected.

It was initially difficult to get the IP working as intended due to undocumented requirements, but these problems were successfully resolved[1].

The test data to load was generated by combining two 131,072-byte dummy bitstreams with the `bin_packer` tool (invocation and output in Appendix B.1.2) that each contain ascending series of 32,768 4-byte numbers. Without any real reconfiguration taking place, the actual content is of no importance. Its primary purpose is to show that it is being sent to the ICAP correctly and in entirety.

The main entity of the test design called `top_sim` only contains the PRC, the ICAP primitive instantiation, and a small delay process that starts perpetually requesting PR after reset has been deasserted for a few cycles. It is connected to the first AXI_HP bus of the Zynq-7000 PS. The simulated clock rate is 100 MHz so that the times measured in the virtual waveforms are comparable to the real system.

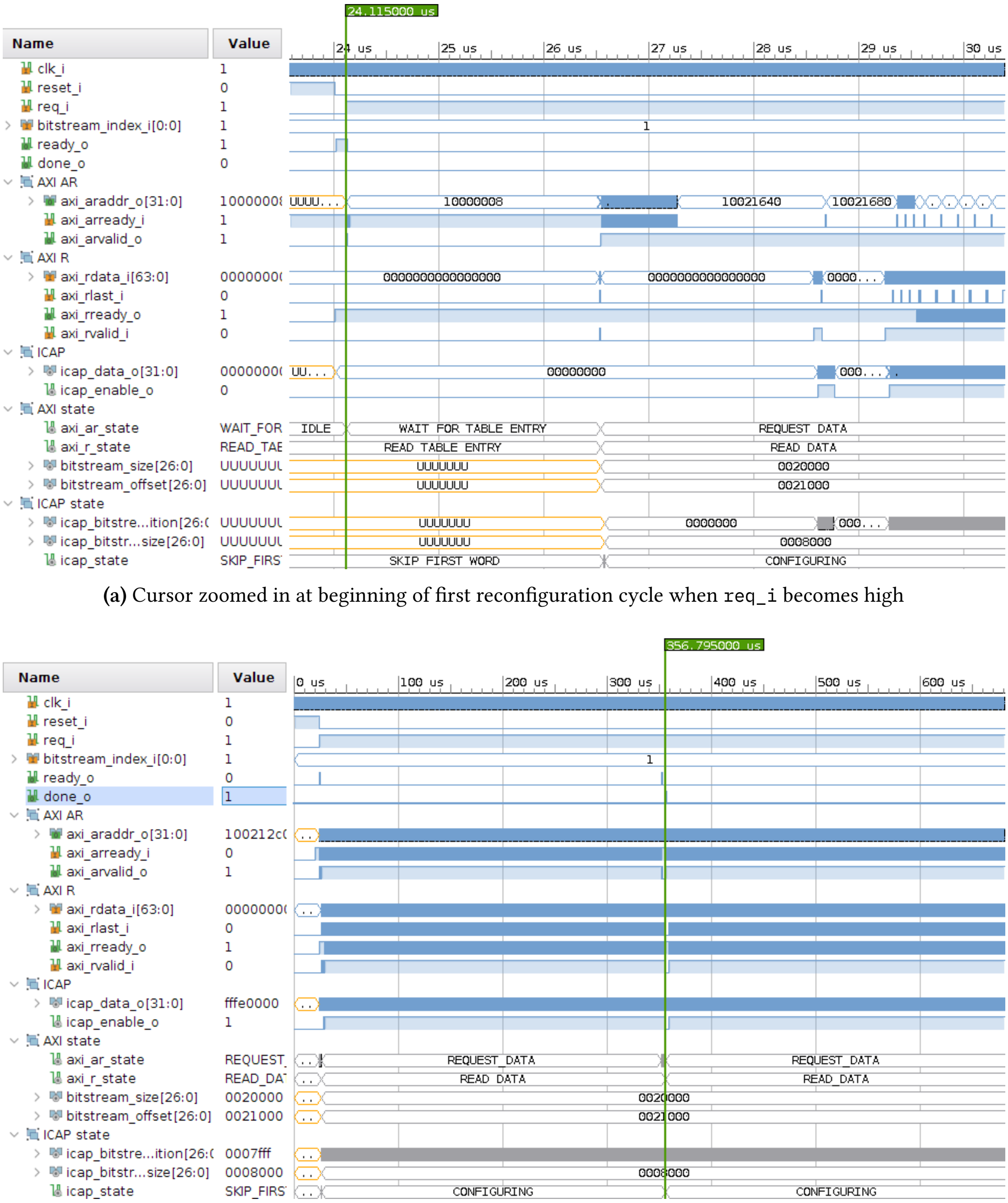
5.1.2 Results

The file produced by `bin_packer` was manually inspected and confirmed to conform to the intended layout. All PRC functionality and behavior could be verified in behavioral simulation. No AXI protocol violations were recorded. Figure 5.1a shows the start phase of the first reconfiguration request after reset. The waveform allows observing all behavior described in Section 4.3.3: After starting to process the reconfiguration request at exactly 24.115 µs simulation time when both `req_i` and `ready_o` are asserted, the AXI read address channel FSM in turn requests the corresponding entry from the bitstream index table. Since the AXI base address was set to `0x10000000` hexadecimal, bitstream index 1 (i.e. the second bitstream) was requested, and one entry is 8 bytes large, the address of `0x10000008` observed in `axi_araddr_o` is correct.

Then both AXI state machines wait for the information to arrive and nothing changes until at around 26.5 µs. The table entry is received by the read data FSM, processed, and stored in the internal `bitstream_size` and `bitstream_offset` signals. It is also written to the FIFO, where the ICAP FSM fetches it, divides it by four and remembers it in `icap_bitstream_size`. The division is performed because the ICAP operates on 4-byte units and not on individual bytes.

At the same time, the read address FSM issues requests for subsequent memory locations starting at `0x10021000` (first address on `axi_araddr_o` after `0x10000008`, not visible in the picture),

[1] The exact constraints are unclear, but it seems that for the AXI_HP bus to answer at all, it is necessary to have the design top-level be a block design directly including the Zynq-7000 PS IP. The detailed steps were as follows: The block design HDL wrapper was generated as Verilog file (not VHDL). The SystemVerilog test bench resets the verification IP by calling `fpga_soft_reset` with a true value as argument on the PS instance and successively sets `ARESETN` low. After waiting for 1,000 clock cycles, `ARESETN` is set high again and data loaded into the virtual memory. Finally, `fpga_soft_reset` is called with a false value to get it out of reset. See Appendix B.1.1 for the complete code of the test bench.



(a) Cursor zoomed in at beginning of first reconfiguration cycle when `req_i` becomes high

(b) Overview of full first reconfiguration cycle, cursor at finish when `done_o` becomes high

Figure 5.1: Simulated waveforms showing the custom partial reconfiguration controller perform a reconfiguration. All blue signals are ports of the PRC as shown in Figure 4.2 and explained in Section 4.3.1. The state signals marked in gray in the bottom part of the pictures are internal to the PRC. All numbers are shown in hexadecimal notation.

which is the start address of the second bitstream. The Zynq verification IP queues these requests until it runs out of space in the queue at around 27.3 µs indicated by `axi_arready_i` becoming low. The read data FSM is always ready to receive and forward data to the FIFO after the index table entry has been read. When the first burst of data arrives at around 28.6 µs (as indicated by `axi_rvalid_i`), the ICAP subsequently receives it from the FIFO, asserts `icap_enable_o`, and begins sending the data to the FPGA configuration processor.

After this burst has been processed, the PRC has to wait for data from the AXI bus and the FIFO runs empty, so the data flow to the ICAP stops. From the second burst onwards, a continuous stream of data is flowing. Periodically, the FIFO becomes full and the PRC for some time refuses to accept incoming data (i.e. `axi_rready_o` goes low). This is expected behavior since data is arriving at approximately twice the rate the ICAP can process it.

The waveform of the whole reconfiguration process (Figure 5.1b) furthermore shows that no wait states on the ICAP are encountered after the initialization phase as demonstrated by `icap_enable_o` staying high. Moreover, the PRC becomes ready and accepts the next request a few cycles before reconfiguration finishes at exactly 356.795 µs. There were 228 cycles of inactivity on the ICAP between the first two successive reconfigurations that are lost to waiting for data on the AXI bus. This number is rather high, but might be owing to the characteristics of the simulated AXI bus.

In total, the first reconfiguration took $356.795\ \mu s - 24.115\ \mu s = 332.68\ \mu s$. The average configuration data throughput was $131{,}072\ \text{bytes}/332.68\ \mu s = 393.988\ \text{MB/s}$ or 98.50 % of the 400 MB/s maximum of the ICAP. Given that the AXI_HP bus of the real Zynq-7000 is unlikely to have an access latency that is worse than the functional model used in this test and real-world bitstreams will usually be larger (reducing the effect of access latency), this is a promising result. The next section will verify this assumption in practice and add the ViSARD into the setup.

5.2 Practical verification of basic functionality

In order to assess the basic feasibility of dynamic partial reconfiguration in combination with the ViSARD, a working example was implemented on the target hardware and its functionality verified in practice.

5.2.1 Test setup

This system has the following structure: One ViSARD core plus the custom partial reconfiguration controller is put into a design together with a basic execution control module. One EU slot inside the ViSARD is designated as reconfigurable partition with associated Pblock

(Figure 5.2) and two reconfigurable modules: One calculating the natural exponential function and the other one the square root. The execution control switches back and forth between two programs to execute on the ViSARD, each applying one of those calculations to a fixed value. Before the execution of a new program is started, the PRC is instructed to load the partial bitstream of the EU that the corresponding code needs.

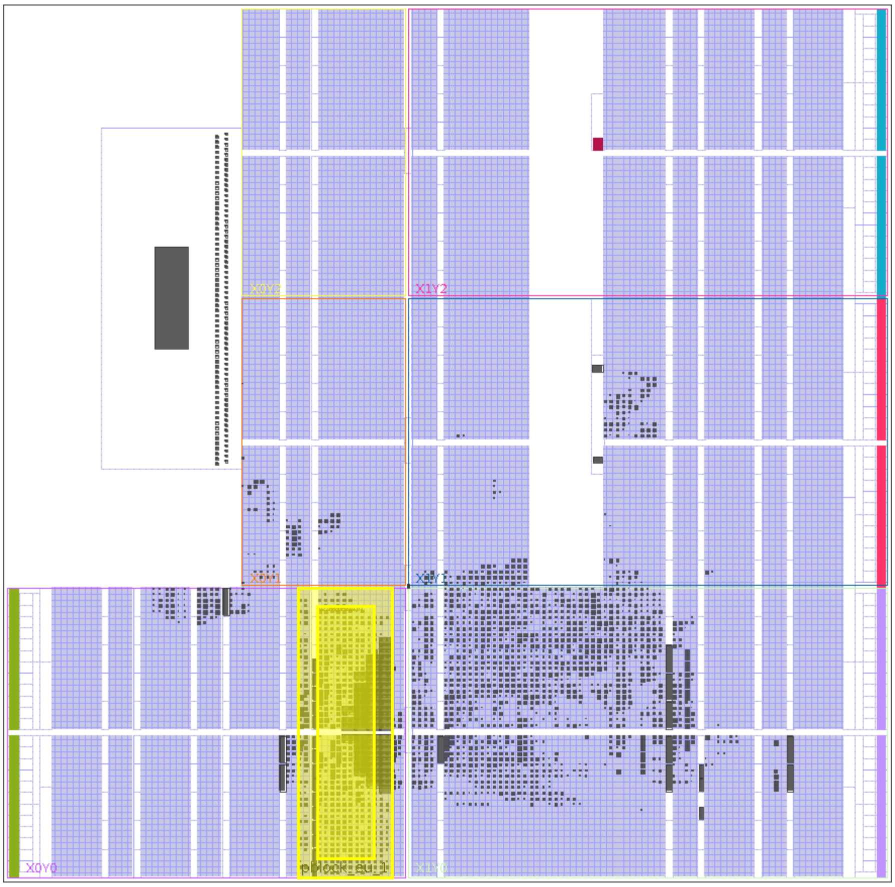


Figure 5.2: Device overview highlighting the Pblock (outlined in yellow) assigned to the reconfigurable partition of the execution units exchangeable in the basic test as seen in the implemented full NatExp design. White vertical columns running through the device contain BRAM or DSP resources (not visually distinguished by the Vivado IDE which generated the image). Resource blocks that appear in dark gray are occupied by the design. Inside the Pblock, larger gray rectangles additionally mark where interconnect resources are reserved for partition pins.

With this test setup, in addition to the statements checked with the simulation in the previous test it can be verified that:

- The software `bin_provider` that copies bitstreams from the eMMC to the DDR3 SDRAM works.
- The principle of loading data from external DDR3 SDRAM via the Zynq AXI_HP interface works.

- The target FPGA can be reprogrammed with a different EU and the ViSARD is able to use it in calculations.
- The custom PRC reaches its design goals in practice, namely that it has low resource usage, predictable timing, and that it achieves high throughput and low latency (see Section 3.1).
- The torCombitgen program introduced in Section 4.7.4 produces usable bitstreams in practice. The full potential of the bitstream minification it performs cannot be evaluated with the example at hand since three or more partial bitstreams are required in order to obtain a methodical advantage over the traditional difference-based bitstream generation that is available in Vivado.

The first test program to run on the ViSARD is:

```
dq testin ? 20
dq testout ? 0

natexp testin ? testout
out testout ? 0
```

It calculates e^{20}, so the output should be 485165195.409790278. As double-precision IEEE-754 floating point number in hexadecimal notation, this is `41bceb088b68e804`[2]. The NatExp EU was set to use no BRAM, a medium amount of DSP resources, and a latency of 20.

The second test program to run on the ViSARD is:

```
dq testin ? 144
dq testout ? 0

sqrt testin ? testout
out testout ? 0
```

It calculates $\sqrt{144}$, so the output should be 12. As double-precision IEEE-754 floating point number in hexadecimal notation, this is `4028000000000000`. The Sqrt EU was set to a latency of 28 cycles.

A Xilinx *Integrated Logic Analyzer* (ILA) IP core was inserted into the design in order to be able to record and display internal signals. This debug core can capture any signal in real time, sample it to BRAM inside the device, and forward all data to the designer's computer for inspection (see [Xil16c] for more information). The limited amount of internal memory available

[2] To easily perform this conversion, one of the various internet pages calculating it on-the-fly in the browser window can be used.

on the target hardware does not allow capturing a whole reconfiguration cycle. However, the ILA is capable of capturing a small window of data when triggered by a configurable condition and immediately re-arm and capture another small window at the next trigger. This way, the beginning and the end of the reconfiguration process can be captured, which are the important phases for the purposes of this test.

A problem presented by this approach is that the resulting waveform does not contain any information on the amount of time that elapsed between the two triggers and – by extension – between start and end of reconfiguration. As this would prohibit making any statement about reconfiguration speed and throughput, a monotonically increasing 32-bit counter was added to the design top-level and included in the recorded waveforms. The counter as well as the whole design runs at 100 MHz and the difference of its value at any two sample points of the waveform will indicate how many cycles have elapsed between them.

Additionally, a dedicated `statistics_collector` component was implemented that monitors the reconfiguration process, counts the cycles between PR request and finish, and finally writes the value to block RAM on the FPGA. This process is repeated until the memory is full after 65,536 samples[3]. This BRAM is available to the Zynq APU via AXI for easy readout, which was performed by a modified `bin_provider` software that sends all recorded values over the serial connection every few seconds. The data received allows to draw conclusions regarding the performance of not only one specific instance of reconfiguration, but the general statistics of the process.

The following steps were taken to prepare the test environment (the invocation and output of important steps can be found in Appendix B.2 for reference):

- Compile both ViSARD test programs and combine their respective program and data memories with `mem_concatenator` (Appendix B.2.1).
- Set up the project in Vivado by following the PR tutorial [Xil18k] and designating the first reconfigurable EU slot in the ViSARD core as reconfigurable partition (named `eu_1`) and the NatExp and Sqrt EUs as reconfigurable modules.
- Activate header-less full binary bitstream generation for all implementation runs[4].
- Implement the full FPGA design including bitstream generation, resulting in two full and two partial bitstreams that each contain either the NatExp or the Sqrt EU. The partial

[3] The AXI BRAM controller [Xil17a] only supports memory sizes that are powers of 2. As the target FPGA contains 612.5 kB of BRAM, the largest usable size is 512 kB. The rest of the design (PRC, ILA, ViSARD, etc.) needs BRAM as well, however, so the next possible lower value of 256 kB was chosen as compromise. Using 4-byte unsigned integers, this amount of memory allows exactly 65,536 samples to be stored.

[4] Vivado Tcl commands:

```
set_property STEPS.WRITE_BITSTREAM.ARGS.BIN_FILE true [get_runs impl_1];
set_property STEPS.WRITE_BITSTREAM.ARGS.BIN_FILE true [get_runs child_0_impl_1];
set_property GEN_FULL_BITSTREAM 1 [get_runs child_0_impl_1]
```

bitstreams are used for partial reconfiguration. The first full bitstream initially configures the FPGA. The second full bitstream is needed for testing the difference-based partial bitstream generation methods that can operate only on full designs at the moment.

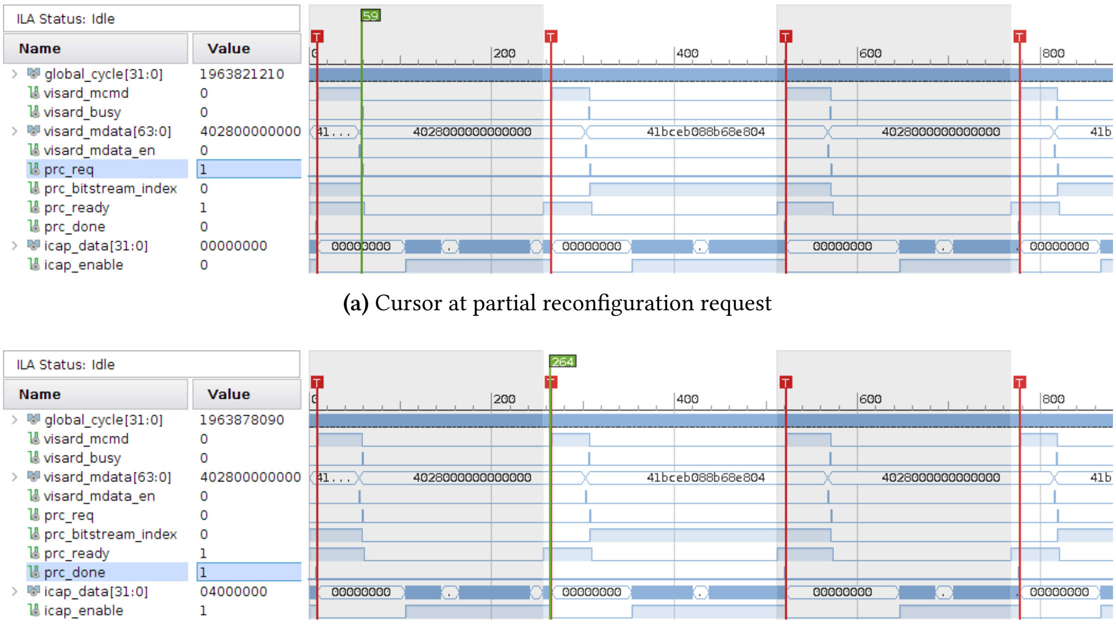
- Run the `bin_packer` utility to combine the partial bitstreams into the `bitstreams` file (Appendix B.2.3).
- Compile `bin_provider` using the Xilinx SDK, resulting in the executable ELF file `bin_provider.elf`.
- Connect the TE0703 baseboard to the designer's computer using a USB and an Ethernet cable.
- Start a *Trivial File Transfer Protocol* (TFTP) server on the host computer for providing the `bin_provider.elf` and `bitstreams` files to the hardware.
- Prepare the boot loader U-Boot installed in the GigaZee QSPI flash[5] to boot the software `bin_provider` from QSPI and transfer `bin_provider.elf` and `bitstreams` over the network connection via TFTP by entering the commands listed in Appendix B.2.4 on the USB serial console.
- Run the commands `env run update_qspi` and `env run update_mmc` in U-Boot to program the on-board QSPI and eMMC memories with `bin_provider.elf` and `bitstreams`, respectively, then reset the board (log in Appendix B.2.5).
- Program the FPGA with the first full bitstream using the Vivado IDE.

For comparing bitstream size savings achieved by different generation approaches, partial bitstreams were produced using the four methods introduced in Section 4.7: Module-based, module-based with Xilinx compression, difference-based, and torCombitgen. The Vivado Tcl commands that were used to generate the bitstreams can be found in Appendix B.2.2.

5.2.2 Results

The waveforms captured when running this test are printed in Figure 5.3 with an explanation of the signals in Table 5.1. The samples show that the ViSARD core outputs the correct exponentiation and square root result alternately in `visard_mdata`. This is the expected behavior and proves that the PRC exchanges the execution unit inside the ViSARD – otherwise it would be impossible for the correct result to appear on the output.

[5] The GigaZee ships with a pre-installed demo design that includes a fully configured first stage boot loader and U-Boot that, in principle, may be used for this purpose. It was observed that this programs the FPGA clock to around 50 MHz, which is half of the 100 MHz the designs are intended to use. A custom boot loader and U-Boot



ILA Status: Idle

Name	Value
global_cycle[31:0]	1963821210
visard_mcmd	0
visard_busy	0
visard_mdata[63:0]	402800000000
visard_mdata_en	0
prc_req	1
prc_bitstream_index	0
prc_ready	1
prc_done	0
icap_data[31:0]	00000000
icap_enable	0

(a) Cursor at partial reconfiguration request

ILA Status: Idle

Name	Value
global_cycle[31:0]	1963878090
visard_mcmd	0
visard_busy	0
visard_mdata[63:0]	402800000000
visard_mdata_en	0
prc_req	0
prc_bitstream_index	0
prc_ready	1
prc_done	1
icap_data[31:0]	04000000
icap_enable	1

(b) Cursor at partial reconfiguration finish

Figure 5.3: Recorded waveforms showing basic partial reconfiguration of the ViSARD working in practice. All numbers captured on buses are shown in hexadecimal notation except for `global_cycle`. The logic analyzer core was set to trigger on `visard_mcmd` high (indicated by red vertical lines). Other ILA settings: 4 windows 256 samples deep each, trigger position in window at 10. Description of signals in Table 5.1.

Table 5.1: Drivers and descriptions of signals captured for demonstrating basic partial reconfiguration functionality

Signal name	Driving component	Description
`global_cycle`	Top level	Global cycle counter
`visard_mcmd`	Execution control	Enable ViSARD
`visard_busy`	ViSARD	Last instruction was executed
`visard_mdata`	ViSARD	Program output data
`visard_mdata_en`	ViSARD	`visard_mdata` valid
`prc_req`	Execution control	Request partial reconfiguration
`prc_bitstream_index`	Execution control	Index of bitstream to load
`prc_ready`	PRC	PRC ready for requests
`prc_done`	PRC	Partial reconfiguration finished
`icap_data`	PRC	ICAP bitstream data word
`icap_enable`	PRC	ICAP clock enable

Speed and timing

The time the reconfiguration takes can be measured by looking at the value of the global cycle counter as previously described. The counter value when reconfiguration is requested is 1,963,821,210 (Figure 5.3a). It becomes 1,963,878,090 when reconfiguration finishes (Figure 5.3b).

build using the Xilinx SDK was performed to correct this and programmed to the QSPI flash. Important points are to use the Trenz Electronic patches for setting the device Ethernet address and to modify the boot loader to always activate the PL level shifters even when no PL design is included in the boot configuration.

The difference between these values is 56,880 cycles. At 100 MHz, one clock cycle equals 10 ns. Therefore, the reconfiguration took a total of 56,880 · 10 ns = 568.80 µs.

The partial bitstreams used had a size of 227,336 bytes each, leading to a throughput average of 227,336 bytes/568.80 µs = 399.677 MB/s in this specific measurement. This is 99.92 % of the maximum specified ICAP throughput of 400 MB/s, so the goal of having very fast reconfiguration is definitely reached.

Data recorded by the `statistics_collector` (Figure 5.4) indicates that 56,880 cycles is the most common time required for reconfiguration, with the minimum being 56,879 cycles and the maximum 56,911 cycles. The mean value is 56,880.7 cycles with a standard deviation of 4.9 cycles. Correspondingly, on average $(568.807\ \mu\text{s})^{-1} = 1758.07$ reconfigurations can be performed per second and the long-term average throughput is 227,336 bytes/568.807 µs = 399.672 MB/s. As the latency of the PRC is completely deterministic and static, the variance is the result of the Zynq PS memory controller having different access latencies depending on when the request is made. This verifies that the PRC is able to deliver hard real-time reconfiguration in practice with the limitation of assuming a bitstream provider that can satisfy this constraint as well.

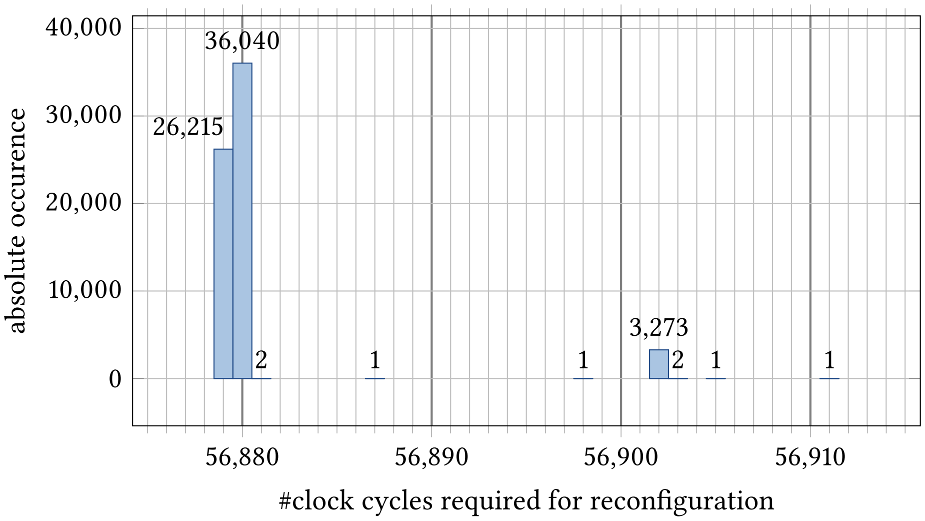


Figure 5.4: Histogram of clock cycle count required for one exchange of EUs via partial reconfiguration in the basic test scenario measured using 65,536 subsequent samples. Cycle count is the number of cycles elapsed between assertion of `prc_req` and successive assertion of `prc_done`.

Resource utilization

The resources used by the design and its components were extracted from the Vivado IDE and listed in Table 5.2. The PRC needs about 250 LUTs plus one BRAM tile for the FIFO, so it is small considering that the target device contains 53,200 LUTs. The resource usage of the execution units mostly agrees with the data previously presented in Table 3.2, with the slight increase in LUT usage most likely attributable to additional LUTs inserted in the RP as route endpoints for the unused partition pins allocated to the second input operand (both NatExp and Sqrt use only one input operand).

Table 5.2: Primary resources used in the basic test design, by component. All data was extracted from Vivado utilization reports and the ILA and debug hub subtracted from full designs sizes. Effective flip-flop numbers are calculated as explained in Section 3.2.6. No additional debugging or measurement components were present. The equivalent non-PR design implements the same functionality (NatExp and Sqrt), but without a reconfigurable partition. It uses identical IP cores and settings, ViSARD test program, and execution control unit, but the PRC is replaced by a dummy that pretends that reconfiguration finishes instantly without doing anything.

	Component	LUTs	FFs eff.	BRAMs	DSPs	(FFs)
(a)	PRC	251	502	1	0	(294)
(b)	NatExp RM	2,096	4,192	0	15	(1,112)
(c)	Sqrt RM	1,763	3,526	0	0	(1,654)
(d)	PR design (complete with NatExp RM)	3,326	6,652	4	18	(2,311)
(e)	PR design (static part): *(d)* - *(b)*	1,230	2,460	4	3	(1,199)
(f)	Pblock for PR	2,800	5,600	0	20	(5,600)
(g)	PR design (effective): *(e)* + *(f)*	4,030	8,060	4	23	(6,799)
(h)	Equivalent non-PR design	4,745	9,490	3	18	(3,669)
(i)	**PR vs. non-PR: *(g)* - *(h)***	**-715**	**-1,430**	**+1**	**+5**	(+3,130)

When comparing the resource usage of this partial reconfiguration design with an equivalent non-PR one, it is not immediately obvious what numbers should be examined. The Xilinx tools can of course report the usages of the complete designs, but additional considerations have to be taken into account: The PR design has resources in the Pblock(s) designated for reconfiguration that are blocked and not usable for implementing additional logic, but do not appear in those numbers. Therefore, the approach taken in this thesis is to take the static part of the PR design and add the resources allocated for reconfiguration to it. Because of that, PR will seem to increase raw FF usage for strongly LUT-bounded designs (and vice versa) even though the flip-flops would not be usable anyway. Additionally, both the PR and non-PR raw FF numbers underrepresent the real demand by ignoring the structure of 7 series FPGA logic slices which effectively prohibits using most flip-flops in other components of a design (as explained in Section 3.2.6). Comparisons will therefore be based on the effective FF usage calculated as twice the LUT usage. The raw FF usage numbers are included for reference.

Applying this methodology to the results at hand, partial reconfiguration in this basic test requires about 700 LUTs and 1,400 FFs less, while a moderate increase in BRAM demand (caused by the PRC) and in DSP usage (caused by surplus resources in the Pblock) is also identifiable. All in all, the savings in logic resources amount to approximately 2 CLB columns and are therefore significant even though this test was only designed to prove the general principle. It is expected that for larger designs, the resource footprint of the PRC (which is always the same irrespective of the design size) as well as the LC waste caused by reconfigurable regions (which is more pronounced in small Pblocks) will diminish in comparison.

Bitstream size

All partial bitstreams described in the test setup were generated and verified to work in practice. Regarding their sizes, Table 5.3 shows that the Xilinx-flavored compression achieves around 15 % reduction in the present example, while the difference-based methods cut the size in half. This seemed like a commendable result at first, but closer inspection of the generated bitstreams revealed the reason why the resulting bitstreams were so close to 50 % of the original size: The default and compressed bitstreams generated by Vivado write every configuration frame twice with different contents, while the other methods skip this and only write the final value.

Table 5.3: Basic test bitstream sizes generated by different methods

	Bitstream size [bytes]			
Method	**NatExp**		**Sqrt**	
Default	227,336	**(100.00 %)**	227,336	**(100.00 %)**
Xilinx compression	195,424	(85.96 %)	190,660	(83.87 %)
Default without blanking	113,772	**(100.00 %)**	113,772	**(100.00 %)**
Difference	113,748	(99.98 %)	113,748	(99.98 %)
torCombitgen	113,732	(99.96 %)	113,732	(99.96 %)

The reason for this behavior was found in [Xil18m, p. 123]: Vivado starting with version 2016.1 inserts special blanking events into all partial bitstreams for 7 series devices after Xilinx had realized that without these, reconfiguration might cause glitches in the static logic under rare circumstances. Although the user guide claims that this procedure “results in an increased size of the partial bit files” and that “compression can be used to reduce these effects”, the numbers obtained in this test make it clear that the bitstream gets inflated to double the size and that compression is able to compensate only for a small portion of this increase. That Vivado does not insert these blanking events when using the `-reference_bitfile` option is most likely due to the fact Xilinx does not usually advertise this option for dynamic partial reconfiguration.

With this knowledge, the blanking events were removed from the default-generated bitstream in order to have a comparable base for assessing the methods. The result is that the size of the bitstream generated with the difference-based Vivado method differs insignificantly (by 0.02 %) from that baseline. This is virtually the same for the torCombitgen output – due to its principle of operation, torCombitgen cannot produce any bitstream that is smaller than the differences between two participating top-level designs.

The Vivado-generated bitstreams include additional commands for shutting down and restarting the device, which were removed in order to make them suitable for DPR. That these were present at all is a strong indicator that Xilinx intends this method to be used with static reconfiguration.

The slightly smaller size yielded by the difference-based methods is explained by some writes to internal registers of the configuration controller that are missing compared to the default partial bitstreams. They are partly not documented at all and partly are supposed to mask memory cell contents for readback according to the documentation [Xil18c]. Configuration readback is not utilized here, so the purpose of this commands remain unclear and partial reconfiguration still worked correctly without them.

At least in this example, the RMs were dissimilar enough so that no configuration frames between them were identical and could be skipped during reconfiguration. Whether the non-blanking bitstreams can be used in practice at all is questionable considering the possibility of rare glitches, but this does by itself not completely invalidate the difference-based approach: Other devices such as UltraScale+ FPGAs have differing blanking strategies in any case and other designs might have equivalent frames that can be omitted.

5.3 Two cores with parallel calculation/reconfiguration

The next test is quite similar to the preceding section in structure and execution, but it adds another ViSARD core in order to test calculation and reconfiguration in parallel.

5.3.1 Test setup

The design from Section 5.2 is taken as basis and extended by instantiating a second, identical ViSARD core and connecting it to a tweaked execution control unit. The new behavior is to first program the first EU (NatExp) into the first core, followed by the second core, and after that do the same for the second EU (Sqrt). After reconfiguration of a core is finished, it is immediately started and the reconfiguration of the other respective core is initiated. This process is repeated ad infinitum. Performing partial reconfiguration in this way enables a more efficient use of the FPGA configuration controller in the time domain since it is possible to run calculation and reconfiguration in parallel on different cores. With a single core, the current implementation does not allow DPR to be performed as long as the core is running a program.

The goals of this test are to verify

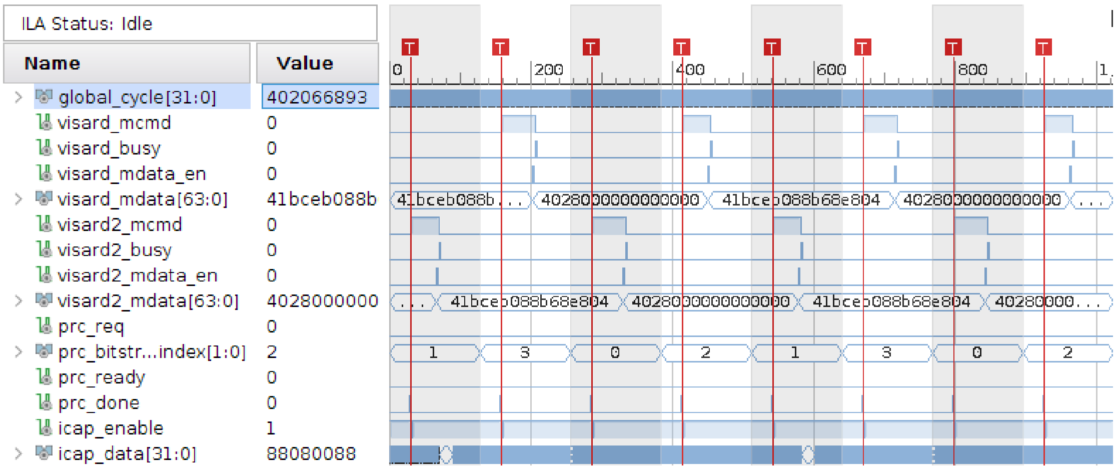
- that parallel calculation/reconfiguration is feasible, practical, and more time-efficient,

- that the PRC is not only capable of performing one reconfiguration at very high speed, but can also sustain this throughput over longer periods of time without pauses, and
- that the PRC is able to overlay an ongoing reconfiguration with the buffering of the next partial bitstream, bringing the latency between successive reconfiguration cycles down to almost zero.

The bitstreams were ordered in the following form in the packaged file: First both partial bitstreams for the first core (i.e. NatExp at position 0 and Sqrt at position 1), then both for the second core (i.e. NatExp at position 2 and Sqrt at position 3).

5.3.2 Results

As before, the first step of verification was to check whether the ViSARD cores produce the desired outputs, which are identical to the basic test. The logic analyzer waveform captured shows that this is the case (Figure 5.5). Both cores are running alternately and each one is outputting the correct NatExp or Sqrt result. The bitstream index loaded is progressing correctly from 0 (NatExp on 1st core) to 2 (NatExp on 2nd core) to 1 (Sqrt on 1st core) and finally to 3 (Sqrt on 2nd core). Reconfiguration is performed in parallel to one ViSARD core running, as indicated by `icap_enable` being asserted at the same time `visard_mcmd` is set for one of the cores.

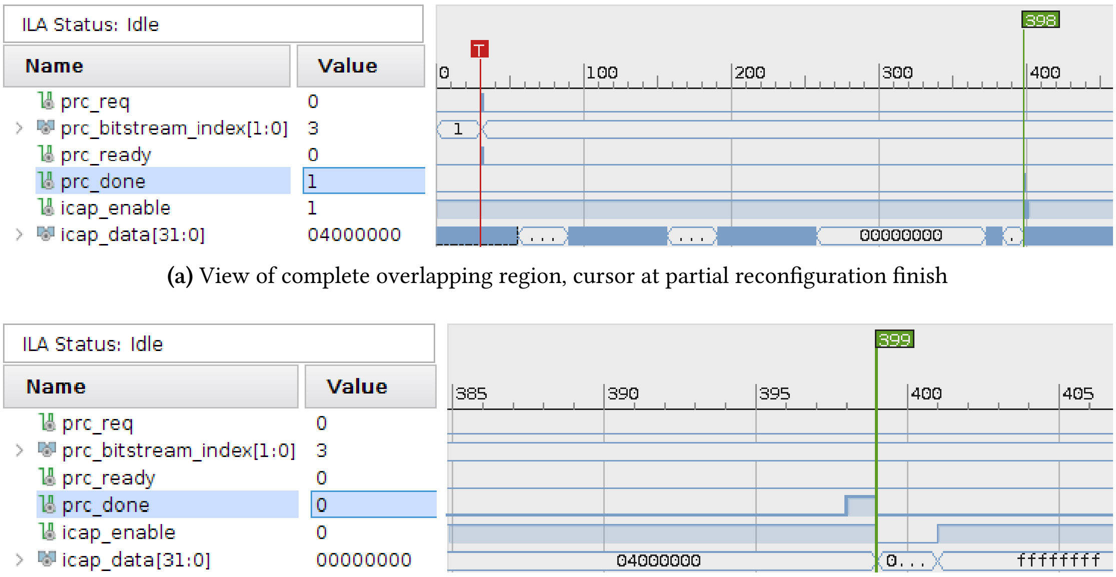


ILA Status: Idle

Name	Value
global_cycle[31:0]	402066893
visard_mcmd	0
visard_busy	0
visard_mdata_en	0
visard_mdata[63:0]	41bceb088b
visard2_mcmd	0
visard2_busy	0
visard2_mdata_en	0
visard2_mdata[63:0]	4028000000
prc_req	0
prc_bitstr...index[1:0]	2
prc_ready	0
prc_done	0
icap_enable	1
icap_data[31:0]	88080088

Figure 5.5: Recorded waveform showing parallel reconfiguration and calculation of two ViSARD cores working in practice. All numbers captured on buses are shown in hexadecimal notation except for `global_cycle`. The logic analyzer core was set to trigger on `visard_mcmd` or `visard2_mcmd` high (indicated by red vertical lines). Other ILA settings: 8 windows 128 samples deep each, trigger position in window at 30. Description of signals in Table 5.1 (prefix `visard_` indicates the first, prefix `visard2_` the second ViSARD core).

Closer inspection of one reconfiguration process reveals that the PRC is ready to preload the next bitstream 368 cycles before all data was sent to the ICAP (cf. Figure 5.6a: `prc_ready` is set in cycle 30, `prc_done` finally in cycle 398). The following reconfiguration request is issued immediately when the PRC is ready. This in most cases allows enough time to fill the FIFO inside the controller with enough data so that reconfiguration can continue almost uninterruptedly to the next bitstream when one bitstream ends. Figure 5.6b shows that the ICAP is idle for 2 cycles between reconfigurations. This exactly matches the considerations previously outlined in Section 4.3.3 when implementing the PRC.


ILA Status: Idle

Name	Value
prc_req	0
prc_bitstream_index[1:0]	3
prc_ready	0
prc_done	1
icap_enable	1
icap_data[31:0]	04000000

(a) View of complete overlapping region, cursor at partial reconfiguration finish

ILA Status: Idle

Name	Value
prc_req	0
prc_bitstream_index[1:0]	3
prc_ready	0
prc_done	0
icap_enable	0
icap_data[31:0]	00000000

(b) Zoom on `icap_enable` transition

Figure 5.6: Recorded waveforms showing signal transitions during overlapping bitstream preloading and PR. All numbers captured on buses are shown in hexadecimal notation. The logic analyzer core was set to trigger on `prc_req` high (indicated by red vertical line). Other ILA settings: 1 window 1024 samples deep, trigger position in window at 30.

Statistical observation of reconfiguration times reveals that as before, the Zynq DDR memory controller sometimes needs a little bit longer to supply the requested data. The average reconfiguration time is noticeably less than without overlapping bitstream preloading, so the goal of speeding up the process was achieved. Using the data shown in Figure 5.7, the mean number of clock cycles required for PR is calculated to be 56,836.7 with a standard deviation of 4.2, which makes the process 0.1 % faster than in the previous test.

The average ICAP throughput derived from this (using again 227,336 bytes for the bitstream size and a clock period of 10 ns) is 227,336 bytes/568.367 µs = 399.981 MB/s, which is 99.995 % of the 400 MB/s maximum (as opposed to the 99.92 % previously achieved). Further sophistication of the PRC may bring this to 100 %, but the difference is considered to be too marginal to justify complicating the architecture and code for that gain. Finally, both the minimum of 56,836 cycles and the maximum of 56,862 cycles measured is below the minimum value of 56,879 cycles obtained without bitstream preloading.

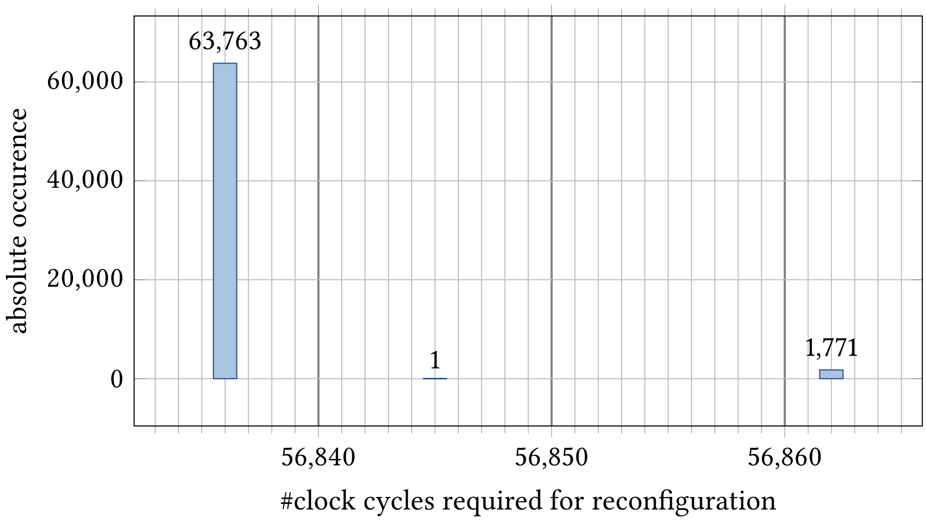


Figure 5.7: Histogram of clock cycle count required for one exchange of EUs via partial reconfiguration in the multi-core test scenario measured using 65,536 subsequent samples. Cycle count is the number of cycles elapsed between successive assertions of `prc_done`. The first sample was discarded as it represents a transitory set-up phase before bitstream loading and reconfiguration can overlap.

5.4 Multiple execution units

The last test performed in this thesis investigates the capabilities of DPR to cover more than two EU configurations and configurations that contain a differing number of EUs.

5.4.1 Test setup

This test is similar in structure to the basic test described in Section 5.2. It uses one ViSARD core that is enhanced for partial reconfiguration of its EUs, the custom PRC, and a similar execution control unit on the top level. `execution_control` was modified to cycle through 3 instead of 2 partial bitstreams to load and programs to run one after another. An ILA debug core is included for capturing the test results.

The 3 EU combinations chosen for this test due to their resource footprint compatibility are[6]:

1. Divide (latency: 30)
2. Sqrt (latency: 28) and SinCos (mode of operation: serial, precision: 48 bits, latency: 57)
3. Sqrt (latency: 28), NatExp (BRAM usage: full, DSP usage: medium, latency: 20), and Multiply (DSP usage: full, latency: 15)

[6] Since the ViSARD does not yet have a very diverse range of specialized EUs fit for DPR, the Multiply EU was included in the last configuration although it is rather small and only that configuration uses DSP and BRAM resources.

Besides showing that RMs can contain more than one and even a differing number of EUs, producing 3 top-level bitstreams will allow this test to compare the difference-based partial bitstream generation method in Vivado with torCombitgen and observe if any significant savings can be achieved. Moreover, having 3 EUs in one RM makes it possible to gauge the influence of partition pins on the resource usage.

In this test, it was attempted to constrain the Pblocks further in order to push the limits of partial reconfiguration. The result is highlighted in yellow in Figure 5.8. For the base rectangle, a region on the device was selected that includes both a BRAM and a DSP column plus a sufficient amount of logic slices so the resource requirements can be fulfilled (overview in Table 5.4).

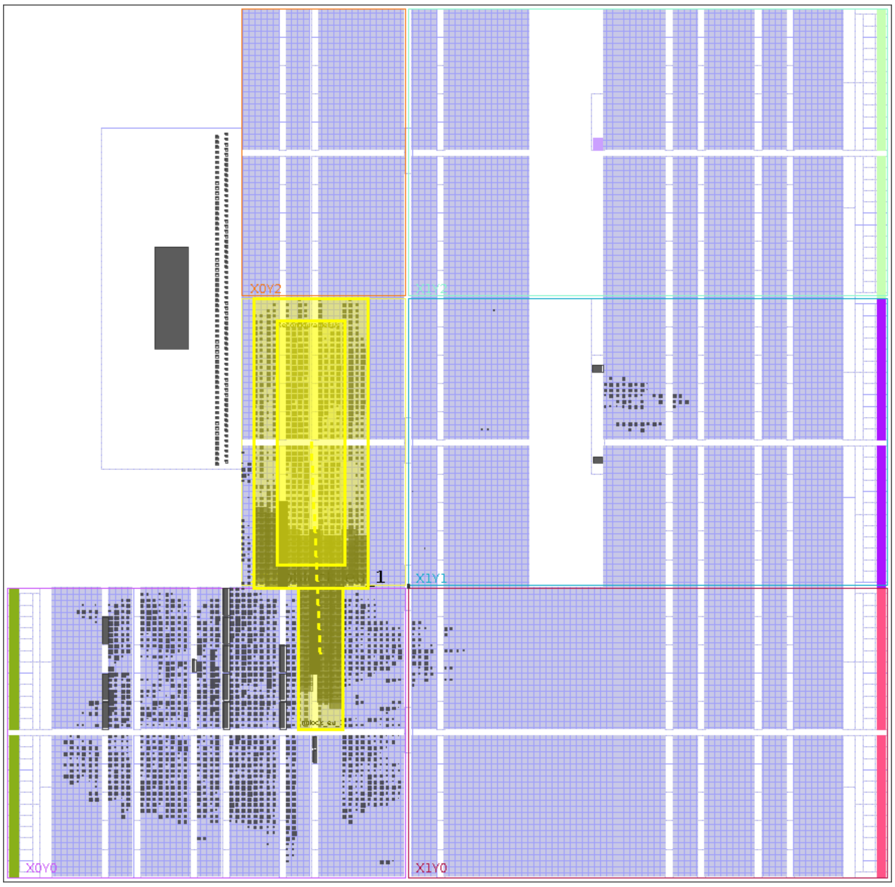


Figure 5.8: Device overview highlighting the Pblock (outlined in yellow) assigned to the reconfigurable partition of the execution units exchangeable in the multi-EU test as seen in the implemented full Divide design. White vertical columns running through the device contain BRAM or DSP resources (not visually distinguished by the Vivado IDE which generated the image). Resource blocks that appear in dark gray are occupied by the design. Inside the Pblock, larger gray rectangles additionally mark where interconnect resources are reserved for partition pins.

Actually, only a quarter of the BRAM resources in the column is required for the reconfigurable modules, but practical experimentation has revealed that even if the surplus memory is excluded from the Pblock, the Xilinx placer will be unable to use it in the static logic. This

Table 5.4: Combined resource usage of the EU configurations chosen for the multi-EU test. Usage numbers taken from Table 3.2.

EU	LUTs	FFs	BRAMs	DSPs
Divide	3,196	3,018	0.0	0
Sum configuration 1	**3,196**	**3,018**	**0.0**	**0**
Sqrt	1,713	1,654	0.0	0
SinCos	1,359	795	0.0	0
Sum configuration 2	**3,072**	**2,449**	**0.0**	**0**
Sqrt	1,713	1,654	0.0	0
NatExp	1,122	1,029	2.5	15
Multiply	220	498	0.0	10
Sum configuration 3	**3,055**	**3,181**	**2.5**	**25**

is to be expected given that configuration frames cover the whole BRAM column, so any contents written there would get overwritten when PR is performed. As a consequence, the whole column was kept part of the Pblock so that the resource cost and savings of PR are assessed realistically.

An extra rectangle of DSPs and logic slices below the main region was added in order to fill up to the respective number of resource units required. It occupies half a clock region in height as an experiment. As previously explained, on the one hand the partial bitstream will still have the same size as when the full clock region had been included. On the other hand, since not all of the DSPs can be used in the RP anyway, leaving the surplus out of the Pblock might make it possible for the placer and router to make use of them. In principle, the same is true for the surplus of logic slices.

As Figure 5.8 shows, DSPs of the static design are placed into the top of the DSP column half below the Pblock rectangle. The logic slices stay conspicuously unoccupied, but when placing the Pblock at various other locations in the device it was observed that sometimes static logic is put into columns that contain reconfigurable logic. Therefore, it seems that additional constraints apply or that the placer is at least quite reluctant to mix static and dynamic logic in one column.

Moving on to the ViSARD setup, the first test program to run on the processor is:

```
dq testin1 ? 50
dq testin2 ? 40
dq testout ? 0

div testin1 testin2 testout

out testout ? 0
```

It calculates $\frac{50}{40}$, so the output should be 1.25. As double-precision IEEE-754 floating point number in hexadecimal notation, this is `3ff4000000000000`.

The second test program is:

```
dq testin ? 0.25
dq testout ? 0

sqrt testin ? testout
sin testout ? testout

out testout ? 0
```

It calculates $\sin\sqrt{0.25}$, so the output should be 0.479425538604203. As double-precision IEEE-754 floating point number in hexadecimal notation, this is `3fdeaee8744b05f0`.

The third test program is:

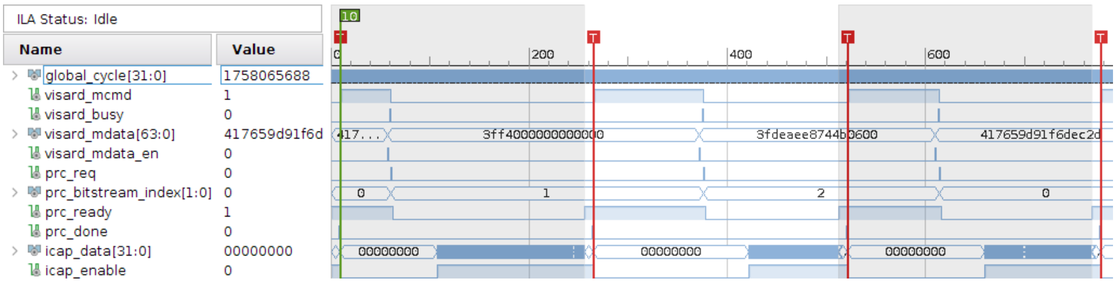
```
dq testin ? 144
dq testout ? 0

sqrt testin ? testout
natexp testout ? testout
mul testout testin testout

out testout ? 0
```

It calculates $144e^{\sqrt{144}}$, so the output should be 23436689.96433656459635. As double-precision IEEE-754 floating point number in hexadecimal notation, this is `417659d91f6dec2d`.

The compilation of the test programs, the concatenation of their resulting memories, and the generation of the bitstreams and packed memory files were all performed analogously to the basic test (see Appendix B.3).

5.4.2 Results

Running the test on the FPGA revealed that the ViSARD produces the exact result expected for the Divide and Sqrt-NatExp-Multiply configurations (Figure 5.9). When the Sqrt-SinCos RM calculation is finished, the result is `3fdeaee8744b0600` (0.4794255386042039) instead of the correct `3fdeaee8744b05f0` (0.479425538604203). The absolute deviation is $9 \cdot 10^{-16}$. As the SinCos EU implementation is limited to 48 bits of precision in calculation (as previously explained in Section 4.2), this is within the expected margin of error and does not indicate a problem introduced by partial reconfiguration. The non-PR design implemented for resource comparison purposes produces the same result.



ILA Status: Idle

Name	Value
global_cycle[31:0]	1758065688
visard_mcmd	1
visard_busy	0
visard_mdata[63:0]	417659d91f6d
visard_mdata_en	0
prc_req	0
prc_bitstream_index[1:0]	0
prc_ready	1
prc_done	0
icap_data[31:0]	00000000
icap_enable	0

Figure 5.9: Recorded waveform showing multi-EU partial reconfiguration of the ViSARD working in practice. All numbers captured on buses are shown in hexadecimal notation except for `global_cycle`. The logic analyzer core was set to trigger on `visard_mcmd` high (indicated by red vertical lines). Other ILA settings: 4 windows 256 samples deep each, trigger position in window at 10. Description of signals in Table 5.1.

5.4.3 Speed and timing

The speed and timing of the reconfiguration was very similar to the prior tests (Figure 5.10). The average cycle count needed for one reconfiguration was 123,573.1 with a standard deviation of 6.6 cycles, allowing for $(1{,}235.731\ \mu\text{s})^{-1} = 809.24$ reconfigurations per second. The increased bitstream size of 494,104 bytes causes the initial buffering latency to carry less weight, so the long-term average throughput of 494,104 bytes/1,235.731 µs = 399.848 MB/s or 99.96 % of the 400 MB/s maximum is slightly higher than in the basic test (Section 5.2).

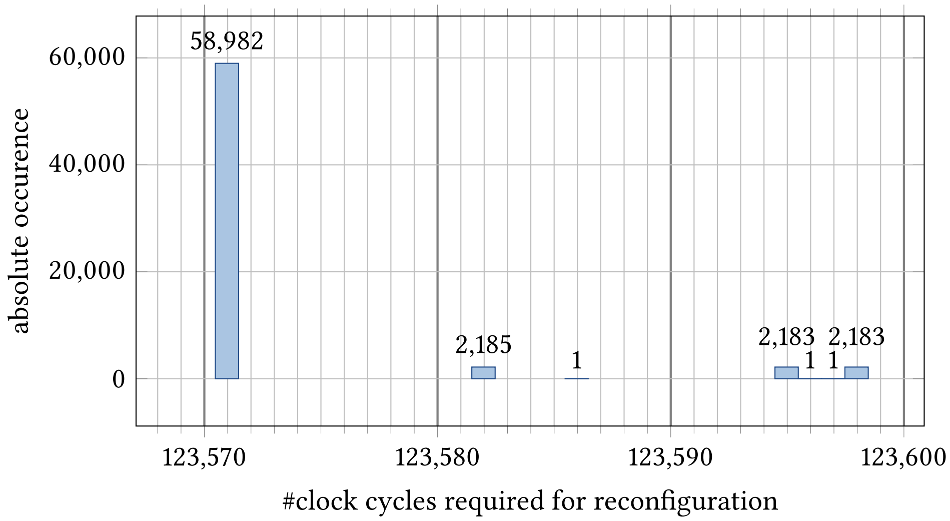


Figure 5.10: Histogram of clock cycle count required for one exchange of EUs via partial reconfiguration in the multi-EU test scenario measured using 65,536 subsequent samples. Cycle count is the number of cycles elapsed between assertion of `prc_req` and successive assertion of `prc_done`.

Resource utilization

As a larger part of the FPGA was reconfigured in this test case, the savings when compared to a completely static design become more pronounced. Table 5.5 lists the resource cost of each RM, which were again roughly equivalent to the sum of resources required for the EUs (Table 5.4) in the respective configuration. Compared to the equivalent non-PR design, around 3,500 LUTs and 7,000 FFs could be saved. This is a very good result that amounts to a relative saving of 40.7 % when directly comparing full PR and non-PR designs. Both BRAM and DSP resources suffered an increase mainly due to functional units that had to be allocated to the PR Pblock but could not be utilized by the dynamic logic. The PRC itself had roughly identical resource usage as in the previous tests in spite of the increase in the number of bitstream index bits (1 instead of 2 in order to support the loading of 3 different partial bitstreams).

Table 5.5: Primary resources used in the multi-EU test design, by component. All data was extracted from Vivado utilization reports and the ILA and debug hub subtracted from full designs sizes. Effective flip-flop numbers are calculated as explained in Section 3.2.6. No additional debugging or measurement components were present. The equivalent non-PR design implements the same functionality (all arithmetic EUs), but without a reconfigurable partition. It uses identical IP cores and settings, ViSARD test program, and execution control unit, but the PRC is replaced by a dummy that pretends that reconfiguration finishes instantly without doing anything.

	Component	LUTs	FFs eff.	BRAMs	DSPs	(FFs)
(a)	PRC	247	494	1.0	0	(292)
(b)	Divide RM	3,273	6,546	0.0	0	(3,018)
(c)	Sqrt-SinCos RM	3,063	6,126	0.0	0	(2,463)
(d)	Sqrt-NatExp-Multiply RM	3,055	6,110	2.5	25	(3,181)
(e)	PR design (complete with Divide RM)	4,575	9,150	4.0	3	(4,261)
(f)	PR design (static part): *(e)* - *(b)*	1,302	2,604	4.0	3	(1,243)
(g)	Pblock for PR	3,800	7,600	10.0	30	(7,600)
(h)	PR design (effective): *(f)* + *(g)*	5,102	10,204	14.0	33	(8,843)
(i)	Equivalent non-PR design	8,607	17,214	5.5	28	(7,945)
(j)	**PR vs. non-PR: *(h)* - *(i)***	**-3,505**	**-7,010**	**+8.5**	**+5**	(+898)

All in all, this result shows the upsides (significant logic slice savings by multiplexing in time and low overhead of the supporting logic such as the PRC) as well as the downsides (difficulty of finding suitable Pblocks and special functional units ending up dormant) of partial reconfiguration as far as FPGA resource utilization is concerned.

Bitstream size

All bitstream generation methods were executed and their output verified to work on the device. The sizes of the bitstreams (Table 5.6) roughly confirm the findings of Section 5.2.2: Xilinx compression allowed a reduction of roughly 10 % to 30 % depending on the module. Since the first two EU configurations (Divide and Sqrt-SinCos) do not make use of BRAM, all BRAM columns get overwritten with zeroes which compresses very well. This is not the case for the third configuration (Sqrt-NatExp-Multiply), so the compression is not as beneficial there.

Table 5.6: Multi-EU test bitstream sizes generated by different methods

	Bitstream size [bytes]					
Method	**Conf. 1 (1 EU)**		**Conf. 2 (2 EUs)**		**Conf. 3 (3 EUs)**	
Default	494,104	**(100.00 %)**	494,104	**(100.00 %)**	494,104	**(100.00 %)**
Xilinx compression	347,276	(70.28 %)	347,276	(70.28 %)	447,960	(90.66 %)
Default without blanking	247,156	**(100.00 %)**	247,156	**(100.00 %)**	247,156	**(100.00 %)**
Difference	247,132	(99.99 %)	181,748	(73.54 %)	247,132	(99.99 %)
torCombitgen	247,116	(99.99 %)	247,116	(99.99 %)	247,116	(99.99 %)

Difference-based methods were evaluated by comparing them against the baseline of a default partial reconfiguration bitstream with the blanking events removed (see the explanation in Section 5.2.2). For the Vivado difference method, partial bitstreams that conform to the switch pattern of the execution control unit were generated. This means that they switch respectively from configuration to configuration in ascending order (and wrapping back to the first one after the third). Consequently, the bitstream for configuration 2 can only be used when configuration 1 is currently loaded and will produce erroneous behavior when configuration 3 was loaded instead.

This explains why only configuration 2 achieves a 26 % saving in size and neither configuration 1 nor 3 is any different from the normal partial bitstream. Configuration 2 is the only case where a switch is performed from a RM not using any BRAM (Divide) to another one not using any (Sqrt-SinCos), so the BRAM columns are already set to zero and do not need to be reprogrammed. torCombitgen cannot reproduce this because it specifically wants to have bitstreams that can be loaded in any order, so the columns have to be written always. As the RMs seem to be considerably different in implementation in spite of Sqrt being used twice, torCombitgen is unable to otherwise trim any configuration frames from the bitstream.

Influence of partition pins

The last results produced as part of this test concern the influence of the partition pin count on PR performance as described in Section 3.2.3. All numbers previously presented were generated with the ALU result multiplexer completely contained in the static portion of the design because that is the more straightforward design. As potential optimization, the multiplexer for the result of the EUs contained in an RM was moved to the dynamic logic (i.e. inside the RM) to reduce the number of partition pins. The static logic still contains a multiplexer switching between EUs that are part of the static design and the result from the reconfigurable module.

Resource usage differences resulting from this change were negligible (Table 5.7). Switching to one output from the RM reduced the amount of LUTs used as tie-off when outputs were unused previously. When none were unused in the case of 3 EUs being present in the RM, LUTs usage increased due to the additional multiplexer. Similarly, removing part of the multiplexer from the static design decreased its LUT usage. At the same time, 64 more flip-flops were required for at present unknown reasons.

Table 5.7: Comparison of LUT and FF resources used in the multi-EU test design when the result multiplexer is completely in the static part of the ALU (design 1) and when it is partly in the RM (design 2). Design 1 is identical to the evaluation in Table 5.5. All data was extracted from Vivado utilization reports and the ILA and debug hub subtracted from full designs sizes. Effective flip-flops numbers are calculated as explained in Section 3.2.6.

Component	Design	LUTs	FFs eff.	(FFs)
Divide RM	1	3,273	6,546	(3,018)
Divide RM	2	3,200	6,400	(3,018)
Difference		**-73**	**-146**	**(±0)**
Sqrt-SinCos RM	1	3,063	6,126	(2,463)
Sqrt-SinCos RM	2	3,048	6,096	(2,458)
Difference		**-16**	**-32**	**(-5)**
Sqrt-NatExp-Multiply RM	1	3,055	6,110	(3,181)
Sqrt-NatExp-Multiply RM	2	3,119	6,238	(3,181)
Difference		**+64**	**+128**	**(±0)**
PR design (complete with Divide RM)	1	4,575	9,150	(4,261)
PR design (complete with Divide RM)	2	4,470	8,940	(4,325)
Difference		**-105**	**-210**	**(+64)**
PR design (static part)	1	1,302	2,604	(1,243)
PR design (static part)	2	1,270	2,540	(1,307)
Difference		**-32**	**-64**	**(+64)**

In any case, these results are as expected and show that the amount of partition pins does not in itself significantly influence resource usage in both the static and the dynamic portion of a DPR design for reasonably sized Pblocks. For the specific case of having a result multiplexer be part of the RM, without inserting additional registers (that would increase EU latency) it creates a combinatorial logic path from the static to the dynamic and back to the static logic. This is usually discouraged due to the impracticality of optimizing across RP boundaries. In this test, the design continued to function as expected without any timing issues. As at the same time no distinct positive impact could be registered, the added complexity of splitting the ALU result multiplexer cannot be recommended at present.

5.5 Summary

This chapter has covered the practical tests of the VHDL code and utilities implemented as part of this thesis as well as some further comparisons between bitstream generation methods and an analysis of the influence of partition pins. A summary of the main findings is given below.

First, a simulation of the custom partial reconfiguration controller showed that bitstreams are correctly loaded from the Zynq-7000 AXI_HP bus with satisfying performance (393.988 MB/s). Table 5.8 lists this throughput result as well as the values achieved in further tests for comparison. The result achieved in simulation was verified on hardware in the next test that combined the PRC and the ViSARD by exchanging one execution unit. This was successful and achieved even higher throughput. As a further optimization, enabling the PRC feature of preloading the next bitstream in the last phase of an ongoing reconfiguration yielded the best result of sustained 399.981 MB/s or 99.995 % of the 400 MB/s specified device maximum. The goal of this thesis of accomplishing very high reconfiguration throughput was therefore reached.

Table 5.8: ICAP data throughput of the custom PRC obtained in practical tests in absolute numbers and as percentage of the maximum of 400 MB/s. Preloading refers to the technique of allowing the PRC to request and buffer the next bitstream from storage while configuration data is still being sent to the ICAP (explained in Section 4.3.3).

Section	Subject	Preloading	Avg. throughput [MB/s]	(% of maximum)
5.1	Simulation	No	393.988	(98.497 %)
5.2	Hardware	No	399.672	(99.918 %)
5.3	Hardware	Yes	399.981	(99.995 %)

As part of this tests, the resources required for the PRC were determined to be 251 LUTs, 294 FFs, and 1 BRAM tile when allowing to switch between two bitstreams (Table 5.2). In all categories, it occupies less than 1 % of the resources available on the device, so it is small and does not constitute considerable overhead. Of the PRCs presented for comparison in Section 3.1.1, usage figures on 7 series devices[7] are available for the RT-ICAP, the Xilinx PRC, and the ZyCAP. They are compared in Table 5.9. As the sources did not consider whether more flip-flops would be blocked by their PRCs than represented by the raw FF numbers (as was otherwise done in this thesis; see Section 3.2.6), the listing is unadjusted for all controllers in order to allow for a fair comparison. The RT-ICAP is competitive in terms of resource usage, but it was conceived for a different purpose and does not have an AXI memory interface. Vastly more logic resources are needed by the ZyCAP (more than twice) and the Xilinx PRC (more than three times). These results show that the Xilinx PRC would not have been suitable for working together with the ViSARD even when its timing characteristics were clear due to its high resource usage.

The multi-EU test in Section 5.4 has further revealed that having Pblocks that do not have simple rectangular shapes and are not aligned in height with configuration frames is challenging for the placer and router. If not all BRAM resources in a column are included in a reconfigurable Pblock, the excluded part is not usable in both the static and the dynamic design part, so this is typically not beneficial. The same is not true for DSP resources, since the practical results show that extra DSP elements in a column can be used in the static design. The results

[7] Comparing logic usage in terms of LUT utilization across device families makes very limited sense since the structure of their logic cells may be very different (e.g. LUTs may have fewer inputs), resulting in significantly different numbers even for the same design.

Table 5.9: Comparison of resource utilization and configuration throughput of the PRC presented in this thesis and other published PRC designs. All numbers were measured on Kintex-7-based FPGAs. Data for the custom PRC is from Table 5.2. Data for the RT-ICAP and the ZyCAP is taken from their respective publications. Data for the Xilinx PRC was measured in a test design (settings were chosen to be most similar to the custom PRC: no optional features (including AXI control interface), minimum FIFO depth of 32 implemented as BRAM, 4 clock domain crossing stages, 1 virtual socket with 2 modules with 1 hardware trigger each).

Controller	LUTs	FFs	BRAMs	Throughput [MB/s]
Present thesis	**251**	**294**	**1.0**	399.981
RT-ICAP [PSS17]	245	101	0.0	382.2
Xilinx PRC [Xil18f]	897	954	0.5	*n/a*
ZyCAP [VF14]	620	806	0.0	382

were inconclusive for logic (LUTs and FFs): In the test design, no static logic elements were placed into the excluded half of the logic slice columns at all. Yet, further experimentation revealed that this is not always the case and sometimes the extra resources will be used. The conditions for that to happen remain unclear. In summary, Pblock rectangles that are not vertically aligned to clock regions do not necessarily provide any resource benefit. It is best to avoid them and look for other placement sites if possible.

Compared to an equivalent non-PR design that performs the same functionality, on the one hand a reduction of 3,505 LUTs and 7,010 effective FFs was achieved by introducing dynamic partial reconfiguration. In relation to the full design sizes, the relative saving is 40.7 %. On the other hand, an increased requirement of BRAM (8.5 additional tiles) and DSP (5 additional tiles) resources resulted from these functional units having to be included in the PR Pblock despite not being used by any RM. Realistically, these dormant resources cannot be completely avoided when dealing with diverse EUs and are considered to be the cost of partial reconfiguration.

For speeding up reconfiguration by selecting the smallest bitstream possible, Xilinx compression (Section 4.7.2) seems to be very beneficial. It achieved savings of 10 to 30 % in bitstream size in the tests and has no downside. The difference-based methods were not able to identify identical configuration frames for logic and interconnect columns inside the reconfigurable partition that could be omitted. The simple difference of two bitstreams allows to save BRAM data in some cases, but is impractical if the order (in time) the reconfigurable modules are needed in is dynamic or otherwise unknown when the hardware is designed. For the ViSARD, this will not be an issue since it is conceived for real-time tasks that typically have static execution order.

The torCombitgen tool has the advantage that the generated bitstreams can be loaded in any order, but as it likewise could not find any frames to omit, the result is effectively equivalent to that of the module-based generation. It has the additional downside (compared to module-based generation) that when multiple RPs are present, operating on full top-level design files oblivious of any partitions makes it difficult to single out the configuration frames for a single RP if not all of them should be reprogrammed at the same time. Further tests on a more varied range of designs and devices are necessary to confirm whether no significant savings can be achieved in all cases or whether there are cases in which torCombitgen will have a positive effect. A possible scenario in that regard could be when unused BRAM columns are included in a larger Pblock because no alternative placement (with no BRAM in-between logic slices) could be found. For the time being, Xilinx compression remains the recommended option for generating small bitstreams.

As final consideration, the influence of the number of partition pins on the RP boundary and more specifically their reduction by moving part of the ALU result multiplexer into the RMs was evaluated. The result was that there is no significant difference in terms of resource usage and that for the special case of the EU calculation result where it is difficult to insert additional register stages, leaving the full multiplexer in the static design is recommended due to timing concerns.

Chapter 6

Conclusion

This thesis presented a complete solution for partial reconfiguration of a ViSARD soft-core-based processing unit on an FPGA on the execution unit level. The system consists of a custom partial reconfiguration controller implemented in VHDL connected via the industry-standard AXI bus to a partial bitstream provider, one or more ViSARD cores (enhanced to support multiple programs and EUs in reconfigurable partitions), and an execution control entity that decides what program to run and which bitstream to load. This final chapter summarizes the main findings and contributions of this research and gives a number of suggestions for further investigation.

6.1 Summary

Firstly, fundamental technologies and principles were introduced in Chapter 2. This started with the operating principle of FPGAs and a short description of important device components. Furthermore, key concepts of digital hardware design were explained and the hardware platform (the Trenz Electronic GigaZee SoM with a Xilinx Zynq-7000 SoC) available for practical tests was presented. After describing the idea of partial reconfiguration, its limitations, and special considerations on the target hardware, the description of the ViSARD and its EUs concluded this chapter.

Chapter 3 explained what components are required for any system introducing PR capabilities to the ViSARD. The essential component delivering the partial bitstreams to the FPGA configuration processor is the PRC (Section 3.1). Options available to achieve its functionality were discussed at length including many PRC designs presented in published literature and solutions offered by Xilinx itself such as the Xilinx PRC IP core. After analyzing all of them, the decision was made that a custom implementation would be the best fit for the particular use case of this thesis.

The architecture was conceived with the test hardware platform in mind, but it is designed to be easily adaptable to other 7 series devices or other FPGA families. To cleanly separate the platform-agnostic and platform-specific parts, the custom PRC (platform-agnostic within an FPGA family) loads partial bitstreams from an AXI bus (connected to a platform-specific endpoint). On the test hardware, this endpoint is an AXI_HP bus of the Zynq-7000 processing system in order to allow rapid access to the DDR3 SDRAM. The `bin_provider` utility described in Section 4.5 was implemented to run on the Zynq APU and copy bitstream data from the on-board eMMC to RAM once at system start-up.

Integration of PR capabilities into the ViSARD as explained in Section 3.2 was achieved by replacing direct floating-point IP core instantiations in the ALU with wrapper entities that can be marked as reconfigurable partition and adding functionality to run multiple different programs within the same program and data memories. Moreover, considerations on the location and shape of Pblocks as well as on the influence of RP partition pins were presented and practical recommendations were derived from them. In order to give designers looking to use a PR-enabled ViSARD core the data they need to make an informed decision about which EUs to reconfigure, all currently available units were analyzed with regard to their resource usage and suitability for PR in Section 3.2.5.

In Chapter 4, important implementational details of the custom PRC (including its user interface), the ViSARD integration, and the accompanying utility programs were presented. Additionally, a planned but not yet realized sine/cosine calculation execution unit was added to the ViSARD in order to have more EUs available for potential reconfiguration. Programs for usage by the designer developed in this thesis are the `bin_packer` utility (Section 4.4) which combines multiple partial bitstreams into one indexed file for easy upload and consumption by the PRC and `mem_concatenator` (Section 4.6.2) which concatenates several ViSARD data or program memory files into one file for having multiple ViSARD programs be part of the same memory block. Both utilities are implemented in the Python scripting language.

Module-based and difference-based generation are two fundamental approaches to producing partial bitstreams for delivery to the FPGA. They were introduced in Section 4.7 together with the compression method integrated into the Xilinx tools and torCombitgen, which represents a new method that stands in-between pure module-based and pure difference-based generation, but had not previously received testing.

All design and implementation concerns were put to the test in Chapter 5. At first, a simulation of the custom PRC confirmed that bitstreams are correctly loaded over the AXI bus and the process works in principle. After that, several hardware tests integrating the PRC and ViSARD in one system as intended verified that the reconfiguration works in practice and delivers sizable benefits. For an advanced configuration that includes five EUs in three configurations, about

3,500 LUTs in absolute numbers or about 40 % compared to an equivalent non-PR design could be saved.

When overlaying reconfiguration of one ViSARD core with calculation on another core and using the bitstream preloading feature of the PRC which allows to request the next bitstream from memory before an on-going reconfiguration is finished, long-term average throughput on the Internal Configuration Access Port of the FPGA was shown to stand at 399.981 MB/s. This is 99.995 % of the device-specified maximum of 400 MB/s and affirms that the PRC developed in this thesis allows for very high-speed reconfiguration. Resource usage of the PRC was low at 251 LUTs, 294 FFs, and 1 BRAM tile in the first hardware test.

Bitstream size reduction was most pronounced when using the Xilinx compression method. It is free of adverse effects and does not incur any area or run-time penalty, so it is always recommended to be used. Difference-based generation methods (including torCombitgen) did not allow for omission of configuration frames and therefore did not produce significant size savings in the presented test cases. As far as torCombitgen is concerned, this indicates that it is not particularly fit for the use case of exchanging execution units, but it might still achieve size reductions in other scenarios.

Partition pins were shown to have no significant influence on the resource usage inside and outside of the reconfigurable partition and for the special case examined here (pin savings by moving part of the ALU result multiplexer) reducing them is unlikely to have any beneficial effect. Further tests considering their exact influence on timing and reachable frequencies were therefore not performed.

The PR system developed in the present thesis was proven to have a small resource overhead in the form of the custom PRC, allow for reconfiguration with very high speed and low latency, achieve a sizable resource saving in test designs, and realize predictable timing characteristics fit for hard real-time systems given an AXI slave also satisfying that condition.

Dynamic partial reconfiguration as a technique was shown both in its practical usefulness and its limitations. Prior theses such as [Sch10; Siv10; Seg13] have cited major problems with the tool flow as implemented in the Xilinx Integrated Synthesis Environment (ISE), the predecessor of the Vivado design suite. With the newest available Vivado version at the time of the writing of this thesis (2018.2), the PR flow was generally usable except for some minor bugs such as the inability to associate design files with VHDL libraries in the "Add source" window of the IDE or some problems with non-functioning IP cores in reconfigurable modules. Many limitations such as the inability to share IP cores between RMs or the complete inavailability of the simulation feature in PR designs still severely degrade the design experience and hinder the adoption of partial reconfiguration.

Seen in its entirety, this thesis has shown that despite the improved tool support, PR still requires considerable development effort in order to integrate into a preexisting system and demands many questions to be answered such as the PRC to use, the bitstream storage location, and if any significant gains can be achieved on the target hardware in the first place. At present, partial reconfiguration can most realistically and economically be used in situations that allow for the usage of stock components.

6.2 Future work

There are several topics where additional research, experimentation, and tests could allow for further insights and resource and/or time savings.

First of all, this thesis exclusively analyzed the case of double-precision floating point EUs. All considerations and experiments would have to be repeated for a ViSARD core configured in single precision in order to evaluate whether the results and gains are comparable. It is expected that the gains achievable will be lower overall, since single-precision EUs take up considerably less space in the first place.

Moreover, only reconfiguration on the EU level was considered here. It might make be beneficial to instead reconfigure the whole ALU, a whole ViSARD core, or multiple cores. Working at the processor core level would for example allow to use the ViSARD as-is without major modifications (at least in theory), but introduce more logic that is identical in every RM. Practically speaking, there may be issues with e.g. reset behavior that were not relevant when only exchanging EUs. All of this remains to be investigated and compared to the results of EU-level reconfiguration.

torCombitgen is another matter that warrants further attention. It was not able to reduce the size of bitstreams generated in the test scenarios described, but when for example exchanging whole ViSARD cores as proposed above, it is conceivable that parts of the identical logic could lead to identical configuration frames that allow torCombitgen to omit them, thereby leading to faster reconfiguration.

As discovered in Section 5.4, difference-based bitstream generation allows to save BRAM configuration frames when BRAM is used only in a subset of configurations. If BRAM is relevant for exclusively one configuration, it is not necessary to rewrite it at all. This could be investigated further and lead to the development of a tool that removes BRAM configuration frames from a bitstream and merges the contents from one specific RM into the initial top-level bitstream.

Furthermore, ViSARD utilities such as the assembler or the model-based code generator do not presently have any special features for PR. Especially the automatic generation of PR designs from models provides plenty opportunity for research and could lead to a very user-friendly way of leveraging partial reconfiguration. Modifying the assembler to take the PR process itself into account and to keep track of which EUs are available at what time could allow exchanging them while the ViSARD core is running. Compared to the approach of using multiple cores and reconfiguring one of them while others are running presented here, there is the advantage of potentially having all cores perform calculations at all times.

As far as the work directly implemented here is concerned, tooling improvements such as adding template-based generation of the ALU/EU VHDL code (as proposed in Section 3.2.2) could be made to ease custom-tailoring the ViSARD for a PR use-case.

Although a specific Zynq-7000 SoC was used in practical tests, the results should be comparable for all 7 series devices due to their similar architecture. It is desirable to perform the same kind of research on the newest generation of devices manufactured both by Xilinx and other suppliers. Particularly, Xilinx UltraScale+ FPGAs and their doubled reconfiguration speed of 800 MB/s would make for an interesting target.

Finally, some other techniques surrounding PR could further increase its usefulness to the ViSARD. In particular, bitstream relocation allows to place one partial bitstream into different locations on the device. This could prove to be advantageous when integrating a large number of ViSARD cores on one FPGA which would otherwise require as many partial bitstreams to be generated and stored. Some approaches can be found e.g. in [FA15; DHK17; BKT14; RFG16]. An option to make partial reconfiguration faster is overclocking the ICAP beyond its specified maximum frequency. Experiments on Virtex-5 devices in [HKT11] were successful up to 550 MHz (2200 MB/s throughput), while research done on Zynq-7000 devices in [Nan+17] achieved up to 280 MHz (790 MB/s throughput).

Appendix A

Source code

A.1 PRC

See the files `prc.vhd`, `types.vhd`, and `utility.vhd` in the external appendix [Ker19].

A.2 bin_packer

See the file `bin_packer.py` in the external appendix [Ker19].

A.3 mem_concatenator

See the file `mem_concatenator.py` in the external appendix [Ker19].

A.4 bin_provider

See the file `bin_provider.cc` in the external appendix [Ker19].

Appendix B

Test setups

For all logs in this chapter, text in **bold** is to be entered by the user.

B.1 Simulation

B.1.1 Test bench (Verilog)

See the file `tb.sv` *in the external appendix [Ker19].*

B.1.2 Bitstream packing

```
$ bin_packer.py bitstreams dummy_bitstream1 dummy_bitstream2
Packing 2 bitstreams to bitstreams...
0: @0x001000 size 0x020000: dummy_bitstream1
1: @0x021000 size 0x020000: dummy_bitstream2
Bitstreams successfully packed
Final size: 266240 (0x041000) bytes
```

B.2 Basic test

B.2.1 Generation of test program and data memory files

```
$ VisardCompilerCli.exe 1 -input test_sqrt.txt loopable -output test_sqrt.txt
Files loaded, starting compilation
Using optimization: FCFS
Source compiled, time: 19ms

Total instructions:    44

```

```
Source instructions:    2  (4.5%)
Added NOPs:      42  (95.5%)
Deadlock-resolve instructions:  0  (0.0%)


First Clock of input file 1: 4

Compilation success
$ VisardCompilerCli.exe 1 -input test_natexp.txt loopable -output test_natexp.txt
Files loaded, starting compilation
Using optimization: FCFS
Source compiled, time: 19ms

Total instructions:    36

Source instructions:    2  (5.6%)
Added NOPs:      34  (94.4%)
Deadlock-resolve instructions:  0  (0.0%)


First Clock of input file 1: 4

Compilation success
$ mem_concatenator.py test-prog-c0.txt test_natexp-prog-c0.txt test_sqrt-prog-c0.txt
Concatenating 2 memory files to test-prog-c0.txt...
@    0:    36 lines from test_natexp-prog-c0.txt
@   36:    44 lines from test_sqrt-prog-c0.txt
@   80: --end--
Concatenation successful
$ mem_concatenator.py test-data.txt test_natexp-data.txt test_sqrt-data.txt
Concatenating 2 memory files to test-data.txt...
@    0:     3 lines from test_natexp-data.txt
@    3:     3 lines from test_sqrt-data.txt
@    6: --end--
Concatenation successful
```

B.2.2 Bitstream generation Tcl script

See the file `write_partial_bitstreams_basic_test.tcl` in the external appendix [Ker19].

B.2.3 Bitstream packing

```
$ bin_packer.py /srv/atftp/bitstreams prc_basic_test.runs/impl_1/visard_inst_ALU_ReconfigurableEUs_natexp_partial.bin prc_basic_test.runs/child_0_impl_1/visard_inst_ALU_ReconfigurableEUs_sqrt_partial.bin
Packing 2 bitstreams to /srv/atftp/bitstreams...
0: @0x001000 size 0x037808: prc_basic_test.runs/impl_1/visard_inst_ALU_ReconfigurableEUs_natexp_partial.bin
1: @0x039000 size 0x037808: prc_basic_test.runs/child_0_impl_1/visard_inst_ALU_ReconfigurableEUs_sqrt_partial.bin
Bitstreams successfully packed
Final size: 460808 (0x070808) bytes
```

B.2.4 U-Boot preparation commands

See the file `u-boot_preparation.txt` *in the external appendix [Ker19].*

B.2.5 Boot log

See the file `u-boot_launch.log` *in the external appendix [Ker19].*

B.3 Multi-EU test

B.3.1 Generation of test program and data memory files

```
$ VisardCompilerCli.exe 1 -input test_divide.txt loopable -output test_divide.txt
Files loaded, starting compilation
Using optimization: FCFS
Source compiled, time: 21ms

Total instructions:    46

Source instructions:    2  (4.3%)
Added NOPs:      44  (95.7%)
Deadlock-resolve instructions:  0  (0.0%)


First Clock of input file 1: 4

Compilation success
$ VisardCompilerCli.exe 1 -input test_sqrt_sincos.txt loopable -output test_sqrt_
↪  sincos.txt
Files loaded, starting compilation
Using optimization: FCFS
Source compiled, time: 20ms

Total instructions:    105

Source instructions:    3  (2.9%)
Added NOPs:      102  (97.1%)
Deadlock-resolve instructions:  0  (0.0%)


First Clock of input file 1: 4

Compilation success
$ VisardCompilerCli.exe 1 -input test_sqrt_natexp_multiply.txt loopable -output
↪  test_sqrt_natexp_multiply.txt
Files loaded, starting compilation
Using optimization: FCFS
Source compiled, time: 20ms

Total instructions:    87
```

```

Source instructions:    4  (4.6%)
Added NOPs:      83  (95.4%)
Deadlock-resolve instructions:  0  (0.0%)


First Clock of input file 1: 4

Compilation success
$ mem_concatenator.py test-prog-c0.txt test_divide-prog-c0.txt test_sqrt_
↪  sincos-prog-c0.txt test_sqrt_natexp_multiply-prog-c0.txt
Concatenating 3 memory files to test-prog-c0.txt...
@    0:    46 lines from test_divide-prog-c0.txt
@   46:   105 lines from test_sqrt_sincos-prog-c0.txt
@  151:    87 lines from test_sqrt_natexp_multiply-prog-c0.txt
@  238: --end--
Concatenation successful
$ mem_concatenator.py test-data.txt test_divide-data.txt test_sqrt_sincos-data.txt
↪  test_sqrt_natexp_multiply-data.txt
Concatenating 3 memory files to test-data.txt...
@    0:     4 lines from test_divide-data.txt
@    4:     3 lines from test_sqrt_sincos-data.txt
@    7:     3 lines from test_sqrt_natexp_multiply-data.txt
@   10: --end--
Concatenation successful
```

B.3.2 Bitstream generation Tcl script

See the file `write_partial_bitstreams_advanced_test.tcl` in the external appendix [Ker19].

B.3.3 Bitstream packing

```
$ bin_packer.py /srv/atftp/bitstreams prc_advanced_test.runs/impl_1/visard_inst_
↪  ALU_ReconfigurableEUs_divide_partial.bin prc_advanced_test.runs/child_0_impl_
↪  1/visard_inst_ALU_ReconfigurableEUs_sqrt_sincos_partial.bin prc_advanced_
↪  test.runs/child_1_impl_1/visard_inst_ALU_ReconfigurableEUs_sqrt_natexp_
↪  multiply_partial.bin
Packing 3 bitstreams to /srv/atftp/bitstreams...
0: @0x001000 size 0x078a18: prc_advanced_test.runs/impl_1/visard_inst_ALU_⌋
↪  ReconfigurableEUs_divide_partial.bin
1: @0x07a000 size 0x078a18: prc_advanced_test.runs/child_0_impl_1/visard_inst_ALU_⌋
↪  ReconfigurableEUs_sqrt_sincos_partial.bin
2: @0x0f3000 size 0x078a18: prc_advanced_test.runs/child_1_impl_1/visard_inst_ALU_⌋
↪  ReconfigurableEUs_sqrt_natexp_multiply_partial.bin
Bitstreams successfully packed
Final size: 1489432 (0x16ba18) bytes
```

Bibliography

[Alt18a] *Cyclone V Device Handbook Volume 1: Device Interfaces and Integration.* CV-5V2. Altera Corporation, Nov. 23, 2018. URL: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_5v2.pdf` (visited on 2018-12-08).

[Alt18b] *Stratix V Device Handbook Volume 1: Device Interfaces and Integration.* SV-5V1. Altera Corporation, Aug. 9, 2018. URL: `https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stratix-v/stx5_core.pdf` (visited on 2018-12-03).

[ANW10] Fakhar Anjam, Muhammad Nadeem, and Stephan Wong. "A VLIW Softcore Processor with Dynamically Adjustable Issue-Slots." In: *2010 International Conference on Field-Programmable Technology.* Beijing, China: IEEE, Dec. 2010, pp. 393–398. DOI: `10.1109/FPT.2010.5681444`.

[ARM10] *AMBA 4 AXI4-Stream Protocol Specification Version 1.0.* ARM, 2010.

[ARM11] *AMBA AXI and ACE Protocol Specification: AXI3, AXI4, and AXI4-Lite; ACE and ACE-Lite (Issue D).* ARM, 2011.

[Ash08] Peter J. Ashenden. *The Designer's Guide to VHDL.* 3rd ed. The Morgan Kaufmann Series in Systems on Silicon. Amsterdam; Boston: Morgan Kaufmann Publishers, 2008. ISBN: 978-0-12-088785-9.

[Bec+16] Andreas Becher et al. "Hybrid Energy-Aware Reconfiguration Management on Xilinx Zynq SoCs." In: *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig).* Cancun, Mexico: IEEE, Nov. 2016, pp. 1–7. DOI: `10.1109/ReConFig.2016.7857177`.

[Bha+12] S. Bhandari et al. "High Speed Dynamic Partial Reconfiguration for Real Time Multimedia Signal Processing." In: *2012 15th Euromicro Conference on Digital System Design.* Cesme, Izmir, Turkey: IEEE, Sept. 2012, pp. 319–326. DOI: `10.1109/DSD.2012.74`.

[Bio+16] Alessandro Biondi et al. "A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs." In: *2016 IEEE Real-Time Systems Symposium (RTSS)*. Porto, Portugal: IEEE, Nov. 2016, pp. 1–12. DOI: `10.1109/RTSS.2016.010`.

[BKT14] Christian Beckhoff, Dirk Koch, and Jim Torresen. "Portable Module Relocation and Bitstream Compression for Xilinx FPGAs." In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. Munich, Germany: IEEE, Sept. 2014, pp. 1–8. DOI: `10.1109/FPL.2014.6927480`.

[BLC07] Tobias Becker, Wayne Luk, and Peter Y.K. Cheung. "Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration." In: *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*. Napa, CA, USA: IEEE, Apr. 2007, pp. 35–44. DOI: `10.1109/FCCM.2007.51`.

[But11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd ed. Real-Time Systems Series. New York: Springer, 2011. ISBN: 978-1-4614-0675-4.

[CF15] Luis Andres Cardona and Carles Ferrer. "AC_ICAP: A Flexible High Speed ICAP Controller." In: *International Journal of Reconfigurable Computing* 2015 (2015), pp. 1–15. ISSN: 1687-7195, 1687-7209. DOI: `10.1155/2015/314358`.

[Cla+10] Christopher Claus et al. "Towards Rapid Dynamic Partial Reconfiguration in Video-Based Driver Assistance Systems." In: *Reconfigurable Computing: Architectures, Tools and Applications*. Vol. 5992. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 55–67. ISBN: 978-3-642-12133-3. DOI: `10.1007/978-3-642-12133-3_8`.

[Cla11] Christopher Claus. "Zum Einsatz dynamisch rekonfigurierbarer eingebetteter Systeme in der Bildverarbeitung." Doctoral thesis. München: Technische Universität München, 2011.

[CMS06] Christopher Claus, Florian Helmut Muller, and Walter Stechele. "Combitgen: A New Approach for Creating Partial Bitstreams in Virtex-II Pro Devices." In: *Workshop on reconfigurable computing Proceedings (ARCS 06)* (2006), pp. 122–131. URL: `http://download.lis.ei.tum.de/publications/pub_232333.pdf` (visited on 2018-11-21).

[DdD07] Jeremie Detrey and Florent de Dinechin. "Floating-Point Trigonometric Functions for FPGAs." In: *2007 International Conference on Field Programmable Logic and Applications*. Amsterdam, Netherlands: IEEE, Aug. 2007, pp. 29–34. DOI: `10.1109/FPL.2007.4380621`.

[DeH99] André DeHon. “Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, Why You Don’t Really Want 100% LUT Utilization).” In: *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays - FPGA ’99*. Monterey, California, United States: ACM Press, 1999, pp. 69–78. DOI: `10.1145/296399.296431`.

[DHK17] Khoa Dang Pham, Edson Horta, and Dirk Koch. “BITMAN: A Tool and API for FPGA Bitstream Manipulations.” In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. Lausanne, Switzerland: IEEE, Mar. 2017, pp. 894–897. DOI: `10.23919/DATE.2017.7927114`.

[Di+12] Xie Di et al. “A Design Flow for FPGA Partial Dynamic Reconfiguration.” In: *2012 Second International Conference on Instrumentation, Measurement, Computer, Communication and Control*. Harbin City, Heilongjiang, China: IEEE, Dec. 2012, pp. 119–123. DOI: `10.1109/IMCCC.2012.35`.

[FA15] Timm Friedrich and Kurt Franz Ackermann. “A Flexible Co-Processing Approach for SoC-FPGAs Based on Dynamic Partial Reconfiguration and Bitstream Relocation Methods.” In: *2015 10th International Symposium on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*. Bremen, Germany: IEEE, June 2015, pp. 1–7. DOI: `10.1109/ReCoSoC.2015.7238088`.

[FMM12] Umer Farooq, Zied Marrakchi, and Habib Mehrez. *Tree-Based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*. New York: Springer, 2012. ISBN: 978-1-4614-3593-8.

[Gro08] Ian Grout. *Digital Systems Design with FPGAs and CPLDs*. Burlington: Newnes, 2008. ISBN: 978-0-7506-8397-5.

[Guo+17] Wang Guohua et al. “A Tiny and Multifunctional ICAP Controller for Dynamic Partial Reconfiguration System.” In: *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. Pasadena, CA, USA: IEEE, July 2017, pp. 71–76. DOI: `10.1109/AHS.2017.8046361`.

[HKT11] Simen Gimle Hansen, Dirk Koch, and Jim Torresen. “High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro.” In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. Anchorage, AK, USA: IEEE, May 2011, pp. 174–180. DOI: `10.1109/IPDPS.2011.139`.

[HPA07] John L. Hennessy, David A. Patterson, and Andrea C. Arpaci-Dusseau. *Computer Architecture: A Quantitative Approach*. 4th ed. Amsterdam; Boston: Morgan Kaufmann, 2007. ISBN: 978-0-12-370490-0.

[Hüb+10] Michael Hübner et al. "Fast Dynamic and Partial Reconfiguration Data Path with Low Hardware Overhead on Xilinx FPGAs." In: *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW).* Atlanta, GA: IEEE, Apr. 2010, pp. 1–8. DOI: `10.1109/IPDPSW.2010.5470736`.

[IEE08] *IEEE Standard for Floating-Point Arithmetic.* IEEE Std 754-2008. IEEE, 2008. DOI: `10.1109/IEEESTD.2008.4610935`.

[Inf] Information Sciences Institute, University of Southern California. *Torc Home.* URL: `http://torc-isi.github.io/torc/` (visited on 2018-11-22).

[Int08] *Quantities and Units – Part 13: Information Science and Technology.* IEC 80000-13:2008. International Organization for Standardization, 2008.

[Int11] *Quantities and Units – Part 1: General.* IEC 80000-1:2009/Cor 1:2011. International Organization for Standardization, 2011.

[Int18a] *Intel Stratix 10 Configuration User Guide.* UG-S10CONFIG. Intel Corporation, Nov. 2, 2018. URL: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-config.pdf` (visited on 0218-12-03).

[Int18b] *Intel® Arria® 10 Device Overview.* A10-OVERVIEW. Intel Corporation, Dec. 6, 2018. URL: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_overview.pdf` (visited on 2018-12-08).

[Int18c] *Intel® Quartus® Prime Standard Edition User Guide: Partial Reconfiguration.* UG-20179. Intel Corporation, Sept. 24, 2018. URL: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qps-pr.pdf` (visited on 2018-12-03).

[Kaf11] William Kafig. *VHDL 101: Everything You Need to Know to Get Started.* Burlington: Newnes, 2011. ISBN: 978-1-85617-704-7.

[KBH99] Bernardo Kastrup, Arjan Bink, and Jan Hoogerbrugge. "ConCISe: A Compiler-Driven CPLD-Based Instruction Set Accelerator." In: *Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines.* Napa Valley, CA, USA: IEEE Comput. Soc, 1999, pp. 92–101. DOI: `10.1109/FPGA.1999.803671`.

[Ker15] Philipp Kerling. "Design, Implementation, and Test of a Tri-Mode Ethernet MAC on an FPGA." Bachelor's thesis. Ilmenau: Technische Universität Ilmenau, 2015.

[Ker19] Philipp Kerling. *External Appendix to the Thesis "Conceptual Design and Realization of a Dynamic Partial Reconfiguration Extension of an Existing Soft-Core Processor"*. Ilmenau: Technische Universität Ilmenau, 2019. DOI: `10.22032/dbt.38247`. URN: `urn:nbn:de:gbv:ilm1-2019200184`.

[KF14] Michael Kirchhoff and Wolfgang Fengler. "Realization of an Embedded Hard Realtime Softcore Processor." In: *Autonomous Systems 2014 Proceedings of the 7th GI Workshop*. 2014, pp. 33–42.

[Kir+17] Michael Kirchhoff et al. "Optimizing Compiler for a Specialized Real-Time Floating Point Softcore Processor." In: *2017 8th Annual Industrial Automation and Electromechanical Engineering Conference (IEMECON)*. Bangkok, Thailand: IEEE, Aug. 2017, pp. 181–188. DOI: `10.1109/IEMECON.2017.8079585`.

[Kir+18] Michael Kirchhoff et al. "Increasing Efficiency in Data Flow Oriented Model Driven Software Development for Softcore Processors." In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Tokyo, Japan: IEEE, July 2018, pp. 806–811. DOI: `10.1109/COMPSAC.2018.10343`.

[Kir14] Michael Kirchhoff. "Methodik und Spezialisierung von Softcore-Prozessoren auf Basis von VHDL." Master's thesis. Ilmenau: Technische Universität Ilmenau, Jan. 21, 2014.

[Koc13] Dirk Koch. *Partial Reconfiguration of FPGAs: Architectures, Tools and Applications*. Lecture Notes in Electrical Engineering v. 153. New York: Springer, 2013. ISBN: 978-1-4614-1224-3.

[LD09] Victor Lai and Oliver Diessel. "ICAP-I: A Reusable Interface for the Internal Reconfiguration of Xilinx FPGAs." In: *2009 International Conference on Field-Programmable Technology*. Sydney, Australia: IEEE, Dec. 2009, pp. 357–360. DOI: `10.1109/FPT.2009.5377616`.

[Liu+09] Ming Liu et al. "Run-Time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration." In: *2009 International Conference on Field Programmable Logic and Applications*. Prague, Czech Republic: IEEE, Aug. 2009, pp. 498–502. DOI: `10.1109/FPL.2009.5272463`.

[Max08] Clive Maxfield. *FPGAs: Instant Access*. The Newnes Instant Access Series. Burlington: Newnes, 2008. ISBN: 978-0-7506-8974-8.

[MMF14] Marcus Müller, Torsten Machleidt, and Wolfgang Fengler. "SoC Design for Complex Standalone Optical Measurement Devices." In: *Autonomous Systems 2014 Proceedings of the 7th GI Workshop*. 2014, pp. 66–75.

[Nan+17] A. Nannarelli et al. "Robust Throughput Boosting for Low Latency Dynamic Partial Reconfiguration." In: *2017 30th IEEE International System-on-Chip Conference (SOCC).* Munich: IEEE, Sept. 2017, pp. 86–90. DOI: `10.1109/SOCC.2017.8226013`.

[NDB05] Derek Nowrouzezahrai, Brian Decker, and William Bishop. "Efficient Double-Precision Cosine Generation." In: *Proceedings of the 2005 International Conference on Computer Design.* June 2005. URL: `http://www.cim.mcgill.ca/~derek/files/cosine_report.pdf` (visited on 2018-11-21).

[NL16] Amor Nafkha and Yves Louet. "Accurate Measurement of Power Consumption Overhead during FPGA Dynamic Partial Reconfiguration." In: *2016 International Symposium on Wireless Communication Systems (ISWCS).* Poznan, Poland: IEEE, Sept. 2016, pp. 586–591. DOI: `10.1109/ISWCS.2016.7600972`.

[Oli18] Oliscience. *Home :: OpenCores.* 2018. URL: `https://opencores.org/` (visited on 2018-10-07).

[Öls15] Daniel Ölschlegel. "Erstellung einer grafischen Design-Bibliothek mit Code-Generator für eine Assemblersprache." Diploma thesis. Ilmenau: Technische Universität Ilmenau, Mar. 2, 2015.

[Ope10] OpenCores. *Wishbone B4.* 2010. URL: `https://cdn.opencores.org/downloads/wbspec_b4.pdf` (visited on 2018-10-07).

[Ort+03] Fernando E. Ortiz et al. "A Study on the Design of Floating-Point Functions in FPGAs." In: *Field Programmable Logic and Application.* Vol. 2778. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1131–1134. ISBN: 978-3-540-40822-2. DOI: `10.1007/978-3-540-45234-8_137`.

[PSS17] Luca Pezzarossa, Martin Schoeberl, and Jens Sparso. "A Controller for Dynamic Partial Reconfiguration in FPGA-Based Real-Time Systems." In: *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC).* Toronto, ON, Canada: IEEE, May 2017, pp. 92–100. DOI: `10.1109/ISORC.2017.3`.

[RA14] Jaime Correa Rodriguez and Kurt Franz Ackermann. "Leveraging Partial Dynamic Reconfiguration on Zynq SoC FPGAs." In: *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC).* Montpellier, France: IEEE, May 2014, pp. 1–6. DOI: `10.1109/ReCoSoC.2014.6861353`.

[RFG16] Jens Rettkowski, Konstantin Friesen, and Diana Gohringer. "RePaBit: Automated Generation of Relocatable Partial Bitstreams for Xilinx Zynq FPGAs." In: *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Cancun, Mexico: IEEE, Nov. 2016, pp. 1–8. DOI: `10.1109/ReConFig.2016.7857186`.

[Sch10] Hans-Christian Schwannecke. "Dynamische Partielle Rekonfiguration." Diploma thesis. Ilmenau: Technische Universität Ilmenau, Dec. 16, 2010.

[Seg13] Emil Segerblad. "Evaluation of Partial Reconfiguration in FPGA-Based High-Performance Videosystems." Master's thesis. Västerås: Mälardalen University, 2013. URL: `http://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-19203` (visited on 2018-10-05).

[SGS98] Sergej Sawitzki, Achim Gratz, and Rainer G. Spallek. "CoMPARE: A Simple Reconfigurable Processor Architecture Exploiting Instruction Level Parallelism." In: *Proceedings of the 5th Australasian Conference on Parallel and Real-Time Systems*. Adelaide, Australia: Springer, Sept. 1998, pp. 213–224.

[Siv10] Jacob Siverskog. "Evaluation of Partial Reconfiguration for FPGA Debugging." Thesis. Linköping: Linköpings universitet, 2010. URL: `http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-57714` (visited on 2018-10-05).

[Smi10] Gina R. Smith. *FPGAs 101: Everything You Need to Know to Get Started*. Burlington: Newnes, 2010. ISBN: 978-1-85617-706-1.

[Tar+14] Jimmy Tarrillo et al. "Dynamic Partial Reconfiguration Manager." In: *2014 IEEE 5th Latin American Symposium on Circuits and Systems*. Santiago, Chile: IEEE, Feb. 2014, pp. 1–4. DOI: `10.1109/LASCAS.2014.6820293`.

[Tar17] Vaibbhav Taraate. *PLD Based Design with VHDL*. New York, NY: Springer Berlin Heidelberg, 2017. ISBN: 978-981-10-3294-3.

[Trea] Trenz Electronic GmbH. *TE0703-04 Digital Picture (Top View)*. URL: `http://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/Modules_and_Module_Carriers/4x5/4x5_Carriers/TE0703/REV04/Pictures/TE0703-04%20top.jpg` (visited on 2018-10-08).

[Treb] Trenz Electronic GmbH. *TE0720-02 Digital Picture (Top View)*. URL: `http://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/Modules_and_Module_Carriers/4x5/TE0720/REV02/Pictures/TE0720-02-2IFC3%20top.jpg` (visited on 2018-10-08).

[Tre16] *TE0703 TRM.* rev0.1. Trenz Electronic GmbH, Nov. 22, 2016. URL: http://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/Modules_and_Module_Carriers/4x5/4x5_Carriers/TE0703/REV04/Documents/TRM-TE0703-01-04.pdf (visited on 2018-10-08).

[Tre17] *TE0720 TRM.* v.85. Trenz Electronic GmbH, Nov. 14, 2017. URL: http://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/Modules_and_Module_Carriers/4x5/TE0720/REV03/Documents/TRM-TE0720-03.pdf (visited on 2018-09-26).

[VF14] Kizheppatt Vipin and Suhaib A. Fahmy. "ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq." In: *IEEE Embedded Systems Letters* 6.3 (Sept. 2014), pp. 41–44. DOI: 10.1109/LES.2014.2314390.

[VF18] Kizheppatt Vipin and Suhaib A. Fahmy. "FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications." In: *ACM Computing Surveys* 51.4 (July 25, 2018), pp. 1–39. DOI: 10.1145/3193827.

[Wag17] Lothar Wagner. "Methodische Konzeption und praktische Realisierung einer Erweiterung eines Single-Core Softprozessors zu einer Multi-Core Architektur." Master's thesis. Ilmenau: Technische Universität Ilmenau, Aug. 31, 2017.

[WAN10] Stephan Wong, Fakhar Anjam, and Faisal Nadeem. "Dynamically Reconfigurable Register File for a Softcore VLIW Processor." In: *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010).* Dresden: IEEE, Mar. 2010, pp. 969–972. DOI: 10.1109/DATE.2010.5456908.

[WH95] M.J. Wirthlin and B.L. Hutchings. "A Dynamic Instruction Set Computer." In: *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines.* Napa Valley, CA, USA: IEEE Comput. Soc. Press, 1995, pp. 99–107. DOI: 10.1109/FPGA.1995.477415.

[Xil12] *AXI Reference Guide.* UG761 v14.3. Xilinx, Inc., Nov. 15, 2012. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf (visited on 2018-10-05).

[Xil14] *Programming ARM TrustZone Architecture on the Xilinx Zynq-7000 All Programmable SoC User Guide.* UG1019 v1.0. Xilinx, Inc., May 6, 2014. URL: https://www.xilinx.com/support/documentation/user_guides/ug1019-zynq-trustzone.pdf (visited on 2018-11-19).

[Xil15a] Xilinx, Inc. *Performance and Resource Utilization for Partial Reconfiguration Controller v1.0.* 2015. URL: https://www.xilinx.com/support/documentation/ip_documentation/ru/prc.html (visited on 2018-12-04).

[Xil15b] *Zynq-7000 All Programmable SoC Software Developers Guide.* UG821 v12.0. Xilinx, Inc., Sept. 30, 2015. URL: `https://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf` (visited on 2018-10-05).

[Xil16a] *7 Series FPGAs Configurable Logic Block User Guide.* UG474 v1.8. Xilinx, Inc., Sept. 27, 2016. URL: `https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf` (visited on 2018-10-05).

[Xil16b] *AXI HWICAP v3.0 LogiCORE IP Product Guide.* PG134. Xilinx, Inc., Oct. 5, 2016. URL: `https://www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v3_0/pg134-axi-hwicap.pdf` (visited on 2018-09-26).

[Xil16c] *Integrated Logic Analyzer v6.2 LogiCORE IP Product Guide.* PG172. Xilinx, Inc., Oct. 5, 2016. URL: `https://www.xilinx.com/support/documentation/ip_documentation/ila/v6_2/pg172-ila.pdf` (visited on 2018-11-05).

[Xil17a] *AXI Block RAM (BRAM) Controller v4.1 LogiCORE IP Product Guide.* PG078. Xilinx, Inc., Dec. 20, 2017. URL: `https://www.xilinx.com/support/documentation/ip_documentation/axi_bram_ctrl/v4_1/pg078-axi-bram-ctrl.pdf` (visited on 2018-10-05).

[Xil17b] *CORDIC v6.0 LogiCORE IP Product Guide.* PG105. Xilinx, Inc., Dec. 20, 2017. URL: `https://www.xilinx.com/support/documentation/ip_documentation/cordic/v6_0/pg105-cordic.pdf` (visited on 2018-09-18).

[Xil17c] *Floating-Point Operator v7.1 LogiCORE IP Product Guide.* PG060. Xilinx, Inc., Oct. 4, 2017. URL: `https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf` (visited on 2018-09-18).

[Xil17d] *MicroBlaze Micro Controller System v3.0 LogiCORE IP Product Guide.* PG116. Xilinx, Inc., Dec. 20, 2017. URL: `https://www.xilinx.com/support/documentation/ip_documentation/microblaze_mcs/v3_0/pg116-microblaze-mcs.pdf` (visited on 2018-09-26).

[Xil17e] *Processing System 7 v5.5 LogiCore IP Product Guide.* PG082. Xilinx, Inc., May 10, 2017. URL: `https://www.xilinx.com/support/documentation/ip_documentation/processing_system7/v5_5/pg082-processing-system7.pdf` (visited on 2018-10-05).

[Xil17f] *Virtex-7 T and XT FPGAs Data Sheet: DC and AC Switching Characteristics.* DS183 v1.27. Xilinx, Inc., Apr. 6, 2017. URL: `https://www.xilinx.com/support/documentation/data_sheets/ds183_Virtex_7_Data_Sheet.pdf` (visited on 2018-10-17).

[Xil17g] *Zynq-7000 All Programmable SoC Verification IP v1.0.* DS940 v1.0. Xilinx, Inc., Apr. 28, 2017. URL: `https://www.xilinx.com/support/documentation/ip_documentation/processing_system7_vip/v1_0/ds940-zynq-vip.pdf` (visited on 2018-10-05).

[Xil18a] *7 Series DSP48E1 Slice User Guide.* UG479 v1.10. Xilinx, Inc., Mar. 27, 2018. URL: `https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf` (visited on 2018-10-05).

[Xil18b] *7 Series FPGAs Clocking Resources User Guide.* UG472 v1.14. Xilinx, Inc., July 30, 2018. URL: `https://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf` (visited on 2018-10-12).

[Xil18c] *7 Series FPGAs Configuration User Guide.* UG470 v1.13.1. Xilinx, Inc., Aug. 20, 2018. URL: `https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf` (visited on 2018-10-05).

[Xil18d] *AXI Central Direct Memory Access v4.1 LogiCORE IP Product Guide.* PG034. Xilinx, Inc., Apr. 4, 2018. URL: `https://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf` (visited on 2018-10-06).

[Xil18e] *AXI DMA v7.1 LogiCORE IP Product Guide.* PG021. Xilinx, Inc., Apr. 4, 2018. URL: `https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf` (visited on 2018-10-06).

[Xil18f] *Partial Reconfiguration Controller v1.3.* PG193. Xilinx, Inc., Apr. 4, 2018. URL: `https://www.xilinx.com/support/documentation/ip_documentation/prc/v1_3/pg193-partial-reconfiguration-controller.pdf` (visited on 2018-09-26).

[Xil18g] *UltraScale Architecture Configuration User Guide.* UG570 v1.9.1. Xilinx, Inc., Aug. 16, 2018. URL: `https://www.xilinx.com/support/documentation/user_guides/ug570-ultrascale-configuration.pdf` (visited on 2018-10-17).

[Xil18h] *Virtex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics.* DS893 v1.10. Xilinx, Inc., Jan. 8, 2018. URL: `https://www.xilinx.com/support/documentation/data_sheets/ds893-virtex-ultrascale-data-sheet.pdf` (visited on 2018-10-17).

[Xil18i] *Vivado Design Suite 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide.* UG953 v2018.2. Xilinx, Inc., June 6, 2018. URL: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug953-vivado-7series-libraries.pdf` (visited on 2018-10-05).

[Xil18j] *Vivado Design Suite Tcl Command Reference Guide.* UG835 v2018.2. Xilinx, Inc., June 6, 2018. URL: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug835-vivado-tcl-commands.pdf` (visited on 2018-11-21).

[Xil18k] *Vivado Design Suite Tutorial: Partial Reconfiguration.* UG947 v2018.2. Xilinx, Inc., July 13, 2018. URL: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug947-vivado-partial-reconfiguration-tutorial.pdf` (visited on 2018-10-05).

[Xil18l] *Vivado Design Suite User Guide: Embedded Processor Hardware Design.* UG898 v2018.2. Xilinx, Inc., June 6, 2018. URL: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug898-vivado-embedded-design.pdf` (visited on 2018-10-05).

[Xil18m] *Vivado Design Suite User Guide: Partial Reconfiguration.* UG909 v2018.2. Xilinx, Inc., June 22, 2018. URL: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug909-vivado-partial-reconfiguration.pdf` (visited on 2018-10-05).

[Xil18n] *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing.* UG973 v2018.2. Xilinx, Inc., June 6, 2018. URL: `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug973-vivado-release-notes-install-license.pdf` (visited on 2018-10-05).

[Xil18o] *Zynq UltraScale+ MPSoC Data Sheet: DC and AC Switching Characteristics (DS925).* DS925 v1.13. Xilinx, Inc., Aug. 1, 2018. URL: `https://www.xilinx.com/support/documentation/data_sheets/ds925-zynq-ultrascale-plus.pdf` (visited on 2018-10-05).

[Xil18p] *Zynq-7000 AP SoC and 7 Series Devices Memory Interface Solutions v4.1 User Guide.* UG586. Xilinx, Inc., Apr. 4, 2018. URL: `https://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v4_1/ug586_7Series_MIS.pdf` (visited on 2018-10-06).

[Xil18q] *Zynq-7000 SoC (Z-7007S, Z-7012S, Z-7014S, Z-7010, Z-7015, and Z-7020): DC and AC Switching Characteristics.* DS187 v1.20.1. Xilinx, Inc., July 2, 2018. URL: `https://www.xilinx.com/support/documentation/data_sheets/ds187-XC7Z010-XC7Z020-Data-Sheet.pdf` (visited on 2018-09-26).

[Xil18r] *Zynq-7000 SoC Data Sheet: Overview.* DS190 v1.11.1. Xilinx, Inc., July 2, 2018. URL: `https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf` (visited on 2018-09-26).

[Xil18s] *Zynq-7000 SoC Technical Reference Manual.* UG585 v1.12.2. Xilinx, Inc., July 1, 2018. URL: `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf` (visited on 2018-10-05).

[Xil95] Xilinx, Inc. "Introducing the XC6200 FPGA Architecture." In: *XCELL – The quarterly journal for Xilinx programmable logic users* 18 (1995), pp. 22–23. URL: `https://www.xilinx.com/publications/archives/xcell/Xcell18.pdf` (visited on 2018-12-03).

[Zai+08] Izhar Zaidi et al. "Evaluating Dynamic Partial Reconfiguration in the Integer Pipeline of a FPGA-Based Opensource Processor." In: *2008 International Conference on Field Programmable Logic and Applications.* Heidelberg, Germany: IEEE, 2008, pp. 547–550. DOI: `10.1109/FPL.2008.4630005`.