UNIVERSITY OF CALIFORNIA SAN DIEGO

Hardware Development for Non-Hardware Engineers

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Dustin Alexander Richmond

Committee in charge:

Professor Ryan Kastner, Chair
Professor Sorin Lerner
Professor Trong Nguyen
Professor Leonard Porter
Professor Steven Swanson

2018

Copyright

Dustin Alexander Richmond, 2018

All rights reserved.

The Dissertation of Dustin Alexander Richmond is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2018

DEDICATION

This work is dedicated to my parents who have always provided the support I've needed, even when (I thought) I didn't want it.

EPIGRAPH

The ships hung in the sky in much the same way that bricks don't

Douglas Adams

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Ryan Kastner for his support as the chair of my committee. This thesis would not have happened without his support and guidance.

Chapter 2, is an amended reprint of the material as it appears in the ACM Transactions on Reconfigurable Technology Systems 2015. Jacobsen, Matthew; **Richmond, Dustin**; Hogains, Matthew; and Kastner, Ryan. The dissertation author was the co-investigator and author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 2018. **Richmond, Dustin**; Althoff, Alric; and Kastner, Ryan. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in full, is a reprint of the material as it was submitted to International Conference on Field-Programmable Custom Computing Machines 2018. A revised version appears at International Conference on Field Programmable Logic and Applications 2018. **Richmond, Dustin**; Barrow, Michael; Kastner, Ryan. The dissertation author was the primary investigator and author of this paper.

Chapter 6, is currently being prepared for submission for publication of the material. **Richmond, Dustin**; Kastner, Ryan. The dissertation author was the primary investigator and author of this material.

VITA

2012	Bachelor of Science in Computer Engineering, University of Washington, Seattle
2012	Bachelor of Science in Electrical Engineering, University of Washington, Seattle
2012–2018	Research Assistant, Department of Computer Science and Engineering University of California San Diego
2015	Masters of Science, University of California San Diego
2016	Candidate of Philosophy, University of California San Diego
2018	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

"Synthesizable Higher-Order Functions for C++," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

"Everyone's a Critic:A Tool for Exploring RISC-V Projects" in IEEE Conference on Field Programmable Logic and Applications.

"Generating Custom Memory Architectures for Alteras OpenCL Compiler". In Field Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on, pages 2124. IEEE, 2016.

"Resolve: Generation of High-Performance Sorting Architectures from High-Level Synthesis" ACM International Symposium on Field Programmable Gate Arrays, 2016.

"Tunnel Vision: Documenting Excavations in Three Dimensions with Lidar Technology." Advances in Archaeological Practice, 4(2):192204, 2016.

"RIFFA 2.1: A Reusable Integration Framework for FPGA Accelerators." ACM Transactions on Reconfigurable Technology and Systems, 2015.

"Enabling FPGAs for The Masses" arXiv preprint arXiv:1408.5870, 2014.

"A FPGA Design For High-Speed Feature Extraction from a Compressed Measurement Stream." Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on, pages 18, 2013.

"A Data-Intensive Programming Model for FPGAs: A Genomics Case Study." Symposium on Application Accelerators in High-Performance Computing (SAAHPC12), 2012.

"Diffractive Light Trapping in Crystal-Silicon Films: Experiment and Electromagnetic Modeling." Applied optics, 50(29):57285734, 2011.

"Pressureless Nanoimprinting of Anatase TiO2 Precursor Films." Journal of Vacuum Science & Technology B, 29(2):021603, 2011.

ABSTRACT OF THE DISSERTATION

Hardware Development for Non-Hardware Engineers

by

Dustin Alexander Richmond

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2018

Professor Ryan Kastner, Chair

Recent economics in computer architecture, specifically the end of power-density-performance scaling trends and the inefficiencies in modern processors, has motivated more companies to develop custom hardware. Custom hardware improve important metrics that impact revenue: latency, performance, and power. This has led to the widespread deployment of Field Programmable Gate Arrays (FPGAs) in datacenters, automobiles, communications equipment, and more.

However, hardware development is tedious, time consuming, and costly. There are many challenges: languages are domain specific, few reusable libraries exist, and hardware compilation is slow. In addition, development tools are expensive with a high-degree of vendor lock-in and

domain-specific knowledge. These obstacles have limited hardware development to companies that can afford the associated costs and a few knowledgeable hardware engineers.

Applications for hardware pipelines exist at all scales. Machine learning, computer vision, and text processing are common targets for custom hardware and are pervasive in modern applications. While the details differ between specific applications the computations to well known patterns. These patterns provide a means for abstracting hardware development and increase the accesibility of hardware development for non-hardware engineers.

This thesis presents work on increasing the accessibility of hardware development to non-hardware engineers through the use of common parallel patterns. By abstracting recurring patterns in hardware development we can create languages and architectures with performance comparable to custom hardware circuits, and reduce the cost of hardware development.

Chapter 1

Introduction

Improving throughput, reducing latency, and minimizing power consumption are three never-ending, typically conflicting, constraints for computational systems. As Moore's law [1] and Dennard scaling [2] have ended, companies have been forced to find new ways to improve performance and decrease power. Many companies have turned to custom hardware pipelines: bespoke circuits that efficiently perform application tasks previously done on the CPU. These custom hardware pipelines provide throughput, latency, and power benefits by using transistors more efficiently. However, the development of hardware pipelines is difficult and companies have come to face a different kind of constraint: not enough hardware engineers.

Field Programmable Gate Arrays (FPGAs) are a popular choice for implementing custom hardware pipelines. FPGAs are computing architectures with reconfigurable routing and logic gates that allow highly-efficient circuit implementations. FPGAs are more efficient (Performance/Watt) than other computational substrates like Digital Signal Processors (DSPs), CPUs, or GPUs and with lower cost, and faster development cycles than Application Specific Integrated Circuits (ASICs). FPGAs have been used to implement custom hardware pipelines to improve performance [3, 4], increase security [5, 6], and reduce latency [7, 3, 4]. These projects have also been deployed at scale in datacenters [8, 9, 10], in processors [11], in cars [12, 13], and in cell-phone towers [14]. With the widespread deployment and availability of FPGAs, the need for hardware developers is growing.

Unfortunately, developing custom hardware pipelines on FPGAs is notorously difficult. This is because utilizing an FPGA in any system, let alone a data center, is complex, time consuming, and therefore [15]. First, users must identify, design, and optimize a custom hardware architecture for an FPGA. Second, they must choose a communication medium (e.g., PCIe, Ethernet), write and integrate low-level Hardware Development Language (HDL) communication cores. Then users must write kernel drivers to communicate between the CPU and the FPGA. Next, the integration must be compiled, tested, debugged, and verified. These skills are possessed by a small subset of hardware engineers with years of experience — for example — there are 23 (!) named authors on the initial paper describing the FPGA-based Microsoft Catapult project [8].

What is needed are solutions that increase the accessibility of hardware development for non-hardware engineers. This problem has been approached at many abstraction levels: Existing software languages have been extended to create new Hardware Development Languages (HDLs) that reduce verbosity [16, 17, 18]. Communication frameworks have been developed to abstract low-level details of data transfer [19, 20, 21, 22]. High-Level Synthesis (HLS) tools have been developed that emit complete hardware accelerators from algorithm descriptions in software languages [23, 24, 25, 26, 27, 28]. System synthesis tools emit complete systems by integrating HLS tools and communication frameworks [29, 28, 25]. Recent projects have created open-source, high-level libraries to ease FPGA control and communication. Finally, processor instruction sets have been developed that promotes application-specific extensions and customization with support from software toolchains [30]. All these projects have increased the productivity of hardware engineers, but little has been done to increase the general accessibility of hardware development.

Several challenges remain that impede the accessibility of hardware development for non-hardware engineers. First, vendor-specific tools, chip-specific interfaces, and low-level architecture differences inhibit re-use and force designers to re-engineer solutions for every new project. Second, HLS tools lack common abstractions found in software languages, which im-

pedes re-use and familiarity. Three, the increasing number of processor projects, implementation languages, and interface standards hinders evaluation and adoption. Four, hardware targets lack detailed tutorials for non-hardware engineers, preventing uptake. Finally, the FPGA field still suffers from lengthy compile times and that prevent rapid prototyping, experimentation, and development.

This dissertation describes several projects to make hardware architecture development more accessible to non-hardware engineers. We develop RIFFA [20, 21] to address vendor-specific communication barriers in Chapter 2. We create and study a library of synthesizable higher-order functions to provide a common software abstraction in hardware development in Chapter 3. We introduce PYNQ and develop instructional materials, and tools for evaluating soft processor projects in Chapter 4 and Chapter 5. Finally, Chapter 6 demonstrates that an architecture of interconnected soft processors can build a performant general-purpose pipeline accelerator. **In short, this thesis will demonstrate that by abstracting recurring patterns in hardware development we can create languages and architectures with performance comparable to custom hardware circuits.**

The following sections provide an outline and motivation for each chapter of this thesis:

1.1 A (Vendor Agnostic) Reusable Integration Framework for FPGA Accelerators

Hardware pipelines must communicate through external interfaces like Ethernet or PCI Express, or on-chip interfaces like AXI or Avalon. While the underlying protocols are standardized the implementations are subject to vendor-specific wire names, packet formats, and configuration registers. In addition, these interfaces have timing requirements that often necessitate vendor-specific optimizations that can only be performed in low-level Hardware Development Languages (HDLs) inaccessible to non-hardware engineers [15]. For this reason, developing re-usable communication interfaces is inaccessible to even hardware engineers.

The market is filled with expensive solutions from secondary vendors. Companies like Northwest Logic, PLDA, Pico Computing and Xillybus provide PCI Express communication solutions, while Maxeler, Convey and others provide full development environments with languages and simulators. Some companies provide research pricing for academic groups, but the cost can be prohibitive, comes with little support, and no source code. The FPGA community needs solutions that are free and open-source to encourage prototyping and development.

Open-source and research projects have attempted to address this problem: Microsoft Research's Simple Interface for Reconfigurable Computing (SIRC) [31] uses 100 Mb Ethernet on Windows; Open Component Portablity Infrastructure (OpenCPI) [32] provides Ethernet ant PCIe implementations; MPRACE [33] runs only on Linux and exposes a low-level buffer protocol; and DyRACT [34] only supports Xilinx boards. These projects have limited support for operating systems, configurations, vendors, and languages. None of these solutions provide cross-platform support for multiple languages, or vendor hardware restrictions. These problems led to the development of RIFFA 2.0 [20] with broader support for Xilinx PCIe-enabled boards, with four language APIs, and multiple-OS support.

In this chapter, we describe the development of RIFFA 2.1 and RIFFA 2.2 which add pan-vendor support using re-usable hardare cores. We specifically describe our work on RIFFA 2.2, which re-architected the underlying transmit and receive channels for all vendor PCIe Express interfaces.

1.2 Higher-Order Functions for Hardware Accelerators

Another obstacle to hardware architecture development are the abstractions provided by languages. Hardware development is traditionally done with Hardware Development Languages (HDLs). HDLs expose wiring, registers, and logic gates which make them verbose with few abstractions [15]. Modern HDLs alleviate this somewhat by adding programming abstractions like interface structures, standard template libraries, and provide higher-order functions to

implement repeatable patterns [16, 17, 18]. Still, optimizing designs requires detailed knowledge about the underlying architecture, like the number of inputs in a FPGA Look-Up-Table, or bit-width of an addition core, or relative speeds of paths on a chip. This highly domain-specific knowledge is a huge barrier to FPGAs which has lead to a push to make hardware design more like software design and enable more software engineers [35] to write hardware cores.

High-Level Synthesis (HLS) tools are being developed to raise designer's level of abstraction from logic gates, registers, wires, and clocks to algorithm development. These tools take a program in a high-level language like C/C++, schedule operations, and emit an HDL circuit that implements the desired behavior [29, 26, 36, 25, 28]. Annotations added by the user direct tools to implement fine-grain pipeline, coarse-grain pipeline, or vector parallelism. More recent work has focused on the development of system synthesis tools, which emit complete systems by integrating the output of HLS tools and communication frameworks [29, 28, 25]. However, state-of-the-art C/C++ synthesis tools still suffer from some of the same issues of tools they were developed to replace: they lack libraries implemeting common patterns.

We have created an open-source library of higher-order functions synthesizable in C/C++ hardware development tools. We implement six common algorithms on a PYNQ board and conclude that our library produces results that are generally statistically indistinguishable traditional techniques. This library increases the commonalities between hardware architecture and software application development.

1.3 PYNQ: A Hardware Protoyping Environment

Trends toward higher performance and lower power in mobile computing have pushed FPGA vendors to develop System-on-Chip (SoC) devices. These devices combine ARM processors and FPGA fabrics on a single die to provide high performance and low power for embedded devices. They have been widely used in automobiles [13, 12], in communications [37], robotics [38], and computer vision [39, 27]. The breadth these of applications demonstrate the utility of

SoC devices. However, few development environments for these devices exist.

Many prototyping platforms exist on the embedded prototyping market. These are boards, packaged with Integrated Development Environments (IDEs), for example: Arduino provides a simple bare-bones systems programmed with a C++-like language and IDE with hundreds of libraries; Raspberry Pi is a linux-based, low-cost board with Python libraries and the IDLE IDE; BeagleBone is another linux-based system with a Javascript-like language and Cloud9 IDE to name a few of the most popular. All of these projects provide development boards, extensive libraries, and IDEs which increases the accessibility of the platforms.

One obstacle to FPGA development has been the absence of a standard, development platform. Many boards have been provided, but none match the ease, accessibility, and cost of software projects: Avnet has developed many iterations of the ZedBoard with custom linux distributions, Terasic has developed the DE1 educational board, and Digilent has developed the Arty line. These boards are provided with limited libraries, and rely on long compile times in the vendor tool chains for hardware development. In order to make hardware development more accessible, the community needs standard, low-cost, entry-level development boards for rapid prototyping and development, backed by open-source libraries and accessible in high-level languages.

Recently Xilinx has released project PYNQ - a collection of Python libraries and drivers for ZYNQ SoCs with an Ubuntu Linux image. PYNQ-branded boards have low power ARM processors, a small form-factor, and standard pin headers (Arduino, PMOD, and RasPi). Existing PYNQ applications provide common embedded communcation interfaces like I2C, SPI, and UART. Tutorials and documentation are delivered as web-based Jupyter notebooks served by the PYNQ board. Because of the accessibility of the web-based IDE, PYNQ is being rapidly adopted as a teaching platform across several universities, Tampere University of Technology, BYU [40], the University of Washington, and at UC San Diego. The PYNQ project and boards provide many of the abstractions and libraries that are common in software development boards. However, PYNQ still does not solve two issues: long compile times, and few tutorials for building PYNQ

applications.

Building a PYNQ application comprises a swath of interdependent topics: the Vivado block diagram editor, IP and algorithm development, hardware interfaces, physical memory spaces, drivers, and Python packaging. This has been a conscious (and reasonable) design choice of the PYNQ team to focus on a well-built set of libraries and APIs to aid all-programmable development. Instead, the task of developing tutorials and learning materials has been given to the community.

We have created a set of tutorials demonstrating how to build PYNQ applications using High-Level Synthesis (HLS) tools. Our lab curriculum is split into three units where each unit is a set of independent Jupyter Notebooks teaching a concept. At the end of each tutorial the readers have created a working PYNQ application. These tutorials should help readers overcome the introductory challenges of hardware architecture development.

1.4 A Survey of Open-Source RISC-V Processors

The RISC-V project provides a flexible instruction set architecture (ISA) with no licensing fees, a large (and growing) software infrastructure, and broad adoption. The RISC-V ISA is highly flexible with 4 architectural widths, and 13 codified extensions. The flexibility, and openness of the RISC-V specification has lead to a proliferation of open-source RISC-V projects tailored to specific goals. These features have made it a natural choice for imlementing a custom accelerator on an FPGA, since it is easy to tailor to specific application needs.

However the number of open-source RISC-V projects presents a challenge to developers. Cores can use one of many languages, interface standards, architectural features, and worse, different ISA extensions. This makes it difficult to evaluate the relative benefits of open-source RISC-V projects, and presents a challenge to anyone hoping to choose a RISC-V project to use as a hardware accelerator.

To address this problem we have built a tool to evaluate RISC-V projects using PYNQ.

We use our tool to evaluate 5 RISC-V projects and compare against the Xilinx MicroBlaze processor. Our results indicate that while RISC-V projects have a large variance in results, the best RISC-V soft processors are competitive with the Xilinx MicroBlaze on measurements of the maximum clock frequency, area, execution time on standard benchmarks, and executed instructions. Along with tutorials for extending our work to other processors, this work will help engineers overcome the challenges above.

1.5 A General-Purpose Pipeline Architecture

Processors are inefficient on modern workloads [41]. Traditional processor architectures have inherent inefficiencies in computations like convolution that are widespread in computer vision, machine learning, and linear algebra. Data movement and re-use in these applications does not map well to modern processor register files and instruction sets. This has pushed many companies to develop custom hardware to improve performance, reduce latency, and save power. [8, 9, 10, 42].

Hardware development is significantly more challenging than software development. Hardware development requires engineers with extensive domain specific knowledge and experience [35]. Hardware development languages are inabstract, with few high-level application libraries [15]. These problems are compounded by the speed of hardware compilation flows; CPU tool flows compile on the order of miliseconds to minutes, while FPGA binaries take tens of minutes to hours and days [43, 44]. These challenges have limited hardware development to companies that can afford the associated costs.

However, these applications are not limited to companies with deep pockets; the same challenges exist at all scales but the cost of hardware development is prohibitive. What is needed is a pipeline architecture that is general enough to capture common parallel patterns [30] and can be re-programmed from high-level interfaces.

In this chapter we propose a general-purpose pipeline architecture using interconnected

RISC-V processors. We perform a study to understand our architecture's performance relative to optimized libraries on an ARM processor and optimized libraries in hardware. We conclude that a heterogeneous collection of processors provides the best performance, and can outperform two generations of ARM processor with performance comparable to static hardware pipelines.

Chapter 2

RIFFA 2.2: A (Vendor Agnostic) Reusable Integration Framework for FPGA Accelerators

2.1 Introduction

FPGAs are being used by an ever widening audience of designers, engineers, and researchers. These designers are frequently non-hardware engineers as tools such as Xilinx Vivado High Level Synthesis and the Bluespec language are lowering the barriers to entry for FPGA use. Many of these use cases will require high bandwidth input and output between the FPGA and a traditional CPU workstation. When faced with this problem one can either write their own interconnection, license an existing solution from a vendor, or use an open source solution.

Writing an interconnect is a significant amount of work and is a major barrier to FPGA development for non-hardware engineers. FPGAs are flexible enough to connect to virtually any device. However, this flexibility also makes it challenging to connect to virtually any device. The protocol standards that make workstations interoperable must be implemented from the physical layer on up in order for the FPGA to interface with it. This can be a large obstacle to overcome for most designers. In many cases, implementing the interface logic can match or exceed the effort required for implementing the original application or control logic.

Several commercial solutions exist that can be licensed from vendors such as: Northwest Logic, PLDA, and Xillybus. These solutions are PCI Express (PCIe) based. PCIe based interconnects have become the de facto standard for high bandwidth FPGA-PC communication because of PCIe's ubiquitous presence, low latency, and scalable performance. These solutions are high performing and available for most modern FPGA devices. However, the licensing costs can be prohibitively high and often tied to vendor specific hardware. Some offer special pricing for research based licenses, but no source is supplied.

Open source connectivity solutions exist as well, such as: Microsoft Research's Simple Interface for Reconfigurable Computing (SIRC), the Open Component Portablity Infrastructure (OpenCPI), MPRACE, and DyRACT. These solutions offer interconnections over Ethernet or PCIe. SIRC is a full solution with high level APIs and hardware interfaces. It works out of the box with minimal configuration. Unfortunately, it only runs over 100 Mb Ethernet and is only supported on Windows operating systems. This limits the bandwidth and platform. OpenCPI is designed to use either Ethernet or PCIe to connect components such as FPGAs, GPUs, or DSPs. The APIs are general enough to support a number of different components. This flexibility however makes configuring and using OpenCPI difficult and overwhelming. The MPRACE project provides a PCIe based framework that runs on Linux. It provides a low level buffer management API and a DMA IP controller. This low level API is usable but not as well suited for transferring data to and from software applications. Moreover, it only supports Linux platforms. Lastly, the DyRACT project provides a platform for FPGA based accelerators needing PCIe connectivity, DDR3, and partial reconfiguration. It is a nice framework for those needing partial reconfiguration, PCIe connectivity, a clock manager, and memory controller.

There also attempts to integrate FPGAs into traditional software environments. Commercial solutions such as Maxeler, Convey, and National Instruments provide full development environments along with communications frameworks. This class of solutions works only with custom vendor hardware and includes custom programming languages. Similarly, the Altera OpenCL HLS solution includes support for PCIe based FPGA communication as part of their

SDK. The SDK compiles OpenCL kernels to FPGA primitives and produces an OpenCL execution environment that executes on a FPGA instead of a GPU. There are also many framework level attempts to bridge the communications divide between CPUs and FPGA cores. Projects such as: Hthreads [45], HybridOS [46] and BORPH [47] all address this problem. However these solutions utilize custom operating system kernels and often only support CPUs running on the FPGA fabric. All of these frameworks are quite impressive but impose additional programming and runtime constraints, in addition to vendor hardware lock-in.

These problems inhibit FPGA development for all users by increasing non-recoverable engineering costs, and increasing the domain-specific knowledge necessary for FPGA development. What is needed, is a flexible, multi-vendor, multi-language, no-cost solution so that FPGA development can be available for anyone with a desktop computer.

To address these problems, we have developed RIFFA 1.0 [19], and RIFFA 2 [20]. RIFFA is an open source framework that provides a simple data transfer software API and a streaming FIFO hardware interface. It runs over PCIe and hides the details of the protocol so designers can focus on implementing application logic instead of basic connectivity interfaces. It can be integrated with projects built using a variety of tools and method. Both Windows and Linux operating systems are supported and allow communication between multiple FPGAs per host.

In the sections that follow, we present a detailed description of the RIFFA 2.2 design. RIFFA has gone through several iterations. RIFFA 1.0 [19] supported Xilinx devices with limited PCI Express transfer bandwidth and a synchronous API. RIFFA 2.0 removed dependencies on Xilinx IP, replaced the synchronous software API with an asynchronous stream-based API, and maximized PCI Express bandwidth. RIFFA 2.1 added support for PCIe packet reordering and limited Altera support. RIFFA 2.2 rearchitected the transmit and recieve packet formatters to add support for next-generation Xilinx devices. We present a comparison with earlier releases, an analysis of the architecture, and experimental performance results. The chief contributions of this project are:

- RIFFA 2.0: An open source, reusable, integration framework for multi-vendor FPGAs and workstations in PCI Express Gen 2.0 .
- RIFFA 2.1: Improved packet reordering, scatter gather DMA,
- RIFFA 2.2: Multi-vendor/multi-generation support and re-usable packet engines in PCI Express Gen 2.

The remainder of this chapter is organized as follows: Section 2.2 describes the high-level abstraction provided by RIFFA. Section 2.3 describes the hardware and software architecture that provides this abstraction and the evolution that occured from RIFFA 1.0 to RIFFA 2.2. Section 2.4 reports performance results. Finally, we conclude in Section 2.5.

2.2 Design

RIFFA is based on the concept of communication *channels* between software threads on the CPU and user cores on the FPGA. A channel is similar to a network socket in that it must first be opened, can be read and written, and then closed. However, unlike a network socket, reads and writes can happen simultaneously (if using two threads). Additionally, all writes must declare a length so the receiving side knows how much data to expect. Each channel is independent. RIFFA supports up to 12 channels per FPGA. Up to 12 different user cores can be accessed directly by software threads on the CPU, simultaneously. Designs requiring more than 12 cores per FPGA can share channels. This increases the number of effective channels, but requires users to manually multiplex and demultiplex access on a channel.

Before a channel can be accessed, the FPGA must be opened. RIFFA supports multiple FPGAs per system (up to 5). This limit is software configurable. Each FPGA is assigned an identifier on system start up. Once opened, all channels on that FPGA can be accessed without any further initialization. Data is read and written directly from and to the channel interface. On the FPGA side, this manifests as a first word fall through (FWFT) style FIFO interface for

each direction. On the software side, function calls support sending and receiving data with byte arrays.

Memory read/write requests and software interrupts are used to communicate between the workstation and FPGA. The FPGA exports a configuration space accessible from an operating system device driver. The device driver accesses this address space when prompted by user application function calls or when it receives an interrupt from the FPGA. This model supports low latency communication in both directions. Only status and control values are sent using this model. Data transfer is accomplished with large payload PCIe transactions issued by the FPGA. The FPGA acts as a bus master scatter gather DMA engine for both upstream and downstream transfers. In this way multiple FPGAs can operate simultaneously in the same workstation with minimal CPU system load.

The details of the PCIe protocol, device driver, DMA operation, and all hardware addressing are hidden from both the software and hardware. This means some level of flexibility is lost for users to configure custom behaviors. For example, users cannot setup custom PCIe base address register (BAR) address spaces and map them directly to a user core. Nor can they implement quality of service policies for channels or PCIe transaction types. However, we feel any loss is more than offset by the ease of programming and design.

To facilitate ease of use, RIFFA has software bindings for:

- C/C++
- Python 2.7+
- Java 1.4+
- Matlab 2008a+.

Both Windows 7 and Linux 2.6+ platforms are supported. RIFFA supports the following FPGA families from Xilinx and Altera (annotated with Classic or Ultrascale interface type):

- Xilinx Spartan 6 (Classic Interface)
- Xilinx 7 Series (Classic Interface)
- Xilinx Virtex 6 (Classic Interface)
- Xilinx Ultrascale (Ultrascale Interface)

- Altera Arria II (Classic Interface)
- Altera Cyclone IV (Classic Interface)
- Altera Stratix IV (Classic Interface)
- Altera Stratix V (Classic Interface)

RIFFA designs can make use of PCIe data bus widths: 32, 64, and 128. All PCIe Gen 1 and Gen 2 configurations up to x8 lanes are supported. PCIe Gen 3 up to x4 lanes are supported for all devices.

In the next subsections we describe the software interface, followed by the hardware interface.

2.2.1 Software Interface

The interface for the original RIFFA release [19] was a complicated collection of functions that provided users with an array of data transfer options and threading models. These functions were designed under the assumption that every PC initiated call to the FPGA would result in a transfer of data in both directions. It also required data transfers in either direction to be initiated by the PC. User IP cores would need to be designed with this paradigm in mind to function properly with RIFFA 1.0. RIFFA 2 does not impose such restrictions. The interface on the software side has been distilled down to just a few functions. Moreover, data transfers can be initiated by both sides; PC functions initiate downstream transfers and IP cores initiate upstream transfers. The complete RIFFA 2 software interface is listed in Table 2.1 (for the C/C++ API). We omit the Java, Python, and Matlab interfaces for brevity.

There are four primary functions in the API: `open`, `close`, `send`, and `receive`. The API supports accessing individual FPGAs and individual channels on each FPGA. There is also a function to list the RIFFA capable FPGAs installed on the system. A reset function is provided that programmatically triggers the FPGA channel reset signal. This function can be useful when developing and debugging the software application. If installed with debug flags turned on, the RIFFA library and device driver provide useful messages about transfer events. The messages will print to the operating system's kernel log. RIFFA includes this functionality

Table 2.1. RIFFA 2.2 software API (C/C++).

Function Name & Description
`int fpga_list(fpga_info_list * list)` Populates the `fpga_info_list` pointer with info on all FPGAs installed in the system.
`fpga_t * fpga_open(int id)` Initializes the FPGA specified by id. Returns a pointer to a `fpga_t` struct or `NULL`.
`void fpga_close(fpga_t * fpga)` Cleans up memory and resources for the specified FPGA.
`int fpga_send(fpga_t * fpga, int chnl, void * data, int len, int offset, int last, long timeout)` Sends `len` 4-byte words from `data` to FPGA channel `chnl`. The FPGA channel will be sent `len`, `offset`, and `last`. `timeout` defines how long to wait for the transfer. Returns the number of 4-byte words sent.
`int fpga_recv(fpga_t * fpga, int chnl, void * data, int len, long timeout)` Receives up to `len` 4-byte words from the FPGA channel `chnl` to the `data` buffer. The FPGA will specify an offset for where in `data` to start storing received values. `timeout` defines how long to wait for the transfer. Returns the number of 4-byte words received.
`void fpga_reset(fpga_t * fpga)` Resets the FPGA and all transfers across all channels.

because visibility into hardware bus transfers can be very useful when communicating with custom designed FPGA cores.

The software API has only one function to send data and only one to receive data. This has been intentionally kept as simple as possible. These function calls are synchronous and will block until the transfer has completed. Both take byte arrays as parameters. The byte arrays contain the data to send or serve as the receptacle for receiving data. In the send data function, the `offset` parameter is used as a hint for the FPGA. It specifies an offset for storing data at the destination. This can be used to support bulk memory copies between the PC and memory on the FPGA. The `last` parameter is used to group together multiple transfers. If `last` is 0, the destination should expect more transfers as part of the same group. The final transfer will should have `last` set to 1. This grouping is entirely user specified and can be useful in situations where memory limitations require sending multiple partial transfers. Lastly, the `timeout` parameter specifies how many milliseconds to wait between communications during a transfer. Setting this value to an upper bound on computation time will ensure that RIFFA does not return prematurely.

Setting a zero timeout value causes the software thread to wait for completion indefinitely.

```
char buf[BUF_SIZE];
int chnl = 0;
long t = 0; // Timeout
fpga_t * fpga = fpga_open(0);
int r = read_data("filename", buf, BUF_SIZE);
printf("Read %d bytes from file", r);
int s = fpga_send(fpga, chnl, buf, BUF_SIZE/4, 0, 1, t);
printf("Sent %d words to FPGA", s);
r = fpga_recv(fpga, chnl, buf, BUF_SIZE/4, t);
printf("Received %d words from FPGA", r);
// Process results ...
fpga_close(fpga);
```

Figure 2.1. RIFFA 2.2 software example in C.

Figure 2.1 shows an example C application using RIFFA. In this example, the software reads data from a file into a buffer, sends the data to the FPGA, and then waits for a response. The response is stored back into the same buffer and then processed. In this example, the same buffer is used to store the file data and the FPGA result. This is not a requirement. It just makes for a simpler example.

In Practice

Our experience with this interface is positive, but not without drawbacks. Removing the expectation of function-call style bi-directional data transfer allows users to design systems with more flexibility. The software interface can be easily understood as there exists a single function for each basic operation (open, close, send, receive). The interface also allows users to develop IP cores that perform either stream processing or traditional von Neumann style computing (bulk memory copies to FPGA memory).

The interface however does not support some common use cases very well. Specifically, many programming models require some type of read/write capability for FPGA based registers. Strictly speaking, this is supported. But it requires users to write an IP core that maps data

transferred with `offset` parameters to an internal register array (for example). These register accesses require a DMA transfer for every read/write, which is inefficient.

Additionally, the lack of non-blocking function calls makes it cumbersome to perform common stream processing tasks. Consider the example in Figure 2.1. Because the calls `fpga_send` and `fpga_recv` are blocking, all the data must be transferred to the FPGA via the `fpga_send` call before the call to `fpga_recv` can be made. The upstream transfer cannot begin until the call to `fpga_recv` is made. This arrangement may be a problem if the IP core attached to the channel is designed to start sending a response before it receives all the data. Many streaming oriented designs will produce output in this fashion and attempt to start an upstream transaction while the downstream transaction is still running. To avoid a deadlock, users will need to use two threads; one for the call to `fpga_send` and one for the call to `fpga_recv`. This allows both calls to execute in an time overlapping manner as the IP core would expect.

RIFFA would benefit from an expanded software API that supports programmed I/O and non-blocking function calls. Future versions may include these features.

2.2.2 Hardware Interface

The interface on the hardware side is composed of two sets of signals; one for receiving data and one for sending data. These signals are listed in Table 2.2. The ports highlighted in blue are used for handshaking. Those not highlighted are the FIFO ports which provide first word fall through semantics. The value of `DWIDTH` is: 32, 64, or 128, depending on the PCIe link configuration.

Figure 2.2 shows a Verilog example of an IP core that matches the C example code from Figure 2.1. In this example, the IP core receives data from the software thread, counts the number 4-byte words received, and then returns the count.

For upstream transactions, `CHNL_TX` must be set high until all the transaction data is consumed. `CHNL_TX_LEN`, `CHNL_TX_OFF`, and `CHNL_TX_LAST` must have valid values until the `CHNL_TX_ACK` is pulsed. The `CHNL_TX_ACK` pulse indicates that channel has read the parameters and

Table 2.2. RIFFA 2.2 hardware interface

Signal Name	I/O	Description
CHNL_RX_CLK	O	Clock to read data from the incoming FIFO.
CHNL_RX	I	High signals incoming data transaction. Stays high until all data is in the FIFO.
CHNL_RX_ACK	O	Pulse high to acknowledge the incoming data transaction.
CHNL_RX_LAST	I	High signals this is the last receive transaction in a sequence.
CHNL_RX_LEN[31:0]	I	Length of receive transaction in 4-byte words.
CHNL_RX_OFF[30:0]	I	Offset in 4-byte words of where to start storing received data.
CHNL_RX_DATA[DWIDTH-1:0]	I	FIFO data port.
CHNL_RX_DATA_VALID	I	High if the data on CHNL_RX_DATA is valid.
CHNL_RX_DATA_REN	O	Pulse high to consume value from on CHNL_RX_DATA.
CHNL_TX_CLK	O	Clock to write data to the outgoing FIFO.
CHNL_TX	O	High signals outgoing data transaction. Keep high until all data is consumed.
CHNL_TX_ACK	I	Pulsed high to acknowledge the outgoing data transaction.
CHNL_TX_LAST	O	High signals this is the last send transaction in a sequence.
CHNL_TX_LEN[31:0]	O	Length of send transaction in 4-byte words.
CHNL_TX_OFF[30:0]	O	Offset in 4-byte words of where to start storing sent data in the CPU thread's receive buffer.
CHNL_TX_DATA[DWIDTH-1:0]	O	FIFO data port.
CHNL_TX_DATA_VALID	O	High if the data on CHNL_TX_DATA is valid.
CHNL_TX_DATA_REN	I	High when the value on CHNL_TX_DATA is consumed.

started the transfer. The CHNL_TX_DATA_OFF value determines where data will start being written to within the PC's receiving byte array (this is the hardware equivalent to the software `offset` parameter). This is measured in 4-byte words. As described in Section 2.2.1, CHNL_TX_LAST must be 1 to signal the end of a set of transfers. This will unblock the thread waiting in `fpga_recv`. Data values asserted on CHNL_TX_DATA are consumed when both CHNL_TX_DATA_VALID and CHNL_TX_DATA_REN are high.

The handshaking ports are symmetric for both sets of signals. Thus, the behaviors are reversed with downstream transactions. The user core is notified of a new downstream transaction when CHNL_RX goes high. The user core must acknowledge the transaction by asserting CHNL_RX_ACK high for at least one cycle. The CHNL_RX_ACK pulse indicates that the user core has read the parameters and that data can now be delivered into the FIFO. This barrier serves to separate FIFO data between downstream transfers. Back to back transfers may otherwise keep the FIFO full and there would be no way to delineate transfer boundaries.

As with upstream transactions, data will be made available on CHNL_RX_DATA and is valid when CHNL_RX_DATA_VALID is high. Each cycle CHNL_RX_DATA_VALID and CHNL_RX_DATA_REN are high, data present on CHNL_RX_DATA is considered consumed. The channel may present new valid data on that port the following cycle. The values for CHNL_RX_LAST, CHNL_RX_LEN, and CHNL_RX_OFF correspond to those provided via the fpga_send function call on the host PC and should be used as intended.

Timing diagrams for these signals illustrate the upstream and downstream transfers on a cycle by cycle basis. They are available on the RIFFA website: http://riffa.ucsd.edu.

In Practice

The interface requires providing a length value for transfers. This can be problematic in situations where the transfer length is unknown. Consider, for example, a data compression IP core that compresses data received from the host PC. To accommodate this type of situation, the core could buffer the compressed data until it is all generated and then start a upstream transfer. Another solution is to buffer data in chunks and send each in its own transfer. To avoid such workarounds, RIFFA supports early termination on upstream transfers. Designs with unknown upstream transfer size can set CHNL_TX_LEN to an upper bound value, start the transaction and send data as it is generated by the core. When the core finishes producing output, it can lower CHNL_TX, regardless of how much data was actually transferred. The RIFFA channel will interpret this behavior as an early termination and complete the transfer with the data sent thus far. As long as CHNL_TX is high, the transaction will continue until CHNL_TX_LEN words have been consumed.

Despite a minimal set of signals in the hardware interface, we have found that the handshaking protocol can be an obstacle in designs. It requires building a state machine to manage. Many designs simply need a simple FWFT FIFO (or AXI-4 Lite style) interface. In these designs, the handshaking and parameter information are not used but require additional logic and understanding to deal with.

```
parameter INC = DWIDTH/32;
assign CHNL_RX_ACK = (state == 1);
assign CHNL_RX_DATA_REN = (state == 2 || state == 3);
assign CHNL_TX = (state == 4 || state == 5);
assign CHNL_TX_LAST = 1;
assign CHNL_TX_LEN = 1;
assign CHNL_TX_OFF = 0;
assign CHNL_TX_DATA = count;
assign CHNL_TX_DATA_VALID = (state == 5);
wire data_read = (CHNL_RX_DATA_VALID & CHNL_RX_DATA_REN);

always @ (posedge CLK)
  case(state)
  0:  state <= (CHNL_RX ? 1:0);
  1:  state <= 2;
  2:  state <= (!CHNL_RX ? 3:2);
  3:  state <= (!CHNL_RX_DATA_VALID ? 4:3);
  4:  state <= (CHNL_TX_ACK ? 5:4);
  5:  state <= (CHNL_TX_DATA_REN ? 0:5);
  endcase

always @ (posedge CLK)
  if (state == 0)
    count <= 0;
  else
    count <= (data_read ? count+INC:count);
```

Figure 2.2. RIFFA 2.2 hardware example in Verilog.

2.2.3 Changes from RIFFA 1.0

RIFFA 2 is a complete rewrite of the original RIFFA 1.0 release. RIFFA 1.0 only supports the Xilinx Virtex 5 family. RIFFA 2.2 supports all modern FPGA devices from Xilinx and Altera across PCIe Gen 1, Gen 2, and Gen 3[1].

RIFFA 1.0 requires the use of a Xilinx PCIe Processor Local Bus (PLB) Bridge core. Xilinx has since moved away from PLB technology and deprecated this core. The PLB Bridge core limited the PCIe configuration to a Gen 1 x1 link. Additionally, the bridge core did not support overlapping PLB transactions. This did not have an effect on the upstream direction

[1] Up to x4 lanes.

because upstream transactions are one way. Downstream transactions however, must be sent by the core and serviced by the host PC's root complex. Not being able to overlap transactions on the PLB bus results in only one outstanding downstream PCIe transaction at a time. This limits the maximum throughput for upstream and downstream transfers to 181 MB/s and 25 MB/s respectively. The relatively low downstream bandwidth was a chief motivator for improving upon RIFFA 1.0.

RIFFA 1.0 made use of a simple DMA core that uses PLB addressing to transfer data. The hardware interface exposes a set of DMA request signals that must be managed by the user core in order to complete DMA transfers. RIFFA 2 exposes no bus addressing or DMA transfer signals in the interface. Data is read and written directly from and to FWFT FIFO interfaces on the hardware end. On the software end, data is read and written from and to byte arrays. The software and hardware interfaces have been significantly simplified since RIFFA 1.0.

On the host PC, contiguous user space memory is typically scattered across many non-contiguous pages in physical memory. This is an artifact of memory virtualization and makes transfer of user space data difficult. Earlier versions of RIFFA had a single packet DMA engine that required physically scattered user space data be copied between a physically contiguous block of memory when being read or written to. Though simpler to implement, this limits transfer bandwidth because of the time required for the CPU to copy data. RIFFA 2 supports a scatter gather DMA engine. The scatter gather approach allows data to be read or written to directly from/to the physical page locations without the need to copy data.

RIFFA 1.0 supports only a single FPGA per host PC with C/C++ bindings for Linux. Version 2 supports up to 5 FPGAs that can all be addressed simultaneously from different threads. Additionally, RIFFA 2 has bindings for C/C++, Java, Python, and Matlab for both Linux and Windows. Lastly, RIFFA 2 is capable of reaching 97% maximum achievable PCIe link utilization during transfers. RIFFA 1.0 is not able to exceed more than 77% in the upstream direction or more than 11% in the downstream direction.

2.3 Architecture

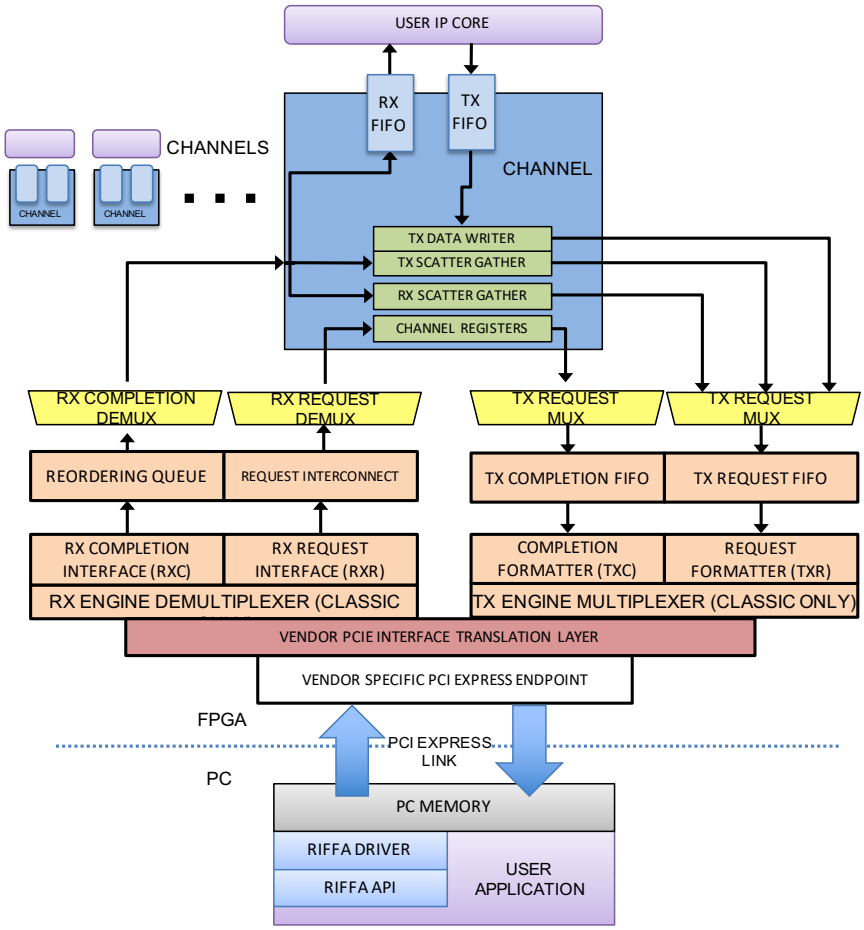


Figure 2.3. RIFFA architecture in RIFFA 2.2. The Channel Layer is described in Section 2.3.1, the DMA Layer (Green) in Section 2.3.1, Multiplexer Layer (Yellow) in Section 2.3.1, Engine Layer (Orange) in Section 2.3.1, and the Translation Layer (Red) in Section 2.3.1

A diagram of the RIFFA architecture is illustrated in Figure 2.3. The top of the diagram shows the hardware architecture, as described in Section 2.3.1. The software architecture is described in Section 2.3.2. The underlying goal of RIFFA architectural decisions is to achieve maximum throughput and minimum latency with the fewest resources. As a result, the architecture can run at line rate, never needing to block or pause, in both upstream (FPGA to Host) and downstream (Host to FPGA) directions.

At each level in the architecture, any potential blocking by a channel or component is

averted through the use of FIFOs, multiplexing, and overlapping execution.

2.3.1 Hardware Architecture

RIFFA's hardware architecture is built from layers of abstractions to hide the underlying PCI Express protocol. At the highest layer of abstraction, the user is presented with a series of channel interfaces. At the lowest layer, RIFFA wraps the vendor-specific PCI Express interface. This is shown in Figure 2.3 and described below.

The RIFFA cores are driven by a clock derived from the PCIe reference clock. This clock's frequency is a product of the PCIe link configuration. It runs fast enough to saturate the PCIe link if data were sent every cycle. User cores do not need to use this clock for their `CHNL_TX_CLK` or `CHNL_RX_CLK`.

Channel Layer

At the top level RIFFA users are presented with the Channel Interface, described in Table 2.2 and shown in blue in Figure 2.3. A set of RIFFA channels provide streaming read and write interfaces, with asynchronous clock-crossings to user cores. These interfaces deal exclusively with payload data.

The FIFO abstraction provided by RIFFA was developed in RIFFA 2. The FIFO interfaces with provide a simple interface for high speed data transfer. The asynchronous clock crossings allow users to operate at whatever clock frequency suits their design instead of operating at the frequency dictated by the PCI Express endpoint.

DMA Engine Layer

The DMA Engine layer resides below the Channel Layer and is shown in green in Figure 2.3. The DMA Engine layer manage reading and writing data from the asynchronous, user-filed, FIFOs to host memory. These transfers happen using the Transmit Scatter Gather Engine, Recieve Scatter Gather Engine, Transmit Writer, and Channel Registers shown in green

in Figure Figure 2.3. This layer was formalized in RIFFA 2.2 to handle the new engine layer interface (specified below).

The Scatter Gather engines are responsible for providing a contiguous memory space abstraction to the channels. When an application sends data to the FPGA, or receives data from the FPGA it must provide a buffer. This buffer appears contiguous to the software application, but in actual fact it is a collection of 4 KB (typically) physical pages. A buffer is presented to the FPGA as a list of pages, or a scatter gather list. The Transmit and Recieve Scatter Gather engines are responsible for reading these lists and providing the physical page addresses as write addresses to the TX Writer, or issuing reads to physical pages that fill the RX Fifo with data. The scatter gather engines are operated using reads and writes to the channel registers. These writes specify the location of scatter gather lists. A complete Scatter Gather transaction is described in 2.3.2.

It is possible (and simpler) to design a scatter gather DMA engine that does not perform such advanced and flexible processing. However, the result is a much lower performing system that does not take advantage of the underlying link as efficiently. There are many examples of this in research and industry.

Multiplexer Layer

The Multiplexer Layer is responsible for routing data between the independent RIFFA channels and is shown in yellow in Figure Figure 2.3. This layer was formalized in RIFFA 2.2. The Multiplexer Layer is responsible for fair arbitration between the independent RIFFA channels on the transmit path, and routing requests back to their source on the receieve path.

Engine Layer

The engine layer is responsible for transmitting and recieving packets to the unified vendor cores. It is shown in orange in Figure 2.3 and in more detail in Figure 2.4 and was added in RIFFA 2.2 to provide pan-vendor support. The abstraction it provides is a wire-based interface

that hides the underlying PCI Express packet format. The recieve path provides two interfaces: Recieve Request Interface (RXR), and Recieve Completion Interface (RXC). These are mirrored on the transmit side: Transmit Request Interface (TXR), and Transmit Completion Interface (TXC).

Inside the Engine Layer, information is formatted into packets. A packet has a three or four word header followed by some number of words of payload (each word is 4 bytes). Data is sent using write packets. Requests for data are made by issuing read packets and receiving completion packets. Completion packets contain the requested data as payload.

The engine layer abstracts vendor-specific packet formats. Xilinx (Pre-Ultrascale) devices have a big-endian data format. Altera devices have a little-endian format, but requires an additional 32-bit word between the header and payload when the incoming (or outgoing) data would not be aligned to a 64-bit boundary. Post-Ultrascale Xilinx devices have a fundamentally different format that is not described in the PCIe Specification; this too is handled by the engine layer.

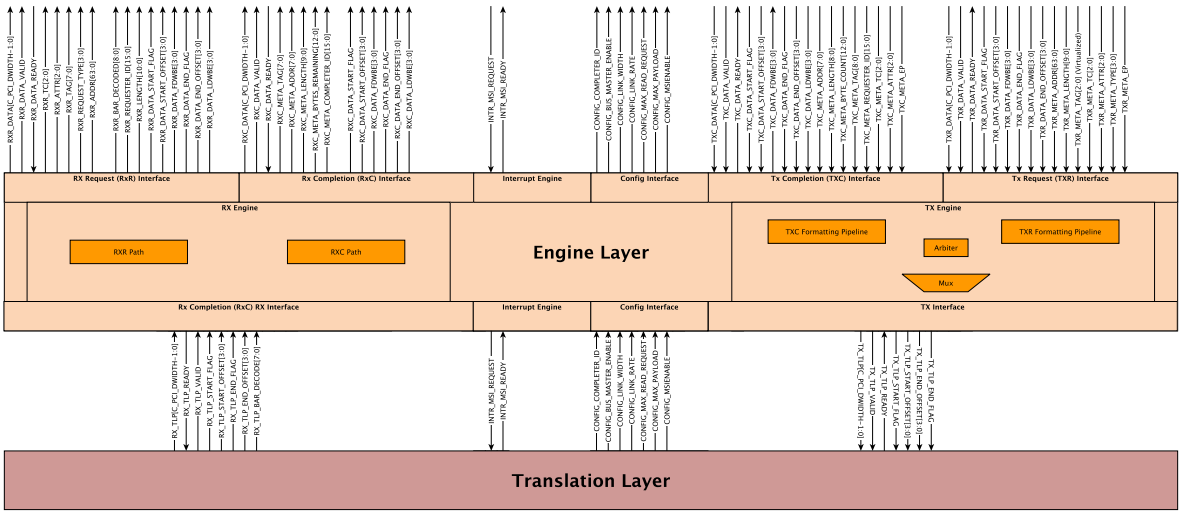


Figure 2.4. Classic Engine Layer Architecture RIFFA 2.2

Finally, the engine layer abstracts the width of the PCIe Express interface. This width

can be 32, 64, or 128 bits wide. Writing a DMA engine that supports multiple widths requires different logic when extracting and formatting PCIe data. For example, with a 32 bit interface, header packets can be generated one 4-byte word per cycle. Only one word can be sent/received per cycle. Therefore, the DMA engine only needs to process one word at a time, containing either header or payload data. However with a 128 bit interface, a single cycle presents four words per cycle. This may require processing three header packets and the first word of payload in a single cycle. If the FPGA vendor's PCIe endpoint supports straddled PCIe transactions, a single cycle on a 128 bit interface can contain data from two different PCIe transactions (completions for example). To achieve the highest performance, the DMA engine must be able to correctly process data for/from those two packets in a single cycle.

The engine layer also contains the reorder buffer, which manages the ordering of read responses. In PCI Express, a single read request can be split into multiple read responses, and interleaved with other in-flight read responses that come before, or after. These packets can arrive out of order and overlap with other completion packets. Packet headers use tag numbers as identifiers for correlating requests with completions. The tag number space is limited and must be managed fastidiously to keep the request pipeline full. Credit limits must be monitored to avoid sending too many packets or causing too many packets to be received.

The PCIe specification allows the host PC's root complex to service outstanding read requests in any order it chooses. This can result in requested data returning to the RX Engine out of order. To reorder the data, each PCIe read request is issued with a unique tag in the header. This tag is selected by the Reordering queue at the time the TX Engine sends the request. The channel number making the request is captured and associated with the tag. When PCIe completion packets arrive at the RX Engine, they contain the request tag in their header. The Reordering Queue stores the payload data in on-chip block RAM (BRAM) in a location corresponding to the tag, and thus the sequence in which it was requested. Tags are issued in increasing order, wrapping at overflow. They are not reused until the received data is consumed by the channel. Received completions are processed in increasing wrapped order as well. In this

manner, data is always returned to the channels in the order in which it was requested. This can be visualized as a two pointer circular tag queue.

A naïve implementation of the Reordering Queue might partition the tag space between channels and use a separate FIFO for each tag. This approach will work, but can consume considerable resources and limit the throughput of each channel. The Reordering Queue instead uses a single logical BRAM to store the payloads for each tag, across all channels. The BRAM is partitioned into equal amounts, with size equal to the maximum read payload size. For the 32 bit bus width, this design is straightforward as only one 32 bit word of payload can be provided each cycle. However, the 64 and 128 bit bus widths may have varying amounts of payload each cycle.

To make efficient use of the BRAM, each word must be written independently, at potentially different RAM positions and word enables. This requires a pair of BRAM ports for each word of payload provided by the bus, and independent RAM positions for each word[2]. This typically means a single BRAM for each word of bus width. A diagram illustrating the use of the Reordering Queue is displayed in Figure 2.5. This scheme requires less BRAMs than one that uses one per tag. There are 32 or 64 tags (depending on configuration) but only four words of payload at the widest 128 bit bus configuration.

In Figure 2.5 the BRAM is initially empty. Three packets arrive (sequence is top to bottom), filling the BRAM. The first and last packet are completions for Tag 1. The middle packet is for Tag 0. Data from the last packet must be written to two separate positions within the Tag 1 buffer space. This is why simply using byte/word write enables will not suffice.

Finally, the reorder buffer also manages control flow for the transmit engine. Hosts control the flow of packets by advertising a credit limit for reads, writes, and completions. These credits represent the maximum number of oustanding packets of each type. The reorder buffer tracks these credits and stalls the transmit path when necessary.

[2]Data in RIFFA is transferred in 32 bit word increments. Therefore, using words as the level of granularity is sufficient.

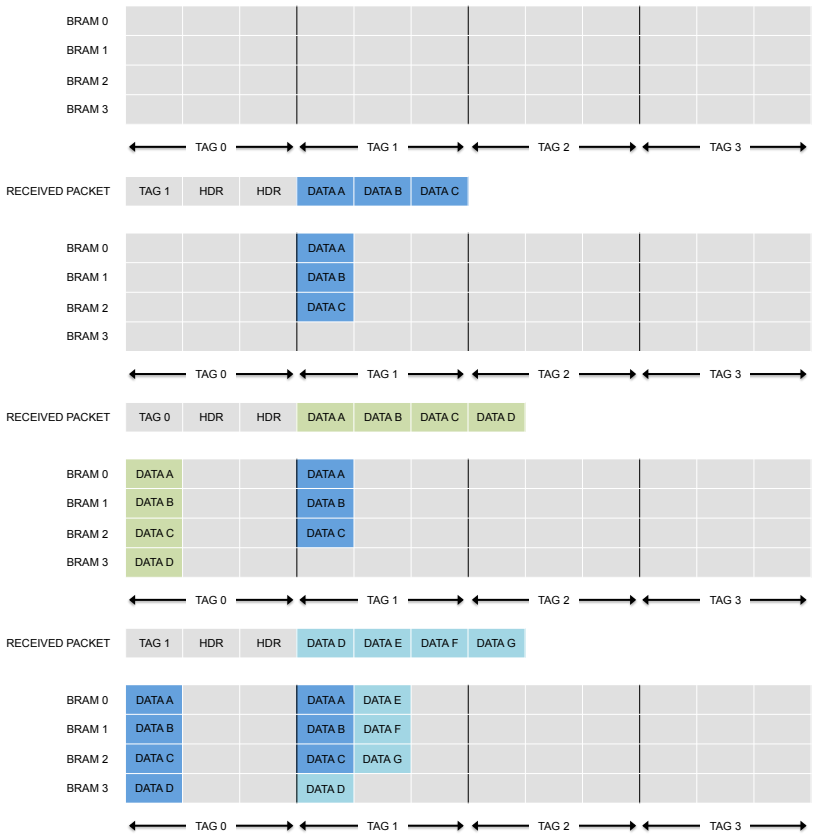


Figure 2.5. Reordering Queue operation for a 128 bit wide bus.

Translation Layer

The translation layer abstracts the three Vendor-specific interface schemes: Altera and Xilinx (Pre-Ultrascale) "Classic" interface specification with a fifo-like interface for Transmit (TX) and Recieve (RX). Xilinx also provides the "Ultrascale" interface for its newer devices that splits these into TXC, TXR, RXC, and RXR interfaces.

Upstream Transfers

Upstream transfers are initiated by the user core via the `CHNL_TX_*` ports. Data written to the TX FIFO is split into chunks appropriate for individual PCIe write packets. RIFFA will attempt to send the maximum payload per packet. It must also avoid writes that cross physical memory page boundaries, as this is prohibited by the PCIe specification. In order to send the data,

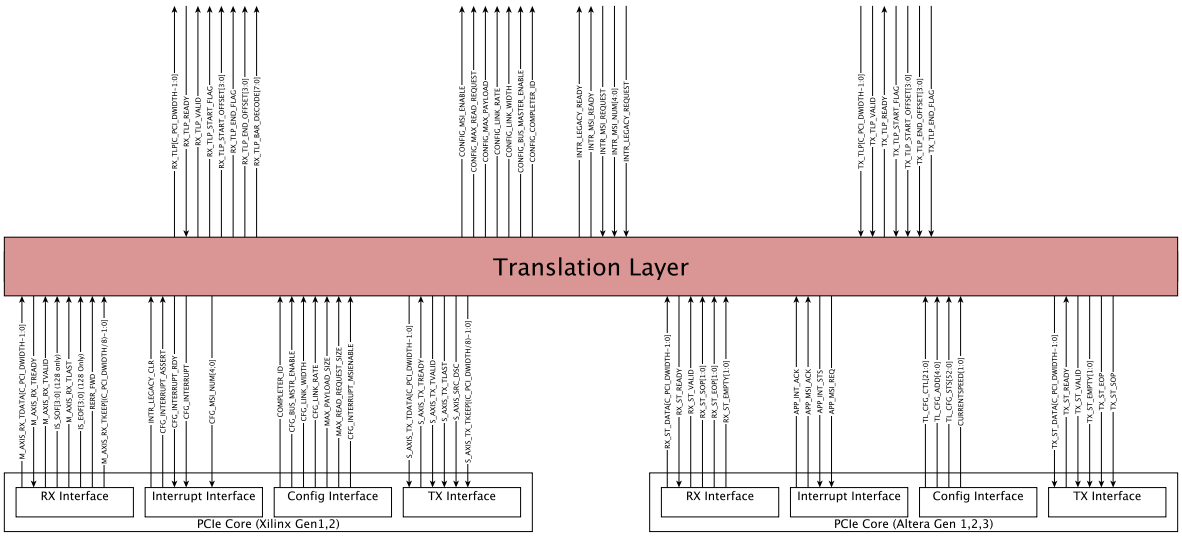


Figure 2.6. Translation Layer for RIFFA 2.2

the locations in host PC memory need to be retrieved. This comes in the form of scatter gather elements. Each scatter gather element defines a physical memory address and size. These define the memory locations into which the payload data will be written. Therefore, each channel first requests a read of list of scatter gather elements from the host. Once the channel has the scatter gather elements, they issue write packets for each chunk of data. Channels operate independently and share the upstream PCIe direction. The TX Engine provides this multiplexing.

The TX Engine drives the upstream direction of the vendor specific PCIe Endpoint interface. It multiplexes access to this interface across all channels. Channel requests are serviced in a round robin fashion. The latency of checking each channel for requests is mitigated by overlapping the search for the next channel during the servicing of the current request. The TX Engine also formats the requests into full PCIe packets and sends them to the vendor specific PCIe Endpoint. The TX Engine is fully pipelined and can write a new packet every cycle. Throttling on data writes only occurs if the vendor specific PCIe Endpoint core cannot transmit the data quickly enough. The Endpoint may apply back pressure if it runs out of transmit buffers. As this is a function of the host PC's root complex acknowledgment scheme, it is entirely system

dependent.

In practice, PCIe Endpoint back pressure occurs most during transfer of small payload PCIe packets. The vendor cores are configured with a fixed amount of header and payload buffer space. Less than maximum sized PCIe packets increases the header to payload ratio. This consumes more header buffers in the vendor Endpoint cores and may cause them to throttle the upstream transmit rate even though payload buffers exist. RIFFA was designed to send maximum sized payload packets whenever possible. Therefore, this tends to not be a problem in practice.

Downstream Transfers

Downstream transfers are initiated by the host PC via the software APIs and manifest on the `CHNL_RX_*` ports. Once initiated, the channel cores request scatter gather elements for the data to transfer. Afterwards, individual PCIe read requests are made for the data at the scatter gather element locations. As with upstream transfers, up to the maximum payload amount is requested and care is taken to avoid requesting data that crosses physical page boundaries[3]. Care is also taken to request data so as to not overflow the RX FIFO. Each channel throttles the read request rate to match the rate at which the RX FIFO is draining. Channel requests are serviced by the TX Engine. When the requested data arrives at the vendor Endpoint, it is forwarded to the RX Engine. There the completion packet data is reordered to match the requested order. Payload data is then provided to the channel.

The RX Engine core is connected to the downstream ports on the vendor specific PCIe Endpoint. It is responsible for extracting data from received PCIe completions and servicing various RIFFA device driver requests. It also demultiplexes the received data to the correct channel. The RX Engine processes incoming packets at line rate. It therefore never blocks the vendor specific PCIe Endpoint core. In addition to higher throughput, this also means the number of completion credits can be safely ignored when issuing requests. Data received by the Endpoint will be processed as soon as it is presented to the RX Engine, avoiding the possibility

[3]Scatter gather elements may be coalesced by the host operating system if they are adjacent and may span more than one page.

of running out of buffer space. After extracting payload data, the RX Engine uses a Reordering Queue module to ensure the data is forwarded to the channel in the order it was requested.

2.3.2 Software Architecture

On the host PC is a kernel device driver and a set of language bindings. The device driver is installed into the operating system and is loaded at system startup. It handles registering all detected FPGAs configured with RIFFA cores. Once registered, a small memory buffer is pre-allocated from kernel memory. This buffer facilitates sending scatter gather data between the workstation and FPGA.

A user library provides language bindings for user applications to be able to call into the driver. The user library exposes the software interface described in Section 2.2.1. When an application makes a call into the user library, the thread enters the kernel driver and initiates a transfer. This is accomplished through the use of the `ioctl` function on Linux and with `DeviceIoControl` on Windows.

The `ioctl` and `DeviceIoControl` functions support flexible, unrestricted communication between user space and kernel space. However, they are unstructured and cannot be used directly by users. This is largely why language bindings are provided. We considered exposing the kernel driver via the standard filesystem functions: `read`, `write`, `open`, `close`, etc. This would remove the need for language specific bindings. Users would be able to use any language and simply call filesystem functions in that language. In fact, the Xillybus driver works this way. The drawback is that any communication not modeled by the filesystem interface will not fit as easily. Functions like `fpga_reset` and `fpga_list` would be difficult to support. Moreover, language bindings provide users additional convenience by performing type conversion and checking.

At runtime, a custom communication protocol is used between the kernel driver and the RX Engine. The protocol is encoded in PCIe payload data and address offset. The protocol consists of single word reads and writes to the FPGA BAR address space. The FPGA communicates

with the kernel driver by firing a device interrupt. The driver reads an interrupt status word from the FPGA to identify the conditions of each channel. The conditions communicated include: start of a transfer, end of a transfer, and request for scatter gather elements. The protocol is designed to be as lightweight as possible. For example, a write of three words are all that is needed to to start a downstream transfer. Once a transfer starts, the only communication between the driver and RIFFA is to provide additional scatter gather elements or signal transfer completion.

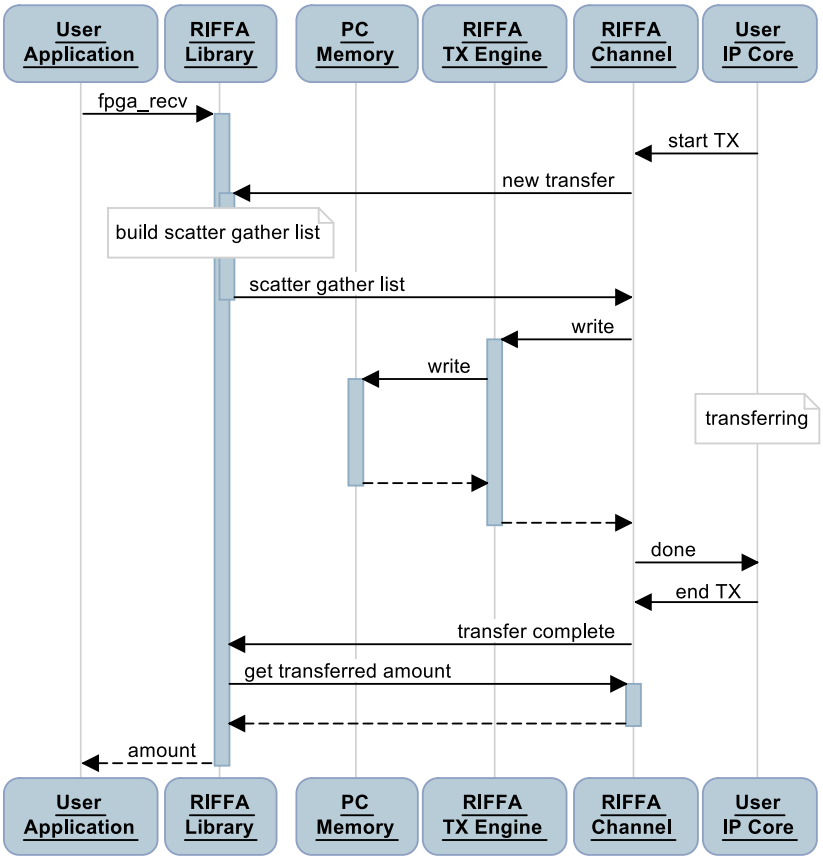


Figure 2.7. Upstream transfer sequence diagram.

Upstream Transfers

A sequence diagram for an upstream transfer is shown in Figure 2.7. An upstream transfer is initiated by the FPGA. However, data cannot begin transferring until the user application calls the user library function `fpga_recv`. Upon doing so, the thread enters the kernel driver

and begins the pending upstream request. If the upstream request has not yet been received, the thread waits for it to arrive (bounded by the `timeout` parameter). On the diagram, the user library and device driver are represented by the single node labeled "RIFFA Library".

Servicing the request involves building a list of scatter gather elements which identify the pages of physical memory correspond to the user space byte array. The scatter gather elements are written to a small shared buffer. This buffer location and content length are provided to the FPGA so that it can read the contents. Each page enumerated by the scatter gather list is pinned to memory to avoid costly disk paging. The FPGA reads the scatter gather data then issues write requests to memory for the upstream data. If more scatter gather elements are needed, the FPGA will request additional elements via an interrupt. Otherwise, the kernel driver waits until all the data is written. The FPGA provides this notification, again via an interrupt.

After the upstream transaction is complete, the driver reads the FPGA for a final count of data words written. This is necessary as the scatter gather elements only provide an upper bound on the amount of data that is to be written. This completes the transfer and the function call returns to the application with the final count.

Downstream Transfers

A similar sequence exists for downstream transfers. Figure 2.8 illustrates this sequence. In this direction, the application initiates the transfer by calling the library function `fpga_send`. The thread enters the kernel driver and writes to the FPGA to initiate the transfer. Again, a scatter gather list is compiled, pages are pinned, and the FPGA reads the scatter gather elements. The elements provide location and length information for FPGA issued read requests. The read requests are serviced and the kernel driver is notified only when more scatter gather elements are needed or when the transfer has completed.

Upon completion, the driver reads the final count read by the FPGA. In error free operation, this value should always be the length of all the scatter gather elements. This count is returned to the application.

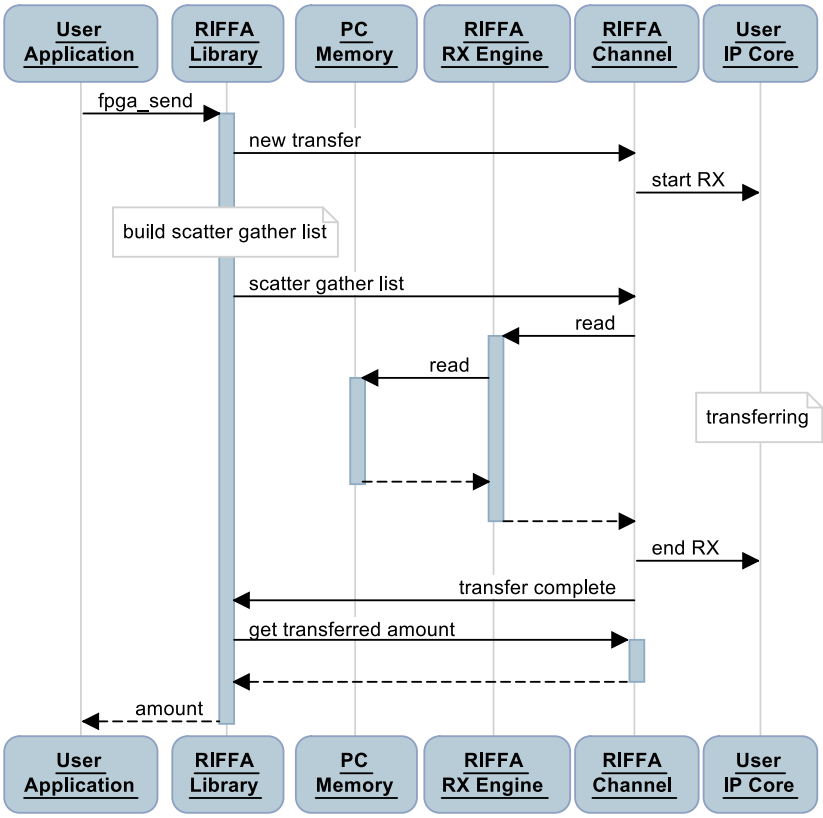


Figure 2.8. Downstream transfer sequence diagram.

The kernel driver is thread safe and supports multiple threads in multiple transactions simultaneously. For a single channel, an upstream and downstream transaction can be active simultaneously, driven by two different threads. But multiple threads cannot simultaneously attempt a transaction in the same direction. The data transfer will likely fail as both threads attempt to service each other's transfer events.

2.4 Performance

We have tested RIFFA 2.2 on several different platforms. A listing of the FPGA development boards and their configurations are listed in Table 2.3. This Table also lists the maximum achieved transfer rates for each configuration. Normalized link utilization is based on maximum negotiated payload limits. Highest values for each interface width are bolded. RIFFA has been

Table 2.3. RIFFA 2 maximum achieved bandwidths, link utilization, and normalized link utilization for upstream (Up) and downstream (Down) directions.

FPGA Board & Configuration	Max. (Up/Down) Bandwidth	Link Utili.	Normalized Link Util.
AVNet Spartan 6 LX150T PCIe Gen 1 x1, 32 bit wide data path, 62.5 MHz	**226 / 212 MB/s**	**90 / 85%**	**96 / 88%**
Xilinx ML605, Virtex 6 LX240T PCIe Gen 1 x8, 64 bit wide data path, 250 MHz	**1816 / 1694 MB/s**	**91 / 85%**	**96 / 87%**
Xilinx VC707, Virtex 7 VX485T PCIe Gen 2 x8, 128 bit wide data path, 250 MHz	**3640 / 3403 MB/s**	**91 / 85%**	**97 / 88%**
Terasic DE4, Altera Stratix IV EP4SGX230KF40 PCIe Gen 1 x8, 64 bit wide data path, 250 MHz	1800 / 1593 MB/s	90 / 80%	96 / 85%
Terasic DE5-Net, Altera Stratix V 5SGXEA7N2F45 PCIe Gen 2 x8, 128 bit wide data path, 250 MHz	3600 / 3192 MB/s	90 / 80%	96 / 85%
Terasic DE5-Net, Altera Stratix V 5SGXEA7N2F45 PCIe Gen 3 x4, 128 bit wide data path, 250 MHz	3542 / 3139 MB/s	90 / 80%	96 / 85%

installed on Linux kernels 2.6 and 3.1, as well as on Microsoft Windows 7. Our experiments were run on a Linux workstation with six 3.6 GHz Intel i7 cores on an Ivy Bridge architecture, using 12 channel RIFFA FPGA designs. The user core on each channel was functionally similar to the module in Figure 2.2. The software was operationally similar to the example listed in Figure 2.1. We used PCIe cores packaged with Xilinx Vivado 2013.2, Xilinx ISE 14.1, and Altera Quartus II 13.1.

Latency times of key operations are listed in Table 2.4. Latencies were measured using cycles counted on the FPGA and are the same across all tested boards and configurations. The interrupt latency is the time from the FPGA signaling of an interrupt until the device driver receives it. The read latency measures the round trip time of a request from the driver to RIFFA (through the RX and TX Engine cores), and back. The time to resume a user thread after it has been woken by an interrupt is the only value that stands out. At 10.4 μs it is the longest delay and is wholly dependent on the operating system.

Bandwidths for data transfers are shown in Figure 2.9 (scale is logarithmic to fit all configurations). The figure shows the bandwidth achieved as the transfer size varies for several

Table 2.4. RIFFA latencies.

Description	Value
FPGA to host interrupt time	3 μs $\pm$ 0.06
Host read from FPGA round trip time	1.8 μs $\pm$ 0.09
Host thread wake after interrupt time	10.4 μs $\pm$ 1.16

PCIe link configurations. Maximum theoretical bandwidths for each configuration are dawn in the same color. The upstream and downstream configurations are provided in the legend. The value in square brackets is the width of the primary bus interface.

Figure 2.9 shows the maximum bandwidth achieved is 3.64 GB/s (using the 128 bit interface). This is 91% utilization of the theoretical link bandwidth. All devices tested reach 89% or higher theoretical bandwidth utilization. This is quite efficient, even among commercial solutions. This is largely due to the low amount of overhead the framework imposes and the fact that RIFFA can run at line rate without blocking. In practice this can keep the PCIe link busy transferring payload data nearly every cycle.

Looking closely at the curves we notice that the upstream direction out performs the downstream direction. This is due to the fact that the FPGA performs reads for downstream transfers and writes for upstream. Upstream transfers are made by the FPGA issuing back to back PCIe writes to the host PC's root complex. Downstream transfers are made by the FPGA issuing back to back PCIe reads. The FPGA must then wait for the host PC's root complex to respond with the requested data. Despite pipelining, this extra one way request adds overhead as the root complex must queue and service each read request.

While not shown on Figure 2.9, RIFFA 1.0 was only able to achieve bandwidths of 181 MB/s and 24 MB/s for upstream and downstream directions respectively. This represents a theoretical link utilization of 73% and 10%. On the same hardware and configuration, RIFFA 2.2 achieves 226 MB/s and 212 MB/s (or 90% and 85%) for the same directions. The relatively poor performance of RIFFA 1.0 was one of the strongest motivators for RIFFA 2.

The performance between the Xilinx and Altera devices is nearly evenly matched. From

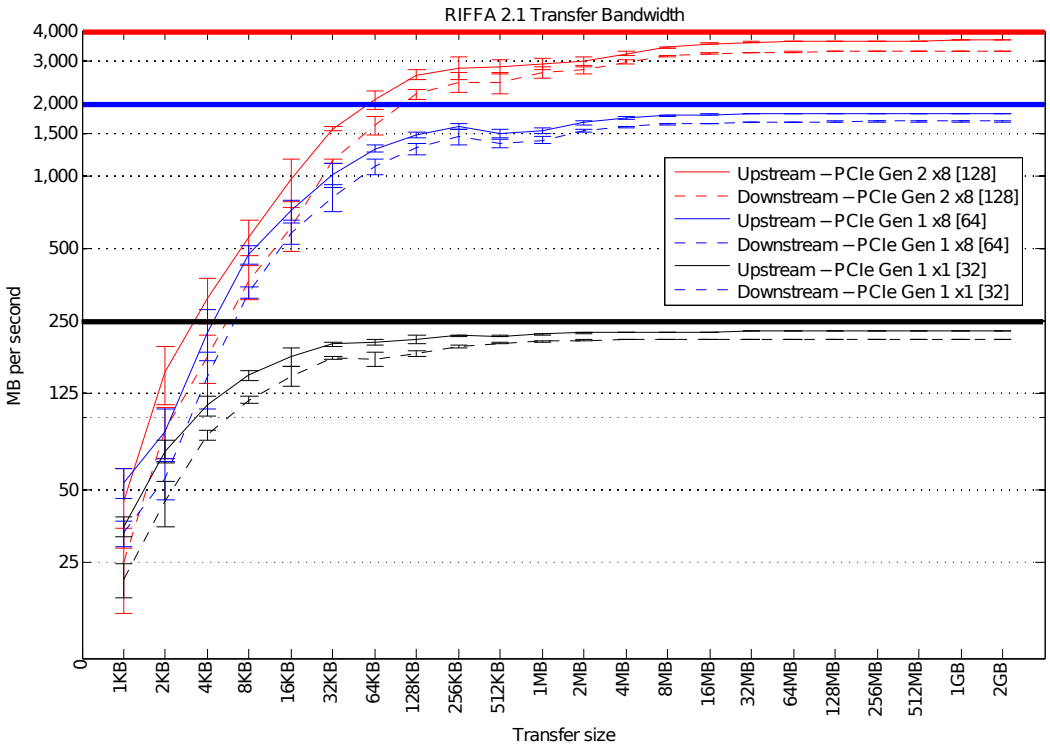


Figure 2.9. Transfer bandwidths as a function of transfer size for several FPGA PCIe link configurations

Table 2.3, it appears that the Xilinx devices out perform the Altera equivalents. However, these differences are primarily in the maximum bandwidth achieved. Figure 2.10 shows the downstream bandwidth for Xilinx and Altera devices configured with a 128 bit interface (scale is linear to show more detail). From this figure, one can see that each device supports a slightly different maximum bandwidth. The difference in maximum bandwidths between the Altera and Xilinx Gen 2 configurations are likely due to packet straddling. On 128 bit wide PCIe configurations, the Xilinx PCIe Endpoint core will provide multiple packets per cycle in the downstream direction. The Altera core will not, which can reduce throughput when saturated.

Within the 256 KB - 2 MB transfer range, the three configurations exhibit slightly different performance. The Gen 3 configuration fares best and has the most even growth. This is not surprising given the Ivy Bridge architecture of the host PC. However the difference in performance between the Altera Gen 2 and Xilinx Gen 2 configurations is a bit puzzling.

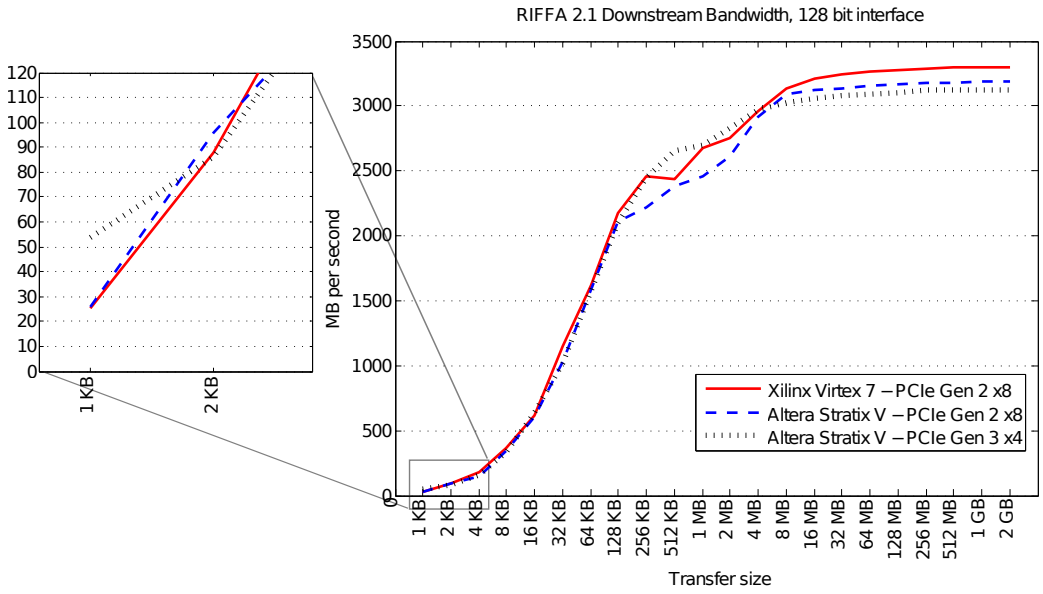


Figure 2.10. Downstream bandwidths as a function of transfer size for Altera and Xilinx devices.

Neither behavior can be attributed to the number of advertised completion credits. Both are endpoint devices and must advertise unlimited completion credits to the root complex. The Xilinx configuration performs better but has half the receive buffer size of the Altera configuration (8 KB vs 16 KB). Thus we can only presume these differences in performance stem from implementation decisions in each vendor's respective PCIe Endpoint core.

Between the Altera Gen 2 and Gen 3 configurations there is a noticeable difference in performance. The difference in maximum bandwidth attained is due to the difference in bandwidths between Gen 2 x8 and Gen 3 x4 links. The Gen 3 lanes are not quite twice as fast as the Gen 2 lanes (500 MB/s vs. 984 MB/s). The maximum theoretical bandwidth for the Gen 3 x4 is actually a bit less than the Gen 2 x8. Given this understanding, the Altera devices perform equally well as measured by the utilizations for both configurations.

One notable exception to this performance equivalence is the bandwidth of the Gen 3 x4 configuration during smaller transfers. Figure 2.10 shows a scaled view of the lower end of the transfer range with more detail. Here we see that for payloads below 2 KB, the use of PCIe Gen 3 can provide double the bandwidth. Moreover, this illustrates that much of the latency in these

small transfers is PCIe protocol specific, not attributed to RIFFA.

2.4.1 Normalized Utilization

Theoretical bandwidth utilization is a commonly compared metric among communication frameworks as it identifies how much overhead the framework imposes. RIFFA 2 performs well using this metric but any utilization less than 100% invites further improvement. A 100% utilization is impossible to achieve because it is calculated by assuming only payload data is sent every cycle. PCIe packets require headers and transfers can require packet exchanges that do not contain payload. Therefore, this is a fundamentally impossible goal.

Identifying and accounting for protocol overheads, for each test case, can be complex and difficult for comparison. However, differences in PCIe configurations between systems can yield considerably different bandwidth utilization values. To account for some of these differences and provide a point of reference for how much they can affect the theoretical utilization metric, we compare against the maximum *achievable* bandwidth. This *achievable* bandwidth is the same as the theoretical bandwidth but also recognizes the existence of packet headers. The header to payload ratio in PCIe packets can be affected significantly by differences in the maximum read/write packet payload. This maximum payload parameter varies across systems. The benefit of using this metric is that it normalizes utilization values across systems, and provides a more system agnostic value.

Using this metric, RIFFA achieves 97% of the maximum achievable bandwidth. This suggests that at most 3% is attributable to RIFFA overhead and link maintenance. In Table 2.3 we provide both the theoretical and achievable bandwidth utilizations for the upstream and downstream directions. The achievable bandwidth utilization is referred to as normalized link utilization in the table. We feel using a normalized utlization that does not vary with PCIe configuration payload limits is a comparably better metric.

2.4.2 Resource Utilization

Resource utilizations for RIFFA, for a single channel, are listed in Tables 2.5 and 2.6. The cost for each additional channel is also listed. Resource values are from the corresponding FPGA devices and configurations listed above. Wider data bus widths require additional resources for storage and PCIe processing. It is useful to note that single channel designs use 4% or less of the FPGA on all our devices. The utilizations listed do not include resources used by the vendor specific PCIe Endpoint core. These core utilizations can vary depending on the configuration values specified during generation and whether a hard block IP exists on the device.

Table 2.5. RIFFA 2 resource utilization on Xilinx devices.

RIFFA (1 channel)	Slice Registers	Slice LUTs	BRAMs	DSP48Es
32 bit Endpoint	4270	3903	12	0
additional channel	2430	2870	7	0
64 bit Endpoint	5350	5110	10	0
additional channel	3175	3584	7	0
128 bit Endpoint	7396	7489	16	0
additional channel	4371	5221	12	0

Table 2.6. RIFFA 2 resource utilization on Altera devices. Altera devices do not support a 32 bit interface.

RIFFA (1 channel)	Resisters	ALMs	RAM bits	DSPs
64 bit Endpoint	5432	5888	406 Kb	0
additional channel	2662	3098	293 Kb	0
128 bit Endpoint	7660	8475	600 Kb	0
additional channel	4146	4608	390 Kb	0

2.4.3 Factors Affecting Performance

Many factors go into attaining maximum throughput. There is enough confusion on the topic that Xilinx has published a whitepaper [48]. The key components affecting RIFFA performance are: transfer sizes, maximum payload limits, completion credits, user core clock frequency, and data copying.

As Figure 2.9 clearly illustrates, sending data in smaller sizes reduces effective throughput. There is overhead in setting up the transfer. Round trip communication between the Endpoint core and the device driver can take thousands of cycles. During which time, the FPGA can be idle. It is therefore best to send data in as large a transfer size as resources will allow to achieve maximum bandwidth.

When generating the vendor specific PCIe Endpoint core, it is beneficial to configure the core to have the maximum values for payload size, read request size, and completion credits. This will give RIFFA the highest amount of capacity when sending data.

The payload size defines the maximum payload for a single upstream PCIe transaction. The read request size defines the same for the downstream direction. At system startup, the PCIe link will negotiate rates that do not exceed these configured maximum values. The larger the payloads, the more payload data each PCIe transaction can carry. This affects effective bandwidth most directly as it reduces the amount time spent transmitting packet header data. Maximum rates can be reached by configuring the IP cores to the highest setting and letting the system determine the negotiated maximum.

Completion credits and their corresponding buffers are used in the PCIe Endpoint to hold PCIe transaction headers and data for downstream PCIe transaction requests. During downstream transfers, completion credits limit the number of in-flight requests. RIFFA does not throttle memory read requests due to completion credit availability. Because RIFFA processes completions at line rate, no data will ever be lost. However more in-flight requests provide greater margin for moving data from the workstation to the user core at maximum bandwidth.

The speed at which a channel's RX FIFO is drained is also a factor. RIFFA will throttle read requests for downstream data to avoid overflowing the channel RX FIFOs (throttled independently). This allows each channel to read received data at whatever rate it chooses. However, to maximize transmission bandwidth, a user core must empty its RX FIFO at the same rate (or faster) than it is filled. Using a clock with a frequency at least as high as that used by RIFFA is recommended to achieve this. Note that the user core can be clocked by any source. It need not

be the same clock that drives the RIFFA.

Lastly, end-to-end throughput performance can be diminished by excessive data movement. Making a copy of a data buffer before sending it to the FPGA takes time. RIFFA's software APIs accept byte arrays as data transfer receptacles. Depending on the language bindings, this may manifest as a pointer, reference, or object. However, the bindings have been designed carefully to use data types that can be easily cast as memory address pointers and be written or read contiguously within the virtual address space without needing to be copied into the kernel (pass by reference, not value). Scatter gather DMA operations perform the actual movement in the physical address space during transfer.

2.5 Conclusion

We have presented RIFFA 2, a reusable integration framework for FPGA accelerators. RIFFA provides communication and synchronization for FPGA accelerated applications using simple interfaces for hardware and software. It is an open source framework that easily integrates software running on commodity CPUs with FPGA cores. RIFFA supports modern Xilinx and Altera FPGAs. It supports multiple FPGAs in a host system, Linux and Windows operating systems, and software bindings for C/C++, Java, Python, and Matlab. This

FPGAs are being used by an ever widening audience of designers, engineers, and researchers. These designers are frequently non-hardware engineers as tools such as Xilinx Vivado High Level Synthesis and the Bluespec language are lowering the barriers to entry for FPGA use. Many of these use cases will require high bandwidth input and output between the FPGA and a traditional CPU workstation. When faced with this problem one can either write their own interconnection, license an existing solution from a vendor, or use an open source solution.

In this chapter, we have also provided a detailed analysis of RIFFA as a FPGA bus master scatter gather DMA design and an analysis of its performance. Tests show that data transfers

can reach 97% of the achievable PCIe link bandwidth. We hope RIFFA will enable designers to focus on application logic instead of building connectivity infrastructure. RIFFA 2 can be downloaded from the RIFFA website at http://riffa.ucsd.edu.

2.6 Acknolwedgements

The authors acknowledge the National Science Foundation Graduate Research Fellowship Program, the Powell Foundation, and ARCS foundation for their support. This research was funded by Xilinx Coporation, and Altera Coporation.

This chapter is an amended reprint of the material as it appears in the ACM Transactions on Reconfigurable Technology Systems 2015. Jacobsen, Matthew; Richmond, Dustin; Hogains, Matthew; and Kastner, Ryan. The dissertation author was the co-investigator and author of this paper.

Chapter 3

Synthesizable Higher-Order Functions for C++

3.1 Introduction

Hardware development tools have been gradually raising their level of abstraction from specifying transistors, to defining gate level circuits, to describing register transfer operations. C/C++ hardware development tools [29, 26, 36, 25, 28] further this trend by enabling the designer to provide algorithmic descriptions of the desired hardware. Yet, despite much progress, there are calls to make hardware design even more like software design, which will allow more software engineers to write hardware cores [35].

A major impediment to this lies in the fact that C/C++ hardware development tools lack many of the conveniences and abstractions that are commonplace in modern productivity languages. Higher-order functions are a prime example. They are a pervasive representation of computational patterns that take other functions as arguments. For example, the higher-order functions `map` and `reduce` shown in Figure 3.1 are the eponymous operators of Google's MapReduce framework [49] and the function `filter` is the semantic equivalent to SQL's `WHERE` clause [50]. Higher-order functions are also useful in hardware development where they can represent common parallel patterns [51, 52] like fast fourier transforms (Figure 3.2), argmin reduction trees [53], sorting networks [54, 50], and string matching [55]. Despite their benefits, higher-order functions are not found in C/C++ hardware development tools.

```python
def mulby2(x):
    return x * 2
def add(x, y):
    return x + y
l = [1, 2, 3, 4]
m = map(mulby2, l)
r = reduce(add, m)
print(r) # Prints '20'
```

(a)

```
array<int, 4> l = {1, 2, 3, 4};
m = map(mulby2, l);
r = reduce(add, m);
```

(b)

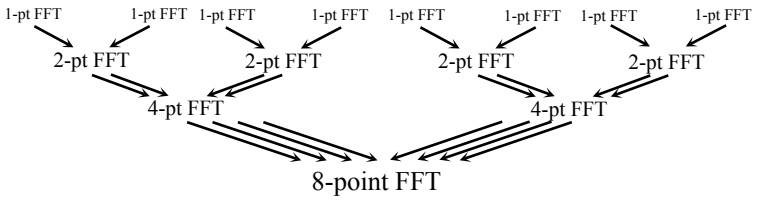
Figure 3.1. (a) Higher-order functions in Python that multiply all values in a list by 2 (`map`), and take the sum (`reduce`). (b) An equivalent in C++ using our library.

Higher-order functions are difficult to implement in C/C++ hardware development tools because parallel hardware must be defined *statically*: types, functions, interfaces, and loops must be resolved at compile time. In contrast, higher order functions typically rely on *dynamic* features: dynamic allocation, dispatch, typing, and loop bounds. Prior work has added higher-order functions to Hardware Development Languages (HDLs) [56, 36, 16], added higher-order functions to domain-specific languages [52], or proposed extensions to C/C++ development tools [51]. None have created synthesizable higher-order functions in a widespread language or tool.

In this chapter we develop a library of synthesizable higher-order functions for C/C++ hardware development tools with a syntax similar to modern productivity languages. Our work leverages recent additions to the C++ language standard to enable seamless integration into a C/C++ hardware design flow.

This work has four main contributions:

- A demonstration of C++ techniques that enable synthesizable higher-order functions
- An open-source library of higher-order functions for C/C++ hardware development tools
- A statistical comparison between our work and loop-based C/C++ implementing six common algorithms on a PYNQ board
- A qualitative comparison between the syntax of our library and a modern high-level language



(a)

```
def fft(sig):
    return divconq(nptfft,
                   bitreverse(sig))
```

(b)

```
template <typename T, size_t LEN>
auto fft(array<T, LEN> sig){
  return divconq(nPtFFTStage, bitreverse(sig));
}
```

(c)

Figure 3.2. (a) A graphical representation of the divide-and-conquer structure of a Fast Fourier Transform (FFT). (b) A python implementation of the FFT algorithm using the higher-order function `divconq` and functions `NPtFFTStage` and `bitreverse`. (c) A C++11 implementation of the Fast Fourier Transform algorithm using `divconq` from our library of higher-order functions.

This chapter is organized as follows: Section 3.2 describes the C++ techniques we use to develop our higher-order functions. Section 3.3 demonstrates how our library can be used to implement the well-known the Fast Fourier Transform (FFT) algorithm, one of many examples in our repository. Section 3.4 presents a comparison of compilation results between our library and loop-based constructs on six common algorithms. Section 3.5 describes related work. We conclude in Section 3.6.

3.2 Building Higher-Order Functions

Higher-order functions are a pervasive abstraction that encapsulate common programming patterns by calling other functions provided as input arguments. Figure 3.1 shows two higher-order functions: `map` applies a function to every element in an array, and `reduce` iterates through an array from left to right applying a function and forwarding the result to the next iteration. Higher-order functions can also implement recursive patterns. Figure 3.2 demonstrates how the

recursive divide and conquer function `divconq` is used to implement the fast fourier transform algorithm. By encapsulating common patterns, higher-order functions encourage re-use.

Higher-order functions for are difficult to implement in C/C++ hardware development tools because parallel hardware must be defined *statically*: types and functions must be resolved, lists that define parallel interfaces must be statically sized, and parallel loops must be statically bounded. In contrast, higher order functions in productivity languages such as Python typically rely on *dynamic* features: polymorphic functions are overloaded with a table of function pointers, functions are passed as global memory addresses for dynamic dispatch, lists are created and resized by memory allocators, and the stack is dynamically resized for recursion. While it is possible to define hardware with dynamic memory allocation, function pointers, and dynamic dispatch the main drawback is efficiency and similarities to general-purpose processors.

In the following subsections we describe how to replace these dynamic features with static techniques to implement synthesizable higher-order functions for C/C++ hardware development tools. By using standard compiler features our work is not limited to a single toolchain. The result of our work is a library of higher-order functions that mimics the behavior of modern productivity languages.

A complete listing of the functions used in this chapter are shown in Table 3.1. The remaining functions can be found in our repository.

3.2.1 Templates (Parametric Polymorphism)

In this section we describe how to use C++ templates to provide the polymorphism required by higher-order functions. Polymorphism is the ability of a data type or function to be written *generically*. For example, the higher-order function `map` must be written generically so that its array argument can be a array of integers, array of arrays, array of classes, or an array of any other type. `map` must also have a generic output type so that it can produce any type of output array. Container classes such as arrays must be able to store integers or booleans. Polymorphism provides the ability to represent repeatable patterns across various input and output types.

Class Templates

Class templates are used to parameterize container classes and algorithms with types, length, and functions. They are pervasive in the C++ the Standard Template Library (STL). We use templated classes like those shown in Figure 3.3 extensively in our work.

Three examples of the the STL `array` class are shown in Figure 3.3a. `arr1` is an `array` of four integers, `arr2` is an `array` of four floats, and `arr3` is a `array` of two array classes, each with four integers (a 2-by-4 matrix). This example demonstrates how template parameters provide generic classes.

Figure 3.3b shows how templated classes are defined. Template parameters can be type names, class names, values, or functions. Template variables can be used to define the type of other template variables. For example `T` is used to define the type of the template parameter `VALUE`.

```
int main(){
  array<int, 4>   arr1 = {1, 2, 3, 4};
  array<float, 5> arr2 = {1.0, 2.0, 3.0, 4.0};
  array<array<int, 4>, 2> arr3 = {arr1};
  return 0
}
```

(a)

```
template <typename T, T VALUE>
class foo{
  T fooval = VALUE;
};
```

(b)

Figure 3.3. (a) Three parameterized instances of the STL `array` class. (b) Defining a templated class `foo`.

Function Templates

Templates are also used to implement polymorphic functions that can handle multiple types with one definition. For example, the higher-order function `map` must be written generically so that its array argument can be a array of integers, array of arrays, array of classes, or an array of any other type. Templates can also be used to pass compile-time constants to a function. This

functionality is required for functions that use the STL `array` class and will be used heavily in our higher-order functions.

```
1  template<typename T>
2  T add(T l, T r){
3    return l + r;
4  }
5
6  template<typename T, unsigned long LEN>
7  int arrayfn(array<T, LEN>& arr){
8    /* Do some array processing*/
9    return 0;
10 }
11
12 int main(){
13   array<int, 3> arr = {0, 1, 2};
14
15   // Three examples without template inference
16   int res1 = add<int>(0, 1);
17   float res2 = add<float>(0.0f, 1.0f);
18   int res3 = arrayfn<int, 3>(arr);
19
20   // The same examples with template inference
21   int res4 = add(0, 1);
22   float res5 = add(0.0f, 1.0f);
23   int res6 = arrayfn(arr);
24   return 0;
25 }
```

Figure 3.4. Two templated functions: `add` and `arrayfn`.

Lines 1-10 in Figure 3.4 show two templated functions: `add` and `arrayfn`. The template parameter `T` provides static type polymorphism to both functions. This means they can be applied to integers, floats, or classes. `arrayfn` has an additional parameter `LEN` that specifies the length of its `array` argument. These functions are called without template inference on Lines 16-19.

Template Inference

Template parameter inference allows template parameters to be inferred from the call site, and is critical for creating succinct higher-order functions that mimic dynamically typed software languages. Template inference starts with the last template parameter, and stops when a parameter cannot be inferred, or all parameters have been inferred.

Figure 3.4 also demonstrates an example of template inference on Lines 21-24. The template parameters of calls `add` and `arrayfn` infer the `T` and `LEN` based on the types of the

input arguments at those callsites. The effect of template inference is to allow designers to write less verbose code.

Functions as Template Parameters (First-Class Functions)

C++ functions can also be passed as template parameters. Unlike software languages, where functions are passed as pointers and dynamically resolved during runtime, functions passed as template parameters are static and synthesizable.

Figure 3.5 demonstrates how the function `truncate` can be passed to the higher-order function `apply` as the template parameter `FN`.

```
1  template <typename TI>
2  unsigned char truncate(TI IN){
3    return (unsigned char)IN;
4  }
5
6  template <typename TO, typename TI, TO (FN)(TI)>
7  TO apply(TI IN){
8    return FN(IN);
9  }
10
11 int main(){
12   int i = 0x11223344;
13   unsigned char res;
14
15   res = apply<unsigned char, int, truncate<int> >(i);
16   // res = 0x44
17   return 0;
18 }
```

Figure 3.5. A C++ function passed as a template parameter.

Template inference cannot be applied to the example in Figure 3.5. `truncate` depends on the type parameters `TI` and `TO`, so it must follow those parameters in the parameter list. `truncate` is not a function argument to `apply` it cannot be inferred. Figure 3.6 demonstrates how we can can aid template inference by wrapping the `truncate` function inside of a struct.

Figure 3.6 demonstrates how the body of the function `truncate` and its template parameters are relocated to the `()` operator inside of the `Truncate` struct. This is often called a class/struct-wrapped function, or functor. By passing the struct `Truncate` we "hide" the template parameters of its function from the template parameter list in `array`. Instead, the compiler infers

```
1  struct Truncate{
2    template <typename TI>
3    unsigned char operator()(TI IN){
4      return (unsigned char)IN;
5    }
6  };
7
8  template <typename TO, class FN, typename TI>
9  TO apply(TI IN){
10   return FN()(IN);
11 }
12
13 int main(){
14   int i = 0x11223344;
15   unsigned char res;
16   // Previously: apply<unsigned char, int, truncate<int> >(i);
17   res = apply<unsigned char, Truncate>(i);
18   // res = 0x44
19   return 0;
20 }
```

Figure 3.6. Wrapping a function inside of a struct.

them when `Truncate` is instantiated and the `()` operator is called on Line 10 of Figure 3.6.

We can simplify this example further and deduce `FN` by passing it as an argument to `apply`, as shown in Figure 3.7. In Figure 3.7, `Truncate` is defined and instantiated as the variable `truncate`. The variable `truncate` is passed as a function argument to `apply`. Passing `truncate` as an argument allows the compiler to infer the template parameter `FN`. Because the variable `__ignored` is never used the example in Figure 3.7 is synthesizable. However, we still cannot infer `TO` unless it is passed as a function argument. To deduce `TO` we must use a new feature from the C++ specification covered in Section 3.2.1.

Type Inference

Section 3.2.1 we showed that we can automatically infer input types using template inference, but could not infer output types since output types are not arguments. Higher-order functions like `map` can return a multitude of types depending on what function is provided. To correctly mimic the behavior of dynamically-typed software languages we must be able to infer the output types automatically.

Figure 3.8 demonstrates how we can remove the `TO` template parameter from `apply`

```
struct Truncate{
  template <typename TI>
  unsigned char operator()(TI IN){
    return (unsigned char)IN;
  }
} truncate;

template <typename TO, typename TI, class FN>
TO apply(FN __ignored, TI IN){
  return FN()(IN);
}

int main(){
  int i = 0x11223344;
  unsigned char res;
  // Prev: apply<unsigned char truncate<int> >(i);
  res = apply<unsigned char>(truncate, i);
  // res = 0x44
  return 0;
}
```

Figure 3.7. Class-wrapped-functions can be inferred by passing them as instances at the callsites.

using the new `auto` keyword. This causes the compiler to deduce the return value from the type of `FN`. Figure 3.8 is functionally identical to Figure 3.7, but the call to `apply` is now less verbose.

```
struct Truncate{
  template <typename TI>
  unsigned char operator()(TI IN){
    return (unsigned char)IN;
  }
} truncate;

// Previously: <typename TO, typename TI, class fn>
template <class fn, typename TI>
// Previously: TO apply(fn _, TI IN)
auto apply(fn _, TI IN){
  return fn()(IN);
}

int main(){
  int i = 0x11223344;
  unsigned char res;
  // Previously: apply<unsigned char>(truncate, i);
  res = apply(truncate, i);
  // res = 0x44
  return 0;
}
```

Figure 3.8. Applying the new `auto` keyword to Figure 3.7 allows us to remove the template parameter `TO`

```
int main(){
  // Constructing three arrays
  array<int, 10> arr = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  array<int, 10> arr1 = range<10, 0, 1>(); // {0...9}
  array<int, 2>  arr2 = construct(1, 2); // {1, 2}

  // Manipulating arrays
  int h = head(arr); // 0
  array<int, 9> t = tail(arr); // {1...9}
  array<int, 10> arr3 = prepend(h, t); // {0...9}
  array<int, 10> arr4 = append(t, h) // {1..9, 0}
  array<int, 20> arr5 = concatenate(arr, arr); // {0...9, 0...9}
  return;
}
```

Figure 3.9. A variety of array constructors and manipulations.

3.2.2 Arrays

Lists are the most common target of higher order functions. In software productivity languages lists are implemented as dynamically allocated chunks of memory (contiguous or linked) that can be created, duplicated, resized, and modified during runtime. Lists in hardware circuits describe static structures that cannot be dynamically modified. To describe these structures in C/C++ we must use static arrays.

For our higher-order functions we use the `array` class from the C++ Standard Template Library to provide list-like functionality for our higher-order functions. The `array` class has major benefits over pointers. Unlike pointers an `array` is parameterized by its length, and propagates this parameter through function arguments and for template inference. Second, `array` is a class with a copy operator. This means it can be returned from a function directly, unlike pointers, which must be passed as an argument to be modified in order to be "returned". This maintains a more Python-like feel for our functions.

Figure 3.9 shows severals examples of how C++ `array` objects are constructed and can be manipulated. While these arrays cannot be dynamically resized, our library also provides a collection of functions for static list manipulation. A few simple examples of list manipulations are shown in Figure 3.9. Thus, the `array` class allows us to provide software-like syntax for C++ hardware tools.

3.2.3 Recursion and Looping

Higher-order functions use loops or recursion to iterate over list elements. Dynamically typed languages like Python can use loops to implement recursion since intermediate type state is propagated during runtime. Statically typed languages like C++ must use recursion since the output of the previous iteration must type-check with the current at compile time. Since C++ hardware development tools are statically typed and no dynamic stack, we must use static recursion.

```
int main(){
  array<int, 10> arr = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  int s = sum(arr); // s = 45
  return 0;
}
```

Figure 3.10. Using the recursive array-sum from Figure 3.11

C++ static recursion uses a technique known as *template metaprogramming* [57]. Template metaprogramming is synthesizable because its is unrolled at compile time, bounded by compile-time template parameters, it eliminates the need for a dynamic stack. Template metaprogramming makes the resulting functions concise. This is shown in Figure 3.10, which calls `sum` to obtain the sum of an array of elements, and `sum`'s implementation in Figure 3.11.

```
template <size_t LEN>
struct _sum{
  auto operator()(array<int, LEN> IN){
    return head(IN) + _sum<LEN-1>()(tail(IN));
  }
};

template <>
struct _sum<0>{
  int operator()(array<int, 0> IN){
    return 0;
  }
};

template <size_t LEN>
int sum(array<int, LEN> IN){
  return _sum<LEN>()(IN);
}
```

Figure 3.11. An array-summation implementation using recursive class templates.

Figure 3.11 shows an implementation of sum that uses template recursion to iterate through the array. Lines 15-18 define the sum method with the template parameter LEN. This is preceded by two definitions of the _sum helper class. The first definition on Lines 1-6 is the recursive definition that is used when the template parameter LEN is non-zero. The second definition on Lines 8-13 is the base case for when LEN is zero. Together these implement the sum method.

When the sum method is called in Figure 3.11 the function creates an instance of _sum<LEN> and calls its () operator. The () operator instantiates an instance of _sum<LEN-1> and calls _sum<LEN-1>'s () operator. This process continues until LEN is equal to 0 and _sum<0> return 0. When the program is run in software the program unwinds the call tree and adds the elements together. Since LEN is a static template parameter this recursion is bounded at compile time and can be synthesized.

3.2.4 Higher-Order Functions

```
1  template <size_t LEN>
2  struct _rhelp{
3    template<typename TI, typename TA, class FN>
4    auto operator()(FN F, TI INIT, array<TA, LEN> IN){
5      return _rhelp<LEN-1>()(F, FN()(INIT, head(IN)), tail(IN));
6    }
7  };
8
9  template <>
10 struct _rhelp<0>{
11   template<typename TI, typename TA, class FN>
12   TI operator()(FN F, TI INIT, array<TA, 0> IN){
13     return INIT;
14   }
15 };
16
17 template <size_t LEN, typename TI, typename TA, class FN>
18 auto reduce(FN F, TI INIT, array<TA, LEN> IN){
19   return _rhelp<LEN>()(F, INIT, IN);
20 }
```

Figure 3.12. Implementation of the function reduce using all of the features described in this section.

We now have all of the pieces to develop our synthesizable higher-order functions: templates, arrays, functions, and recursion. We emphasize that the implementations of our

higher-order functions are complex but that using our functions is quite simple as demonstrated in these examples.

We demonstrate our techniques by implementing the higher-order function `reduce` in Figure 3.12 and follow with an example in Figure 3.13. `reduce` is defined on Lines 17-20. When the function `reduce` is called templates are inferred as described in Section 3.2.1. The template parameter `LEN` specifies the length of the array, parameters `TI` and `TA` provide input polymorphism on the initial value and the array value respectively, and `FN` is the function class from Section 3.2.1. The output type is deduced by the `auto` keyword. `LEN` parameterizes the recursive class `_rhelp` defined on Lines 1-7. The base case when `LEN` is zero is defined on Line 9-15. The recursive behavior follows the description in Section 3.2.3.

Figure 3.13 shows how the array summation function from Figures 3.10 and 3.11 can be re-written using `reduce`. Again, this demonstrates that using our functions is quite simple despite the implementation complexity.

```
1  int main(){
2    array<int, 10> arr = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
3    int s = reduce(add, arr, 0); // s = 45
4    return 0;
5  }
```

Figure 3.13. Array summation from Figure 3.10 re-written using `reduce` defined in Figure 3.12

3.3 Examples

We demonstrate our work by implementing the Fast Fourier Transform (FFT) algorithm with our higher-order function library. We use this example to demonstrate our library and compare its syntax to Python, a modern, dynamically-typed productivity language. We have chosen FFT because it uses many of our higher order functions, is a well-known algorithm in the hardware development community, and has been used as a motivating related hardware development language work, [56, 58]. Further examples are available in our repository, and results are shown in Section 3.4.

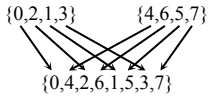
Table 3.1. Summary of the list-manipulation and higher-order functions used in this paper. `FN` is shorthand for a wrapped function from Section 3.2.4

List Function	Description
`array<TA, 2> construct (TA, TA)`	Turn two elements into a 2-element array
`array<TA, LEN + 1> prepend (TA, array<TA, LEN>)`	Add an element to the head of an array
`array<TA, LEN + 1> append (array<TA, LEN>, TA)`	Add an element to end of an array
`array<TA, LENA + LENB> concatenate (array<TA, LENA>, array<TA, LENB>)`	Concatenate two lists into a single list
`TA head(array<TA, LEN>)`	Get the first element (head) of an array
`array<TA, LEN-1> tail (array<TA, LEN>)`	Get a list with all elements except the head (tail)
`array<pair<TA, TB>, LEN> zip (array<TA, LEN>, array<TB, LEN>)`	Combine two lists into a list of pairs
`pair<array<TA, LEN>, array<TB, LEN> unzip (array<pair<TA, TB>, L)`	Split list of pairs into a pair of two lists
Higher-Order Function	**Description**
`auto flip (FN)`	Return a function with the order of its input arguments swapped
`auto compose (FNA, FNB)`	Return a function where FNA is called with the output of FNB
`auto map (FN, array<TA, LEN>)`	Apply a function to each element in a list
`auto reduce (FN, array<TA, LEN>, TI)`	Iterate from left to right applying FN and carrying the result
`auto rreduce (FN, array<TA, LEN>, TI`	Iterate from right to left applying FN and carrying the result
`auto divconq (FN, array<TA, clog2(LOGLEN)>)`	Divide a list into single elements and apply a function to pairs
`auto zipWith (FN, array<TA, LEN>, array<TB, LEN>)`	Combine two lists with a function

The FFT algorithm is developed in several parts. Section 3.3.1 demonstrates `interleave`, which is used in the `bitreverse` function in Section 3.3.2. Section 3.3.3 demonstrates how to implement an N-point FFT Stage function. Section 3.3.4 combines the previous sections into an implementation of the FFT algorithm.

3.3.1 Interleave

The `interleave` function interleaves two lists as shown in Figure 3.14. Figure 3.14a shows a graphical example of interleaving two lists. Figure 3.14b demonstrates a C++ implementation using our synthesizable library, and Figure 3.14c demonstrates a Python implementation.



(a)

```
struct Interleave{
  template <typename T, size_t LEN>
  auto operator()(array<T, LEN> L, array<T, LEN> R){
    auto pairs = zipWith(construct, L, R);
    return rreduce(concatenate, pairs, array<T, 0>());
  }
} interleave;
```

(b)

```
def interleave(L,R):
    pairs = zip(L, R)
    concatenate = lambda p, lst: list(p) + lst
    return rreduce(concatenate, pairs, [])
```

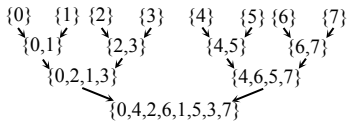
(c)

Figure 3.14. (a) Interleaving two lists graphically (b) Interleaving two lists in C++ (c) Interleaving two lists in Python

Figure 3.14b uses `zipWith` to apply the `construct` function to combine both arrays into a pair-wise array of arrays. Then, `rreduce` applies the `merge` function to attach the front of each 2-element array to the end of the previous array and produce an interleaving. The corresponding Python implementation is shown in Figure 3.14c, with `zip` instead of `zipWith` because Python tuples are easily converted to arrays.

3.3.2 Bit-Reverse

Figure 3.15 shows a bit-reverse permutation for arrays. In the permutation, the element at index N is swapped with the value at P, where P is equal to the a reversal of the bits of N. For example in an 8-element list, if $N = 1 = 3'b001$, then $P = 4 = 3'b100$. Figure 3.15a shows a bit-reversal permutation applied to the list $\{0,1,2,3,4,5,6,7\}$ as a recursive interleaving. This is followed by the synthesizable C++ implementation in Figure 3.15b and the Python implementation in Figure 3.15c. Figure 3.15b implements the bit-reverse permutation using



(a)

```
struct Bitreverse{
  template <typename T, size_t LEN>
  auto operator()(array<T, LEN> IN){
    return divconq(interleave, IN);
  }
} bitreverse;
```

(b)

```
def bitreverse(in):
    return divconq(interleave, in)
```

(c)

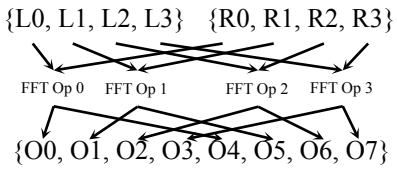
Figure 3.15. (a) Bit-reverse permutation of a list graphically (b) Bit-reverse in C++ (c) Bit-reverse in Python

the higher-order functions we have developed. `divconq` from Table 3.1 is used to divide the input array into single-element arrays. The function `interleave` from Section 3.3.1 is used to interleave the resulting arrays to produce the result. A Python implementation is in Figure 3.15c.

3.3.3 N-Point FFT Stage

The FFT algorithm is implemented by recursively applying an N-Point FFT function to two outputs of two N/2-Point functions in the previous stage. The N-point FFT stage is shown graphically in Figure 3.16a. In an N-point FFT inputs from the “left” and “right” inputs are passed

with the context (tree level and index) to the `fftOp` function. `fftOp` performs computation and then produces two outputs that are de-interleaved in a similar fashion.



(a)

```
struct NPtFFTStage{
  template <typename T, size_t LEN>
  auto operator()(array<T, LEN> l, array<T, LEN> r){
    static const std::size_t LEV = log2(LEN) + 1;
    auto contexts = zip(replicate<LEN>(LEV), range<LEN>());
    auto inputs = zip(l, r);
    auto outpairs = zipWith(fftOp, contexts, inputs)
    auto outputs = unzip(outpairs);
    return concatenate(outputs.first, outputs.second);
  }
} nPtFFTStage;
```

(b)

```
def nptfftstage(L, R):
    lev = log2(len(L) + 1)
    contexts = zip([lev]*L, range(len(L)))
    inputs = zip(L, R)
    outpairs = map(fftOp, zip(contexts, inputs))
    outputs = zip(*outpairs)  # Unzip
    return outputs[0] + outputs[1]
```

(c)

Figure 3.16. (b) N-Point FFT Stage using synthesizable higher-order functions (c) N-Point FFT Stage using Python

We define an `nPtFFTStage` function in Figure 3.16b. This function first computes its level (`LEV`) using the `LEN` template parameter. `LEV` is replicated and paired with an index to produce a context for each `fftOp` function. The function calls the `fftOp` function using `zipWith` to pass the context and input data. The output is de-interleaved using `unzip` and `merge`.

3.3.4 Fast Fourier Transform

We now have all the tools we need to implement widely-used Fast Fourier Transform (FFT) algorithm from Figure 3.2. The divide and conquer, recursive structure of the FFT

algorithm is shown in Figure 3.2a. Each N-point stage of an FFT combines the results of two N/2-point transforms.

The FFT algorithm is implemented in Figure 3.17 using the higher-order function `divconq` with the `nPtFFTStage` function. The input list is bit-reversed to obtain the correct output ordering.

```
1 template <typename T, size_t LEN>
2 auto fft(array<T, LEN> sig){
3   return divconq(nPtFFTStage, bitreverse(sig));
4 }
```

Figure 3.17. FFT Implementation using the functions described in this section

As demonstrated, we have created a library of higher-order functions with a syntax that is similar to a modern productivity language, Python. This is evident from comparing examples in Figures 3.1, 3.2, 3.14, 3.15, 3.16, and 3.17. In all examples, the syntax and structure of the C++ code is very similar to Python. This is in spite of extensive use of C++ templates. More examples can be found in our repository, and in Section 3.4.

3.4 Results

We report the quantitative results of our work by synthesizing six application kernels with our higher-order functions and compare them to loop-based implementations. The kernels we target are described in Section 3.4.1, followed by the experimental setup in Section 3.4.2 and results in Section 3.4.3.

3.4.1 Application Kernels

Fast Fourier Transform

The *Fast Fourier Transform (FFT)* kernel was presented in Section 3.3.4. The FFT kernel is widely used in signal analysis, often for compression. As demonstrated in Section 3.3, the FFT kernel can be implemented with the higher-order `divconq` and `zipWith` functions. The loop-based equivalent is written with a pair of nested loops.

Argmin

Argmin is a function to compute value and index of the minimum element in an array. This function can be used in many streaming data processing applications such as databases [50, 53]. The *Argmin* kernel can be written using the higher-order function `divconq` and function `argminop` as an input. The `argminop` function returns the value and index of the minimum value from its subtrees. The loop-based equivalent is implemented with a pair of nested loops.

Finite Impulse Response Filter

A Finite Impulse Response (FIR) filter is a common signal processing kernel that takes samples from an input signal and performs a dot-product operation with an array of coefficients. These filters can be used for simple high-pass, and low-pass audio filtering. Our FIR filter is composed of two `zipWith` to perform the element-wise multiplication of the input signal array and coefficient array, and `divconq` to produce an addition tree to take the sum of all of the elements. The loop-based equivalent is written as a single for-loop that computes the element-wise multiplication, and a pair of nested for loops to implement the addition tree.

Insertion Sort

Insertion Sort is a streaming function that computes sorted lists. In our kernel, an array of values is streamed into the kernel. The kernel maintains a sorted order for the N minimum or maximum values seen and ejects others. This is identical to the implementation in [54]. The *Insertion Sort* kernel can be written using the higher-order function `reduce`, and a `compareswap` function. The `compareswap` function is applied to each element in the list and swaps the element depending on the sort criteria. The ejected element is carried forward for further comparison.

Bitonic Sort

Bitonic sort is a highly parallel sorting function used widely on GPU and FPGA architectures [54]. It's structure can be described recursively as a tree of butterfly networks.

Smith-Waterman

A systolic array is a common hardware structure for solving dynamic programming problems. In this example we use the Smith-Waterman algorithm from [55]. The systolic array is written as a composition of `zipWith` and `reduce`.

3.4.2 Experimental Setup

For each of the six algorithms described in Section 3.4.1 we developed a loop-based and higher-order-function-based implementation, resulting in twelve designs. Each design targeted 16-element arrays, with types described in Table 3.2. For each design we gathered performance, resource utilization, and maximum frequency results for loop-based and higher-order-function-based variants of the six algorithms. Our results were gathered in Vivado 2017.4 and implemented on a PYNQ development board with a Zynq XC7Z020 SoC.

Performance results were gathered from the Vivado HLS synthesis tool. The tool targeted a clock period of 2 nanoseconds (500 MHz) to guarantee that the architecture was fully pipelined and completely unrolled. These results are reported in Table 3.3.

Table 3.2. Post-Implementation resource utilization for six algorithms on a PYNQ (XC7Z020) in Vivado 2017.4.

Function Name (`Data Type`)	Higher-Order Functions					Loop Based				
	FF	SRL	LUT	BRAM	DSP	FF	SRL	LUT	BRAM	DSP
FFT (`ap_fixed<32,16>`)	21263	2487	8096	0	77	21240	2494	8096	0	77
Argmin (`int`)	2670	8	1573	0	0	2666	10	1575	0	0
FIR Filter (`float`)	14388	277	7306	0	48	14388	272	7305	0	48
Insertion Sort (`int`)	2300	0	935	0	0	2300	0	935	0	0
Bitonic Sort (`int`)	11929	1	4869	0	0	11929	1	4869	0	0
Smith-Waterman (`ap_int<2>`)	895	11	1187	0	0	895	11	1186	0	0

Resource utilization and F_{max} results are reported from a sweep of thirteen Vivado Implementation goals. Resource utilization did not vary across these thirteen runs and are reported in Table 3.2. For each goal we performed a binary search for the maximum frequency, varying the output frequency of the Xilinx Clock Wizard attached to the hardware core. The resulting statistics are reported in Table 3.4. Finally, Table 3.5 presents a statistical analysis of the maximum frequency data we collected.

3.4.3 Analysis

Performance results are shown in Table 3.3. Column 1 displays the name for each of the six application kernels. Columns 2 and 3 show the initiation interval and latency for each higher-order-function-based application kernel. Likewise, columns 4 and 5 show the initiation interval and latency for each loop-based application kernel.

Table 3.3. Performance results from Vivado HLS 2017.4 for six application kernels on 16-element lists

	HOFs		Loop-Based	
Function Name	Interval (Cycles)	Latency (Cycles)	Interval (Cycles)	Latency (Cycles)
FFT	1	59	1	59
Argmin	1	7	1	7
FIR Filter	1	65	1	65
Insertion Sort	1	31	1	31
Bitonic Sort	1	21	1	21
Smith-Waterman	1	16	1	16

From Table 3.3 we conclude that our higher-order functions produce equal performance to fully-unrolled loop-based designs. This is evident from comparing the initiation intervals of columns 2 and 4 and the latencies of columns 3 and 5 in Table 3.3. For all designs, higher-order implementations are equal to fully-unrolled loop-based implementations. We conclude there are no performance penalties associated with our higher-order functions.

Post-Implementation resource utilization is shown in Table 3.2. Columns 2-6 in Table 3.2 show resource utilization for applications written with our higher-order functions and columns 7-11 show resource utilization for applications written with fully-unrolled loops.

From Table 3.2 we conclude that our higher-order functions implementations consume similar resources to loop-based implementations. Higher-order function and loop-based implementations consume equal numbers of DSPs and no BRAMs. The two methodologies also consume similar numbers of Flip-Flops (FFs), Look-Up-Tables (LUTs), and Shift-Register-Look-up-tables (SRLs): The maximum resource difference between the two implementation methodologies is less than 10 resources, a less than 1% difference in most cases. Given these small differences, we conclude that our functions do not produce significant differences in resources consumed.

We have a theory for this behavior. Vivado HLS and SDSoC use an LLVM backend to generate verilog circuits. This verilog is emitted from LLVM IR. We theorize that this behavior is because LLVM IR produced by our higher-order functions is identical to the LLVM IR produced by loop-based designs after code transformations have been performed. However, the designs above generate large LLVM IR files, and it is difficult to differentiate the structures. However, results shown here are consistent with this theory.

Post-Implementation frequency statistics are shown in Table 3.4. The statistics in Table 3.4 are gathered from a sweep of 13 Vivado Implementation goals. Columns 2-4 show the mean, median, and standard deviation of frequency results for application kernels implemented with higher-order functions, and columns 5-7 show corresponding statistics for loop-based designs.

To determine whether the average maximum frequency of higher-order-function-based kernels differs statistically from loop-based kernels we perform an exact permutation test with the null hypothesis of equal means [59, 60]. In considering whether to reject the null hypothesis, we adjust for multiple comparisons using Holm-Bonferroni (H-B) correction, this is necessary because we test each of the six designs independently. The resulting p-values and $\alpha = 0.05$ H-B

Table 3.4. Maximum frequency statistics from 13 implementation runs of Vivado 2017.4

	HOFs F_{max} (MHz)			Loop Based F_{max} (MHz)		
Function Name	Mean	Median	Std.	Mean	Median	Std.
FFT	123.56	124.22	3.35	123.56	123.44	3.18
Argmin	110.64	110.94	2.51	110.91	109.77	2.89
FIR Filter	166.80	165.63	3.12	165.56	166.41	3.57
Insertion Sort	162.83	162.11	3.22	166.47	164.84	5.02
Bitonic Sort	112.77	112.11	3.50	113.85	115.63	2.76
Smith-Waterman	103.73	104.69	4.19	103.58	105.47	5.34

rejection thresholds are reported in Table 3.5.

Table 3.5. *p*-values and H-B rejection thresholds from a permutation test with the null hypothesis of equal means

Function Name	*p*-value	$\alpha = 0.05$ thresh.
FFT Algorithm	0.97656	0.05
Argmin	0.74976	0.01667
FIR Filter	0.22267	0.01
Insertion Sort	0.00830	0.00833
Bitonic Sort	0.23193	0.0125
Smith-Waterman	0.91406	0.025

Out of the six designs, only one (Insertion Sort) would be rejected at the $\alpha = 0.05$ level, and only just barely. Given the results of the Table 3.5 analysis, we conclude that our functions produce maximum frequency results that are generally statistically indistinguishable from those of loop-based designs.

3.5 Related Work

3.5.1 Hardware Development Languages

There have been several hardware-development projects that bring functional languages to bring higher-order functions to hardware development. Lava [56] and Cλash [58] are functional

hardware development languages embedded in Haskell that provide higher-order functions and polymorphism to users. Lava is interesting because the operators are composed from functional definitions of Xilinx primitives, which provides a function abstraction for the user and context for the compiler to improve synthesis.

Higher-order functions originate from, but are not limited to purely-functional languages. The Chisel project [16] uses Scala and provides higher-order functions. Several projects have used Python for hardware development, for example, PyMTL [17] is a project that embeds a hardware development language in Python to raise the level of abstraction. These projects provide higher-order functions, imperative syntax, and polymorphism to generate circuits.

However, HDL projects fail to raise the designer's level of abstraction. The notion of wiring together operations, scheduling, registers, and clocks is pervasive. These concepts are unfamiliar to software developers. In addition, HDL languages do not generate complete systems. C/C++ synthesis tools completely abstract the detailed wiring, scheduling, and clocking concepts and automate core integration with communication and memory interfaces [29, 25, 28].

3.5.2 High-Level Synthesis Languages

High-level synthesis tools were developed to eliminate scheduling, wires, and registers from designer control - but few support higher-order functions. The Bluespec [61] language is one tool that provides higher-order functions. Bluespec is written written as a set of rules that are executed when prerequisites are met. These rules are scheduled by the Bluespec compiler to create a Verilog circuit.

Despite its obvious advantages the syntax and structure of the Bluespec language is substantially different than modern software languages. Our work provides a syntax that is similar to modern software languages and still provides higher-order functions, and automatic scheduling.

3.5.3 Domain-Specific Languages

In [52] the authors develop a domain-specific language with higher-order functions to generate parallel hardware. This domain-specific language is scheduled, translated into Maxeler's vendor-specific dataflow hardware development language, and finally deployed onto the vendor system. By using higher-order functions the authors can deploy highly-parallel systems, with low verbosity and high productivity for software engineers.

Our work does not use a domain specific language. Instead, we provide a familiar software API within C++ synthesis tools. By targeting the C++ compiler toolchain we can rely on a large body of existing work on optimization passes to improve our quality of result.

In addition, papers by [52] and [51] highly complementary to our own. In [52] the authors state: "generating imperative code from a functional language only to have the HLS tool attempt to re-infer a functional representation of the program is a suboptimal solution because higher-level semantic knowledge in the original program is easily lost." Similarly, [51] motivates the need for parallel patterns in C++. We believe our work is a basis for both of these works. We have generated higher-order function interfaces for C++ synthesis, eliminating the need to "re-infer a functional representation".

3.5.4 Our Work

In our work we develop a library of higher-order functions for C/C++ synthesis tools. Using C/C++ synthesis tools avoids the pitfalls of HDLs: low-level wiring, registers, scheduling, and interfaces. Unlike prior work in high-level synthesis, our work is synthesizable to hardware and available in standard tools tools without modifications. Finally, it provides a syntax similar to a modern dynamically typed productivity language within C++.

3.6 Conclusion

In this work, we have demonstrated a library of synthesizable library of higher-order functions for C++. These functions mimic the syntax of a modern dynamically-typed productivity language despite being written in a statically-typed language, for a tool with limited memory primitives.

We demonstrated how we build our higher order functions using C++ templates, and new features in the C++ standard. The library we created uses extensive C++ templates but the API we produced is simple and has similar syntax to Python.

Our results demonstrate that our code generates highly-similar hardware to traditional loop-based high-level synthesis: performance results were equal, differences in resources consumed were small, and the distributions of the maximum frequencies were generally statistically indistinguishable.

There are challenges ahead for this work: First, defining functions is more verbose than in other languages. Second, our work currently instantiates completely parallel computation kernels. Further work is needed to break these kernels into iterative sub-problems and provide a trade-off between performance and area.

In summary, we have made measure steps toward increasing the accessibility of C++ synthesis tools by providing common abstractions that are present in software development environments.

3.7 Acknolwedgements

The authors acknowledge the National Science Foundation Graduate Research Fellowship Program, the Powell Foundation, and ARCS foundation for their support.

This chapter, in full, is a reprint of the material as it appears in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 2018. Richmond, Dustin; Althoff, Alric; and Kastner, Ryan. The dissertation author was the primary investigator and author of this

paper.

Chapter 4

A Lab Curriculum for Hardware/Software Co-Design

4.1 Introduction

PYNQ is a fantastic tool for embedded education. PYNQ boards have low power ARM processors and a small form factor with extensive Python libraries. Existing PYNQ Overlays—FPGA designs with post-bitstream flexibility—provide common communcation interfaces and are delivered with self-documenting Jupyter notebooks. For these reasons, PYNQ is being rapidly adopted as a teaching platform across several universities, Tampere University of Technology, BYU [40], and our own. PYNQ is a natural fit for our classes focus on Hardware/Software (HW/SW) co-design of wireless embedded systems. It provides easy-to-use drivers from a high-level language in an embedded form-factor. With High-Level Synthesis (HLS) and the Vivado block diagram editor PYNQ allows us to avoid low-level tedium and focus on concepts and algorithm development.

However, despite extensive PYNQ API documentation, we have found that there is little instructional materials for building complete PYNQ Overlays. A "complete PYNQ Overlay" comprises a swath of interdependent topics: The Vivado block diagram editor, IP and algorithm development, hardware interfaces, physical memory, drivers, and Python packagaing. This has been a conscious (and reasonable) design choice of the PYNQ team to focus on a well-built set of libraries and APIs to aid all-programmable development. Creating and maintaining a complete

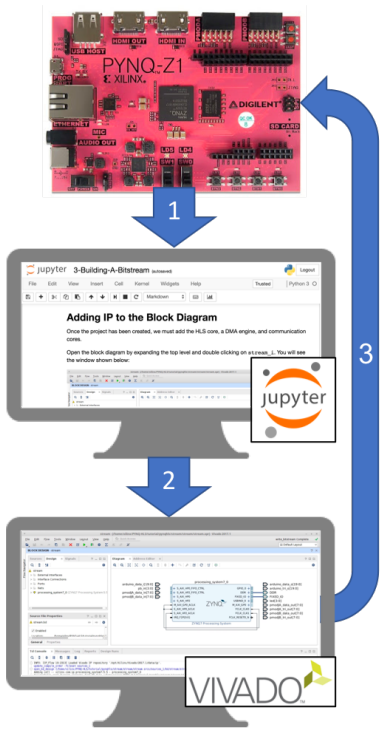


Figure 4.1. High-level flow for our PYNQ labs: Our labs are installed on the PYNQ board and accessed from a host computer (1). Jupyter Notebooks direct the reader through the steps of creating a PYNQ Overlay (2). When the overlay is complete, the PYNQ Overlay is installed and run on the PYNQ board (3).

set of tutorials is time consuming and would take time away from development. Instead, the task of developing tutorials and learning materials has been given to the community.

We have created a lab curriculum that teaches HW/SW co-design through building complete PYNQ Overlays. Our lab curriculum is split into three units where each unit is a set of independent Jupyter Notebooks teaching a concept. The flow of each unit is shown in Figure 4.1: Jupyter Notebooks served by the PYNQ board (1) and accessible from a host web browser. The notebooks instruct readers how to create HLS Cores, Vivado Designs, and Python Classes (2). These parts are packaged as a PYNQ Overlay and deployed pback onto the PYNQ board for testing (3). At the end of each unit the readers have created a working PYNQ Overlay.

The notebooks that comprise each unit are our major contribution and are available on GitHub. We also highlight three contributions that will be useful to the broader embedded

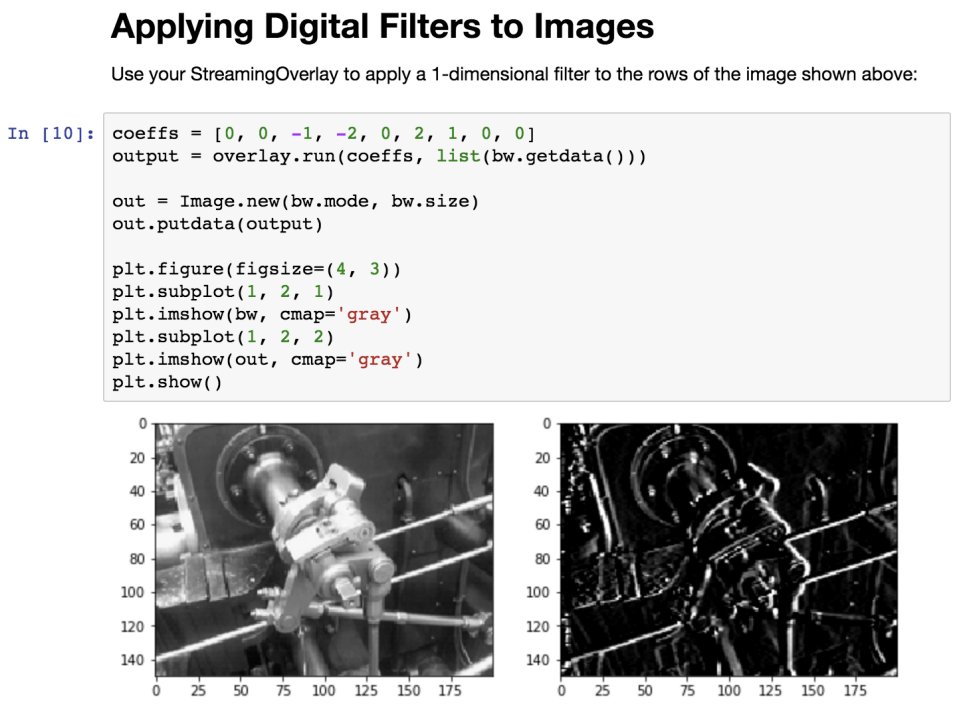


Figure 4.2. A cell from the demo notebook showing how the 9-tap 1-dimensional filter can be applied to images.

development community:

1. An example of Jupyter-based instructional materials
2. Re-usable skeleton files for starting PYNQ Projects
3. Instructions for Pip-packing PYNQ Overlays

The remainder of our paper is organized as follows: Section 4.2 describes the three units we have created and the common hardware/software codesign patterns they target. Section 4.3 describes the 5-part structure of each lab. Section 4.4 discusses related work before concluding in Section 4.5.

4.2 Units

Our curriculum is dividied into three units that can be completed in a three-hour lab session. Each unit teaches readers how to implement a single High-Level Synthesis (HLS) core and PYNQ Overlay for a common computational pattern: Real-Time and Memory-Mapped IO, Streaming Communication, and Shared Memory Communication. We chose these topics because they represent the topics we found most common in our classes and in embedded projects.

Each unit is split into five interactive Jupyter Notebooks described in Section 4.3. Figure 4.1 shows how these notebooks are served by a PYNQ board for a web broswer on the reader's host computer. The reader follows the notebooks to implement a Vivado block diagram that is then loaded onto the FPGA fabric of the PYNQ board. The final steps in each unit interact with the overlay the reader has created.

In the process of completing these notebooks readers are will learn about a diverse set of topics: virtual memory, direct-memory access, interrupts, memory-mapped IO, and physical IO, deterministic runtime, and embedded communication.

The high-level learning objectives for our work aree:

1. Teach readers how to create the components of a PYNQ Overlay
2. Interact with an Hardware Core from Python using PYNQ
3. Instruct readers about the different communication types
4. Demonstrate how to package a PYNQ Overlay for distribution

4.2.1 Real-Time IO Unit

In the Real-Time IO unit we describe how to implement a Real-Time control loop in High-Level Synthesis. The concepts demonstrated in the Real-Time IO unit are: memory-mapped IO, physical IO, deterministic runtime, interrupts, and embedded communication (UART). These

are common computaional patterns in many embedded systems: low-latency motor control [38], industrial circuits [62], sensor fusion, and other applications [63].

The notebooks in this set teach the reder how to implement a HLS core for communicating with sensors. The core contains an AXI-Master interface that it uses to write to an AXI-Slave UART controller connected to PMOD_A on the PYNQ-Z1. Second, the loop flashes LEDs attached to the IO Pins. Third, it reads the buttons on the PYNQ board and writes them to a register location. The HLS Core can operate synchronously, where it executes once and returnsm or asynchronously where it detatches and loops forever. In asynchronous mode, the HLS core runs executes every 50K clock cycles. We chose these examples because they are easy to demonstrate and broadly applicable.

The learning objectives for this unit are:

1. Create an interface for physical input and output in HLS
2. Write an HLS core with a dictated runtime
3. Interact with AXI communication peripherals in HLS

4.2.2 Streaming Communication Unit

The Streaming Communication Unit teaches the reader how to implement a PYNQ Overlay containing an High-Level Synthesis (HLS) core with AXI-Stream interfaces. This is a common computational pattern for high-bandwidth video [39, 27] and audio [37] applications. The concepts demonstrated in this unit are: Memory-Mapped IO and Direct-Memory Access (DMA).

The notebooks in this set teach the reader how to implement a 9-tap, 1-Dimensional digital filter witten in HLS. The filter core has an AXI-Stream input and AXI-Stream output and both interfaces are connected to a Xilinx DMA Engine. The signal length and tap coefficients are configurable. The core processes AXI-Stream beats until length is reached and then terminates. Figure 4.2.2 demonstrates an execution of this overlay from Jupyter Notebooks.

The learning objectives for this unit are:

1. Create a HLS core with an AXI-Stream interface
2. Use a DMA Engine to move high-bandwidth data
3. Learn how to set registers within an HLS Core

4.2.3 Shared Memory Communication

The Shared Memory Communication unit teaches the reader how to implement a matrix-multiply core operates on data in a memory space shared between the ARM PS and the accelerator. This is a common computation pattern in image processing and video applications, particularly those with non-streaming, repetitive, or non-linear access patterns [39]. The concepts demonstrated in the Shared Memory unit are: shared memory, interrupts, and HLS library re-use.

The notebooks in this unit teach the reader how to implement a simple matrix multiplication core in the FPGA fabric. Further steps demonstrate how to re-use a highly optimized HLS matrix mutliplication core provided by Xilinx.

The learning objectives for this unit are:

1. Create a HLS core with an AXI-Master interface
2. Re-use highly-optimized HLS libraries for computation
3. How to use interrupts to signal the ARM Processing System

4.3 Unit Structure

Each unit in our work teaches how to build a complete PYNQ Overlay: a Block Diagram Script (.tcl file), FPGA Bitstream (.bit file), and Python Class, as shown in Figure 4.3. The bitstream and .tcl are generated by the Vivado Block Diagram Editor, while the Python class is written by the user. A PYNQ Overlay is loaded by instantiating an instance of the Python class.

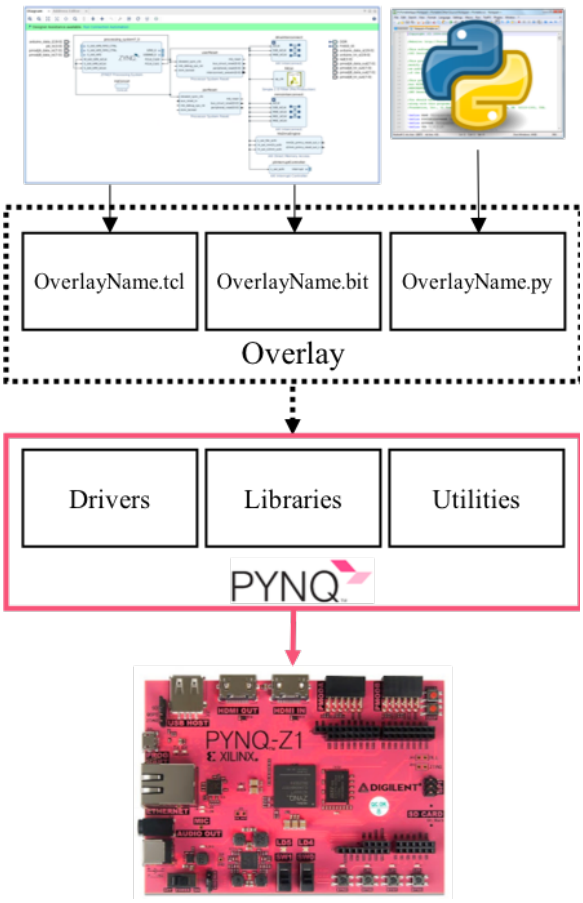


Figure 4.3. The PYNQ Flow. The Vivado Block Diagram editor is used to generate a bitstream and a TCL script. In parallel, a user generates a Python Class representing their overlay. When the Overlay is loaded, PYNQ uses the TCL script to discover hardware IP in the bitstream and load drivers. PYNQ also provides utilities to manage the FPGA.

Underlying PYNQ libraries parse the .tcl file and instantiate driver classes for hardware IP in the bitstream.

Every unit is designed to lead users through this flow using a set of six Jupyter Notebooks: A demonstration notebook, and five instructional notebooks. The demonstration notebook shows the final result using the verified overlay. The five instructional notebooks are organized as follows:

1. Introduction

2. Creating a Vivado HLS IP Core

3. Building a Bitstream

4. Overlays and Bitstreams in PYNQ

5. Packaging an Overlay for Distribution

The tutorials require a PYNQ board connected to a host machine running Windows or Linux with Vivado 2017.1 WebPack edition (free). We recommend an internet connection on the host, but it is not required if the packages can be pre-installed. An internet connection on the PYNQ board is not required, but is recommended.

One innovation in our work is providing operating system agnostic instructions using Cygwin. The reader is required to install Cygwin on Windows (with GNU Make, and Git). On Linux, a terminal emulator running Bash is required.

4.3.1 Demonstration Notebook and Verified Overlay

Each unit starts with a Demonstration Notebook, delivered with a verified pre-built overlay. Each notebook contains at least one example demonstrating the final product. The demonstration notebook for the Streaming Communication unit described in Section 4.2.2 shows how the 1-Dimensional filter can be applied to an image as shown in Figure 4.2, the Shared Memory demonstration notebook peforms a Matrix Multiply computation, and the Real-Time IO demonstration notebook interacts with LEDs, buttons, and peripherals.

The Verified Overlay is the source of known-working files for "skip cells" in the instructional notebooks. The instructional notebooks are described below.

4.3.2 Introduction

The Introduction notebook of each lab describes the concepts to be taught, sets up the reader's host environment, and verifies correct installation. The reader clones our repository on their PYNQ board using an internet connection or SAMBA. The notebook also highlights the

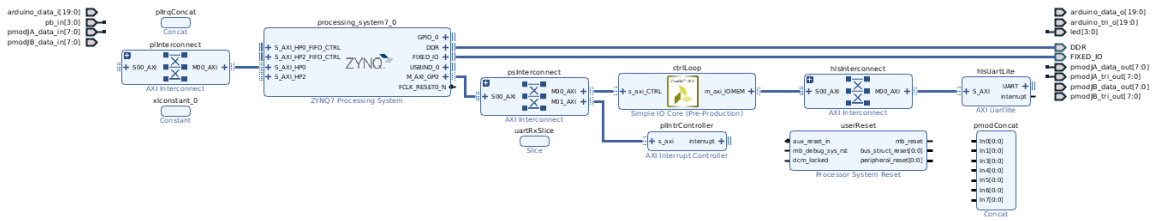


Figure 4.4. A Vivado Block Diagram from the Real-Time IO lab showing the AXI Interface connections. This figure demonstrates how we use *layers* to break down block diagram wiring into manageable steps.

typography that indicates commands to be run on the host versus commands to be run on the PYNQ board.

4.3.3 Creating a Vivado HLS IP Core

Creating a Vivado HLS IP Core leads the user through creating a Vivado HLS IP Core on their host machine. Our labs assume that the reader is already familiar with High-Level Synthesis (HLS) tools and design optimization.

Each unit provides a C/C++ skeleton, a testbench, and a makefile. Readers are required to provide a working C/C++ impelementation. The final step of this notebook runs the testbench and builds the final IP for the next notebook.

We assume that readers are already familiar with the Vivado HLS flow so the main goal of this notebook is to instruct readers how to create different interface types: Physical IO interfaces in the Real-Time IO lab, AXI-Stream interfaces in the Streaming lab, and AXI-Master interfaces in the Shared Memory lab. Our labs focus on the use of `HLS INTERFACE` pragmas and what interfaces can be created.

4.3.4 Building a Bitstream

The third notebook of each lab describes how construct an FPGA bitstream and .tcl script in Vivado. It covers how to create a Vivado block diagram, configure the block diagram IP, set the address map, export the block diagram .tcl file, and compile a bitstream. The .tcl and .bit file

generated in this step are critical components of a PYNQ Overlay as shown in Figure4.3.

Readers are provided with a makefile, a skeleton .tcl file, and verilog top level. These files create a Vivado project and block diagram on the host machine. The block diagram initially contains the correctly-configured ARM Processing System and top-level IO ports for the PYNQ-Z1 board. These files allow users to skip the tedious and error-prone steps of creating a Vivado project, creating a block diagram, instantiating and configuring the ARM processing system, creating a block diagram ports, and wiring the top level file.

Once the step above has been completed, the reader opens the Vivado block diagram. A complete block diagram for the Real-Time IO Overlay is shown in Figure 4.4. Pictures in the notebook demonstrate how to instantiate and configure and block diagram IP, and verify the correctness. The pictures show wiring *Layers*, to break down the wiring into steps: Clocks, Resets, GPIOs, and Interfaces. When complete, the reader exports the block diagram .tcl file and compiles the bitstream using the same makefile described above.

An innovation we provide is infrastructure that simplifies the block diagram export process. We provide a flow that uses unmodified .tcl files exported from the block diagram editor. Our flow sets custom IP paths, sets top-level files, and copies the bitstream to meet PYNQ requirements. As a result, unlike the current PYNQ flow, the .tcl files do not need to be modified after export from Vivado.

4.3.5 Overlays and Bitstreams in PYNQ

The penultimate notebook guides readers through the process of creating a complete PYNQ Overlay - manually loading a bitstream and .tcl file onto a PYNQ board, creating a Python Class that inherits from Overlay, and finally instantiating an Overlay object to interact with the bitstream. These steps are necessary for any PYNQ overlay.

In the first step, readers are directed to copy the outputs of *Building a Bitstream* onto their PYNQ board using the SAMBA. Notebook cells verify that this process has been completed correctly.

Next, readers are directed to create an python class that inherits from the PYNQ Overlay class to interact with their compiled bitstream. The notebook describes how the Overlay class parses the .tcl file and instantiates basic drivers for the HLS core. Once this is finished the notebook loads the and runs verification cells to confirm that the bitstream and Python class work correctly.

In this notebook we provide an innovation that allows users to specify relative paths. Non-PYNQ provided bitstreams require an absolute path (e.g. `/opt/python3.6/.../pynqhls/stream/stream.bit`). We provide a method that searches for bitstreams co-located with the Python class so that the user only needs to specify the name of the bitfile (e.g. `'stream.bit'`). This vastly reduces the by abstracting the details of the underlying Python package structure.

4.3.6 Packaging an Overlay for Distribution

The final notebook teaches users how to distribute their work using the Python package manager, Pip. Distribution is useful for any academic PYNQ project, and can also be used for grading final projects pulled from a GitHub repository.

The notebook creates a setup.py file on the PYNQ board. The setup.py file specifies what files are installed as part of a Python package: the .tcl file and .bit file from Section 4.3.4, the class from Section 4.3.4, and notebooks. Relevant parts of the file are analyzed, such as how to install python packages, naming dependencies, and destinations for non-python files.

The concepts taught by this notebook are useful outside of the classroom. Many Python projects are distributed using Pip, however it is unfamiliar to our community and the traditional use-cases do not describe how to install and distribute non-python files (.bit, .tcl, and .ipynb files) that are used by PYNQ.

4.4 Related Work

In this section we describe related work in two categories: Embedded Projects using FPGA SoC's and Related Tools. We will demonstrate that our tutorials fill an existing need in the embedded and educational FPGA computing space.

4.4.1 Embedded Projects Using FPGA SoCs

Many academic projects have used SoCs. Low-latency control has been implemented on Quadcopters [38], and computationally-demanding power controllers [62]. The computational benefits of FPGAs also make them good for embedded computer vision projects [39] where low-latency compuation on streams is needed for smooth tracking. FPGA SoCs have been used in Software Defined Radio projects, such as GNURadio [37]. The applications these projects target are exemplary of common FPGA computational patterns: streaming for audio, video, and signal encoding, and batch processing for images and video, and low latency control loops.

Our work is useful for embedded projects like the ones shown above. By creating accessible, interactive tutorials that avoid verilog, and provide skeleton projects we believe we have reduced the learning curve for PYNQ and widened its appeal.

4.4.2 Related Tools

Our work intersects with several system-design tools: SDSoC by Xilinx [29], Intel (Altera) OpenCL, and LegUp[25] from University of Toronto. All tools take C/C++ code and emit complete systems for Intel SoCs. Unlike these tools, our tutorials teach users how to integrate HLS into complete systems once an C/C++ core has been developed.

Our demonstrate external communication, which is difficult to implement in the tools described above. No tool above can currently drive external IO pins directly. This concept is demonstrated in our Real-Time IO lab in Section 4.2.1. Vendor tools allow users to communicate with AXI/Avalon peripherals - but this is not the default use case. In SDSoC users must develop

a new SDSoC platform and in Intel OpenCL users must develop Board Support Packages [64]. We feel that easy external communication is necessary for embedded development. For example, Three-Phase motor controllers require direct access to IO pins, and external IMU chips communicate over I2C or SPI. We conclude that there is a gap in applications that is not covered by existing tools.

4.5 Conclusion

In this paper we have described our work to create a lab curriculum for Hardware/Software Co-Design that uses PYNQ. These tutorials cover a diverse set of concepts that are relevant in our classes and projects we are involved in. This fills a gap in existing PYNQ resources that can be reused, modified, and extended in other settings. We hope that other people find these tutorials useful, and that they find time to explore our work and contribute feedback.

4.6 Acknolwedgements

The authors acknowledge the National Science Foundation Graduate Research Fellowship Program, the Powell Foundation, and ARCS foundation for their support.

Chapter 5

Everyone's a Critic: A Tool for Evaluating RISC-V Projects

5.1 Introduction

The RISC-V specification is an open Instruction Set Architecture (ISA) standard for flexible low-cost processors[30]. The specification defines 32-bit, 32-bit embedded, 64-bit, and 128-bit base ISAs and optional extensions for compressed, multiplication, atomic, single, double, and quad-precision floating point instructions. This is flexibility is a major benefit for designers, who choose extensions to fit their design goals and reuse popular open-source software toolchains for compilation.

The flexibility of RISC-V lends itself to highly-parameterized FPGA implementations, but the flexibility of RISC-V also makes comparing these projects difficult. Most projects deviate from the base specification, implement different versions of the specification, or implement different sets of extensions. These changes can cause code that runs on one processor to hang on another. It is even more difficult to compare RISC-V projects to wide-spread and commercially available processors like MicroBlaze - which have different instructions, interfaces, and features. These challenges beg the question: *How do we evaluate RISC-V soft-processor projects?*

We need an evaluation tool for studying RISC-V soft-processors. Such a tool must faciliate experiments by providing an interface to run tests and benchmarks. In addition, it should provide a seamless interface for users to write and run code. Next, it must provide a library

of processors with the ability to switch, modify, and add. Finally, this tool should be flexible enough to handle inter-ISA variations for comparisons within the RISC-V family.

The impact of such a tool would be widespread: Prospective users can make informed choices about RISC-V projects; Researchers can use collections of processor designs for comparative analysis, such as building ISA comparisons[] or for building hardware security metrics; Educators can use this tool to build RISC-V curricula [65]; and the RISC-V community can grow using a flexible testing and development tool that encourages competition

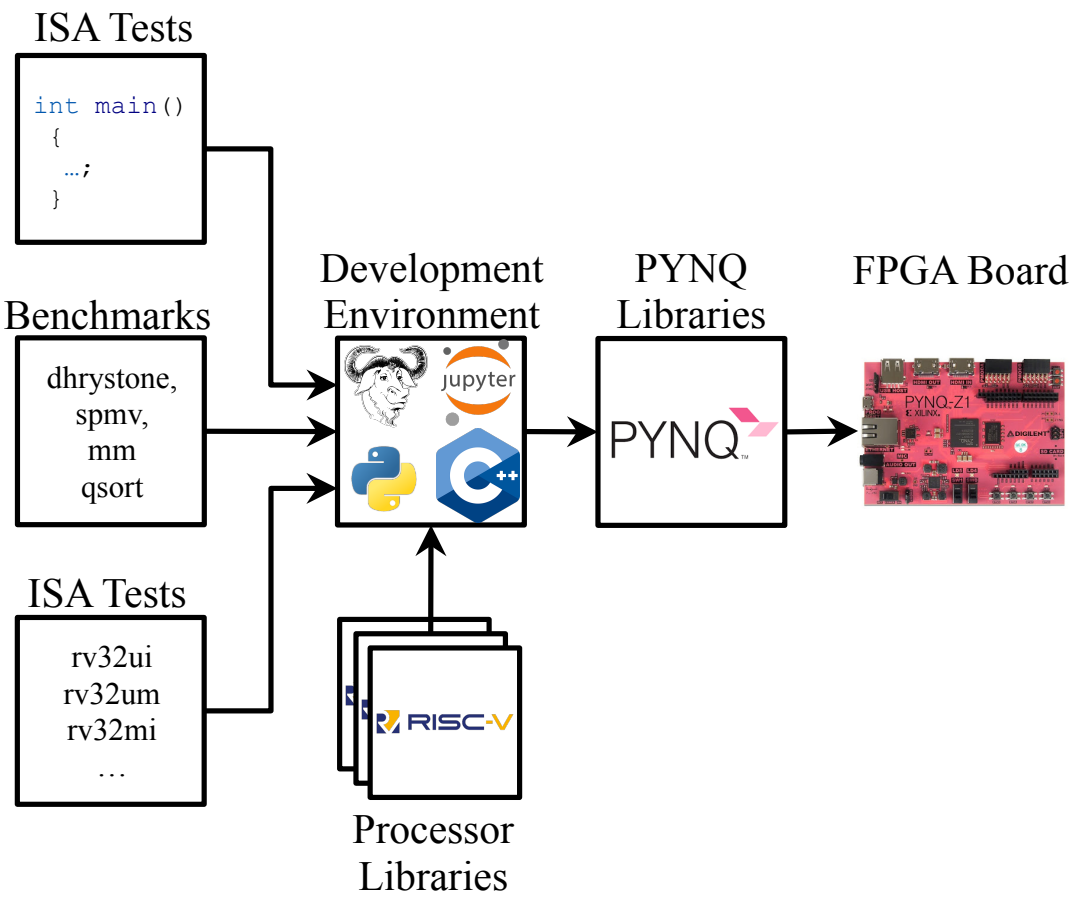


Figure 5.1. Our work presented in this paper. Our tool provides a web interface for compiling C/C++ code, benchmarks, and ISA tests targeting a library of RISC-V soft processors. This project is built on top of other widely used software tools and is easily extensible to other RISC-V projects and soft processor ISAs using our Tutorial

In this paper we demonstrate our open-source tool for evaluating RISC-V soft-processors shown in Figure 5.1. Our tool provides users with an interface for writing and compiling user C/C++ code, standard ISA tests, and benchmarks. Users can choose from a library of RISC-V processors. Our environment uses Jupyter Notebooks, IPython, and standard GNU C/C++ toolchains. All of this is built on top of PYNQ - a collection of Python libraries for ZYNQ All-Programmable Systems on Chip (APSoCs). We use PYNQ to upload bitstreams and executables to the FPGA fabric. Together these parts provide a flexible environment for evaluating RISC-V soft-processor designs.

We use our tool to evaluate 10 soft-processor designs from 5 RISC-V projects. We report report area, frequency, testing, and benchmark results. We find substantial variation in maximum frequency, area, and throughput across RISC-V projects. We also find that few RISC-V processors completely pass the standard RISC-V ISA tests.

We extend our tool to compare these RISC-V soft-processors with 3 MicroBlaze soft-processors and find that the RISC-V projects are competitive with MicroBlaze. One RISC-V project has higher performance with and similar area and frequency across all architectures. Another RISC-V project has higher FMax and smaller area than a corresponding MicroBlaze design. Regardless of architecture, we find that the RISC-V ISA retires fewer overall instructions that MicroBlaze when executing the same benchmark.

We conclude with a summary of our experiences working with RISC-V and MicroBlaze processors. We find that while RISC-V is techincally competitive with MicroBlaze, most RISC-V projects lack sufficient documentation, have non-standard interfaces, and lack integration with standard vendor tools. Despite these, we found that RISC-V was still substantially easier to integrate and work with than MicroBlaze.

Our main contributions are:

- A environment for soft-processor comparison.
- A survey of 13 processors implementing RISC-V and MicroBlaze ISAs.

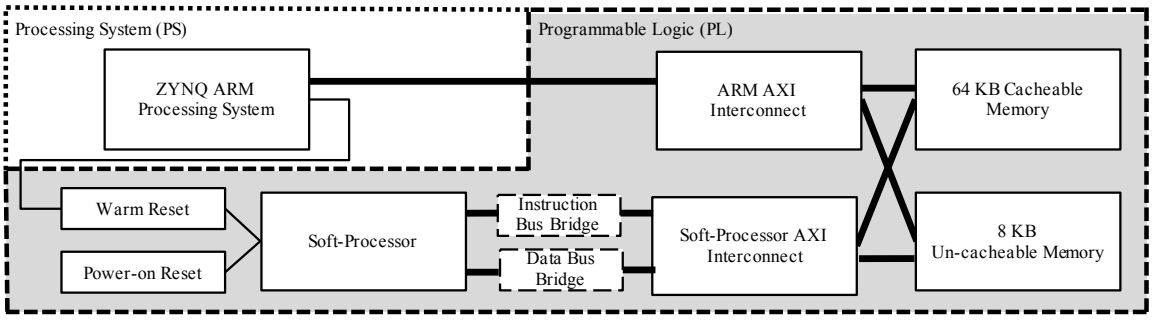


Figure 5.2. A block diagram the soft processor evaluation infrastructure as described in Section 5.2.2.

- An tutorial for adding RISC-V processors to our library.
- A summary of our experience working with RISC-V and Microblaze.

All the work in this paper can be found in our open source repositories, under a BSD-3 license.

This paper is organized as follows: Section 5.1 introduces our paper and the environment we have created. We describe our environment in Section 5.2. Finally, we describe the RISC-V projects we have chosen and report results in Section 5.3. Section 5.4 describes related work in web-based IDEs and soft-processor research. We conclude in Section 5.5

5.2 A RISC-V Soft-Processor Evaluation Tool

In the last few years an overwhelming number of RISC-V cores have been released by academia, industry and hobbyists. For example, the Rocket-Chip from UC Berkeley implements the canonical RISC-V System-on-Chip for architecture research. The academic start-up Vector-Blox has produced Orca to host the MXP vector processor. RoaLogic has developed RV12 a commerical RISC-V IP. Finally, the VexRiscV project implements a high-performance RISC-V processor for FPGAs. These projects are just a small fraction of the list shown in Section 5.4.

Choosing a RISC-V core from the growing list of projects is a daunting task. The RISC-V specification is highly flexible and lends itself to highly-parameterized projects. Many

projects deviate from the base specification, implement different versions of the specification, or implement different sets of ISA extensions. As we show in Table 5.1 RISC-V projects are also written in many hardware development languages. What users need is a tool for evaluating RISC-V projects that allows them to experiment and test projects with minimal committment.

An ideal tool for evaluating RISC-V projects has several components: It provides a library of RISC-V processors for evaluation with a variety of features and ISA extensions. To allow growth, it has an automated path for adding new processors with standard hardware interfaces (AXI, Avalon, and Wishbone). To aid in testing this tool provides standard benchmarks and tests to be run. To encourage development this tool provides an interface for writing, compiling, and uploading code to a processor – much like Arduino or mbed environments. This tool should also provide a software debugger for analyzing faults. Such a tool would help users navigate the overwhelming buffet of RISC-V projects available.

We have developed a tool for RISC-V experimentation that meets many of these requirements. We use FPGAs because they are an ideal platform for prototyping and evaluation. We have chosen to use PYNQ as our delivery platform because it provides low-cost boards, and accessible user interface on top of well-engineered Python libraries.

The components of this tool are descibed in the next sections: Section 5.2 describes the hardware infrastructure we use in our soft-processor bitstreams. Section 5.2.2 describes our our interface for writing, compiling, and running user code, tests, and benchmarks from a browser-based interface. Finally Section 5.2.3 covers our interactive tutorial for adding RISC-V projects to our library.

5.2.1 Hardware Infrastructure

The hardware infrastructure architecture is a test harness for the FPGA Soft-Processor and is shown in Figure 5.2. The components of this test harness are shown below:

- 1 ZYNQ ARM Processing System

- 1 FPGA Soft Processor
- 1 Soft-Processor AXI Interconnect
- 1 ARM PS AXI Interconnect
- 1 64KB on-FPGA Memory
- 1 4KB on-FPGA Memory
- 2 Reset Controllers (Power-On-Reset, and Warm-Reset)

Our infrastructure to designed to provide fair comparisons by minimizing variations introduced by external factors. We use on-FPGA memories instead of DDR memories because they provide a fixed memory latency. One the cacheable memory, and one un-cacheable is provided for all processors so that processors without caches do not benefit from reduced area pressure during compilation. This decision has an added benefit of unifying our standard library functions across processors. These decisions create fairer comparison across soft-processor projects and across architectural variations within a project.

The test infrastructure is controlled by a Python class running on the ARM Processing System. This python class has methods for restarting and running the arm processor, and issuing memory commands to the two soft-processor memories.

5.2.2 Development Environment

Modern embedded IDEs like Arduino and mbed provide the ability to write, upload, and run code from a single interface. The Python class briefly described above can upload program binaries to the 64 KB memory in the test infrastructure. These binaries are compiled in the linux terminal provided on the PYNQ board, or on a separate machine. While this may be trivial for readers with extensive linux knowledge it is far from the modern development environment most have come to expect. What is needed, is an environment to write, test, and develop code from an IDE and push it to the soft-processor.

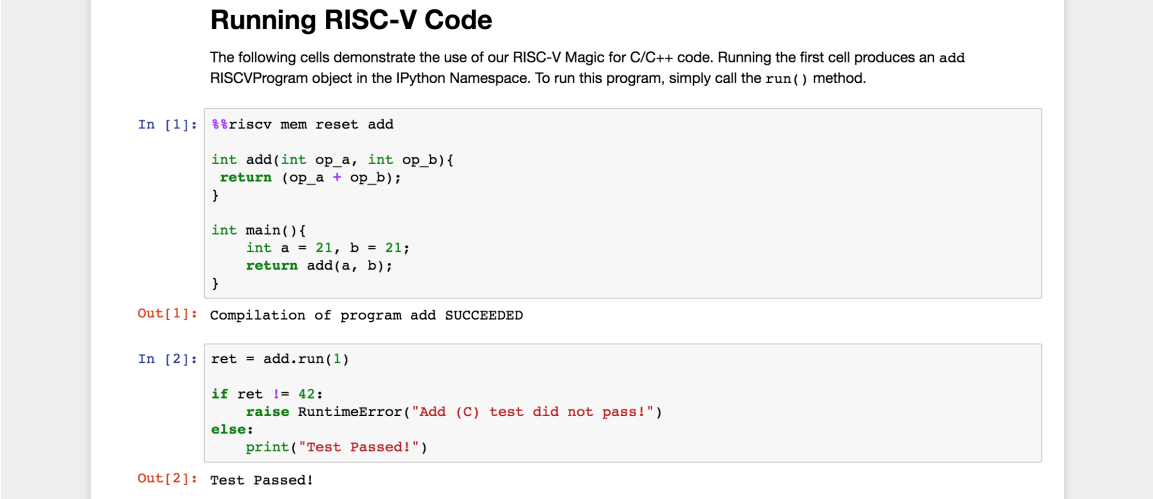


Figure 5.3. Our development environment. The %%riscv IPython magic takes three arguments: The PYNQ MMIO obect representing the soft-processor memory, the PYNQ GPIO object representing the soft processor reset, and the name of the program to be created in the namespace. If compilation succeeds, the program can be run in subsequent cells.

We have created a web-based Development Environment (IDE) using IPython Magics in Jupyter Notebooks that allows users to write, compile, upload, and run high-level code on a RISC-V soft processor. Our work is based on the work in [66].

IPython Magics change the functionality of code cells in Jupyter Notebooks. IPython Magics can be used to run shell commands or change the environment; we use IPython Magics to provide a web interface for C/C++ code complete with syntax highlighting. IPython integration is delivered as two Python classes: `RISCVMagics` and `RISCVProgram`.

The `RISCVMagics` class implements the IPython Magic comands `%%riscvc` (shown in Figure 5.3 and the similar `%%riscvasm`. When one of these commands is typed, this class performs syntax highlighting on C/C++ and assembly in a Jupyter Notebook cell. When the cell is run this class reads the code and compiles it into a RISC-V binary. When the Jupyter Notebooks cell is executed, the cell will compile the program and print SUCCEEDED, or FAILED (and a compiler error message) depending on the compilation outcome.

If the compile succeeds, the `RISCVMagic` class will insert a `RISCVProgram` object repre-

senting the compiled binary into to the local namespace. This is shown in Cell 2 of Figure 5.3. The `RISCVProgram` object represents a compiled instance of a `RISCVMagic` cell and can be used to, start, stop and run a compiled cell.

We also provide GNU Newlib standard libraries, which allow host-board host-board communication with printf and scanf. When combined with the IPython Magics, and the RISC-V GNU toolchain the Newlib libraries provide an easy-to-use IDE for compiling and running code on a RISC-V processor that can be used in classrooms and in research labs.

Table 5.1. Project summary of selected soft-processor projects

Design Name	Language	Target	Memory Intf.	ISA Ver.
ORCA[67]	VHDL	FPGA	AXI, Wishbone, Avalon	v1.7
PicoRV32[68]	Verilog	FPGA	AXI, Wishbone	v1.9
Rocket[30]	Chisel	VLSI	AXI*	v2.0
RV12[69]	SystemVerilog	VLSI	AHB, Wishbone	v1.9
VexRiscV[70]	Spinal HDL	FPGA	AXI*, Avalon	v1.9
MicroBlaze	N/A	FPGA	AXI, LMB, ACE	N/A

Table 5.2. Architectural feature summary of selected soft-processor projects

Design Name	IRQ Handler	Debug Intf.	I & D Cache	Branch Pred.
ORCA[67]	Custom	No	No	Static
PicoRV32[68]	Custom	No	No	No
Rocket[30]	Standard	Required	Required	Optional
RV12[69]	Standard	Required	Optional	Optional
VexRiscV[70]	Standard	Optional	Optional	Optional
MicroBlaze	N/A	Optional	Optional	Optional

5.2.3 Tutorial

Creating a RISC-V overlay can be a daunting process for a variety of reasons: the inaccessibility of HDLs, interface standards, and complexity of vendor tools. In our work alone, we survey 5 RISC-V processors, written in 4 languages, and providing 3 interface standards as shown in Table 5.1.

To address this problem we have created a a tutorial to lead users through the process of adding a RISC-V processor to our library of processors. Our tutorial is a set of interactive Jupyter Notebooks that teach the reader how to build a soft-processor overlay for PYNQ. Each notebook directs the user through a conceptual step in building an overlay, and concludes with notebook cells that verify the notebook has been completed successfully.

By the end of the tutorial readers should understand how the PYNQ libraries use Vivado-generated .tcl files, FPGA .bit files, and user-generated Python files in an Overlay package. Readers will should also be able to trivially extend this tutorial to other soft-processor projects.

Our tutorial is organized as follows:

Configuring the Development Environment

The first notebook of the tutorial teaches the reader how to install dependencies both on a development machine and then on the PYNQ board. The user is directed through installing the Vivado tools, cloning the soft-processor git repository, installing a visual-difftool (such as Meld). On the PYNQ board the user is directed through installing packages from APT, and cloning the necessary RISC-V GNU Toolchain repository.

Building a Bitstream

The second part of the tutorial teaches a reader how to build an FPGA bitstream from a RISC-V processor project. It is divided into three parts three parts: *Packaging a RISC-V project as a Vivado IP*, *Creating a Block Diagram*, and *Building a Bitstream From .tcl*. These components are necessary to use the PYNQ framework.

The notebook begins by teaching the reader how to package the picorv32 RISC-V Processor from [68] as a Vivado IP through pictures. We chose this processor because of its ease of use and integration with Vivado.

The second part of this notebook describes how to build a Block Diagram for PYNQ inside of Vivado IP Integrator. The tutorial provides a "skeleton" .tcl file that creates a project,

instantiates the configured ZYNQ Processing System (PS). The reader then adds the PicoRV32 processor IP and the remaining soft-processor environment IP from Section 5.2 and assigns an address map.

In the final section, the reader validates the design and updates the "skeleton" .tcl file with these modifications. The reader is then asked to compile their RISC-V design using the .tcl file they have created to verify that there are no errors.

Compiling the RISC-V Toolchain

The third notebook of this tutorial teaches the reader how to compile the RISC-V GNU Toolchain. When the compilation is complete subsequent cells update the PATH environment variable on the reader's PYNQ board.

Packaging a Bitstream into an Overlay

The fourth notebook of the tutorial teaches the reader how to package the .tcl file, .bit file, and other newly-generated files into a PYNQ Overlay. The notebook is divided into three parts.

In the first section, this notebook demonstrates how to create a RISC-V overlay class, and declare it as part of a package. This requires creating two files: an __init__.py file that declares the directory as a Python package, and a `riscv` class that inherits from `pynq.Overlay`

Next, the tutorial leads the reader through building the notebooks directory. The notebooks directory holds the .ipynb metadata files for Jupyter Notebooks, but in our tutorial it also holds the RISC-V compilation scripts. The compliation scripts consists of three files: A Makefile for compilng RISC-V code, a reset.S assembly script for initializing the RISC-V processor at reset, a riscv.ld linker script describing the memory layout for the GNU linker tool.

Finally, the tutorial walks the reader through the step-by-step process of creating a setup.py script that will automatically install their overlay alongside PYNQ.

This notebook finishes by verifying the user work: First, the notebook verifies that the RISC-V compilation files function as expected. Second, the notebook notebook verifies that the

Overlay can be imported as a Python package (pre-installation), and finally the notebook runs Pip and then verifies that the overlay is installed correctly post-installation.

Writing and Compiling RISC-V Code

The tutorial concludes in the fifth section by walking the reader through compiling RISC-V code, running the resulting programs, and reading results from the processor memory. In the final step, this tutorial uses the web-based environment described in Section 5.2.2 to compile and upload code to a RISC-V processor.

The tutorial starts by demonstrating how to write programs in assembly, or C/C++, compiling, and then reading the results back from memory. These simple code snippets are loaded into the notebook directory and compiled using the makefile from the previous step. Communication between the ARM PS and RISC-V processor is simple: either "spin" on a memory location, or "sleep" until the program is guaranteed to terminate. Once the RISC-V Program is complete, the ARM processor can read the results from the RISC-V memory space.

Finally, the tutorial introduces the development environment described in Section 5.2.2. Users are directed through installing the IDE package using Pip. These final cells verify that the user has completed the entire tutorial successfully.

5.3 Results

We now use the environment we have developed to gather compilation and performance results for selected RISC-V projects. The scripts and makefiles necessary for recreating our results can be found in our repository [71].

5.3.1 Projects

The projects we have chosen are summarized in Table 5.1. We selected these projects based on several criteria: Source availability, parameterization, documentation, and provision of

[1]This MicroBlaze design includes resources added by the parameter C_DEBUG_ENABLED = 2, which is needed for performance counters used in Table 5.6

Table 5.3. Soft-Processor F_{Max} (MHz) and Resources Consumed

Simple	**F_{Max}(MHz)**	**LUT**	**FF**	**BRAM**	**DSP**
ORCA	87.11	1873	1097	1	4
PicoRV32	176.95	1426	833	0	4
RV12	118.36	3517	2612	0	4
VexRiscV	133.59	1861	1377	1	4
MicroBlaze	133.59	1148	872	0	3
MicroBlaze[1]	133.59	1898	1948	0	3
Cached	**F_{Max}(MHz)**	**LUT**	**FF**	**BRAM**	**DSP**
Rocket	95.31	6166	3360	3	0
RV12	68.75	4393	3037	5	4
VexRiscV	133.59	2146	1807	6	4
MicroBlaze	133.98	1708	1353	6	3
MicroBlaze[1]	133.59	2397	2321	6	3
Predicted	**F_{Max}(MHz)**	**LUT**	**FF**	**BRAM**	**DSP**
Rocket	95.31	6803	4305	3	0
RV12	68.75	4400	3054	5	4
VexRiscV	133.59	2146*	1807*	6	4
MicroBlaze	133.59	1990	1485	7	3
MicroBlaze[1]	133.59	2639	2456	7	3

standard memory interfaces (AXI4, AXI4Lite, AHB, and Avalon). All projects are configured to implement the RV32 Integer instruction set with Multiply/Divide extensions (RV32IM).

Three of the RISC-V projects are FPGA targeted: ORCA, PicoRV32, VexRiscV. The PicoRV32 processor is unique because it is a multi-cycle processor optimized for area and F_{Max}. The ORCA processor is the only surveyed processor that targets all four FPGA vendors: Xilinx, Intel, Mirosemi, and Lattice. Finally, the VexRiscV processor demonstrates the scala-based SpinalHDL language.

Two of the RISC-V designs above are VLSI silicon designs: RV12, and Rocket-Chip. The Rocket-Chip project is the canonical RISC-V implementation[72]. The RV12 is a silicon IP core created by RoaLogic[69] for the eASIC process.

For a fairer area comparison we remove parts from both chips: We modified the Chisel source to remove the Debug module and Boot ROM from the Rocket-Chip processor, and added a top-level parameter to remove the Debug module from the RV12 processor. Both designs provide parameters for multiply latency which is set to 3 in all experiments.

We compare these designs to the standard Xilinx MicroBlaze processor. This processor has been configured for a 32-bit 5-stage pipeline (Performance) and AXI Interfaces. We have enabled the hardware multiply, divide, and barrel-shift instructions for the fairest comparison with the RV32IM implementations above.

For area and F_{Max} results shown in Table 5.3 we show two MicroBlaze results: MicroBlaze, without performance counters (C_DEBUG_ENABLED = 0), and MicroBlaze*, with performance counters (C_DEBUG_ENABLED = 2). Performance counters are necssary for gathering results in Table 5.6 but may not be included in a final MicroBlaze design. Nonetheless, we show these results to highlight the difference between RISC-V and MicroBlaze processors.

From these projects we generate 13 soft-processor overlays for comparison as described in Section 5.2. We group the 13 overlays into 3 categories:

- **Simple**: No caches, no predictor

- **Cached**: 8 KB I & D caches, no branch predictor
- **Predicted**: 8 KB I & D caches, with branch predictor

L1 Instruction and L1 Data caches are each 8 KB, corresponding to a 512-entry direct-mapped cache with a 16 byte line size. A 16-byte line size is the minimum possible for a Xilinx MicroBlaze cache, and 512 entries fits in a 36 KB cache without cascading. With this configuration each cache should consume 3 BRAMs: 2 BRAMs for cache line data, and 1 BRAM for tag data.

Branch predictors are sized to match Xilinx primitives on a per-processor basis. The MicroBlaze processor implements a 4 KB Branch Target Cache to fit in one 36-KBit Xilinx BRAM as recommended by the MicroBlaze user guide. VexRiscV and RV12 processor implement a direct-mapped, 64-entry table of 2-bit counters to fit inside a RAM64M primitive. Rocket-Chip implements a direct-mapped 64-entry Brach History Table of 2-bit counters and fully-associative 8-entry Branch Target Buffer.

5.3.2 Compilation Results

Table 5.3 shows compilation results for 13 FPGA soft-processor designs. These results were gathered using Vivado 2017.1 on Ubuntu 16.04 targeting a PYNQ-Z1 board with a XC7Z020-1 chip. Synthesis and Implementation objectives were set to default.

Frequency results were gathered using a binary search algorithm as part of the overlay .tcl compilation script. Resource utilization results were taken from the maximum frequency compilation result. Only the resources consumed by the FPGA soft-processor are shown.

We highlight three interesting results from our data:

RISC-V designs can be competitive with MicroBlaze

The Simple PicoRV32 design is the only design that consumes less FFs, and produces a higher F_{max} than an equivalent MicroBlaze design. However, it uses 300 more LUTs and 1 more DSP than the MicroBlaze design.

All VexRiscV designs produce F_{max} results that meet equivalent MicroBlaze designs. However, the VexRiscV design is larger by the worst-case 700 LUTs and 500 FFs for the Simple design, and this difference decreases monotonically to the best-case of 250 LUTs and 300 FFs in the Predicted design.

All RISC-V designs use one more DSP than an equivalent MicroBlaze design, except the Rocket-Chip design which uses 0 DSPs. It is not clear whether this can be recovered through design optimization or if this is a limitation of the ISA.

All VexRiscV designs are smaller than MicroBlaze the MicroBlaze is configured with performance counters. These performance cannot be read without enabling the MicroBlaze debugger and come at substantial cost of 600-700 LUTs, and 1000 FFs.

FPGA-Targeted does guarantee minimum area

Of the five RISC-V projects we surveyed three were FPGA-Targeted and two were VLSI-Targeted (Table 5.1). The two silicon-targeted variants produce designs that are almost twice as large as FPGA-targeted variants. These results are likely the consequence of different design tradeoffs.

With the exception of the Simple Orca design, the VLSI-targeted designs also produce the slowest designs out of all the designs surveyed.

Scala-based languages can produce efficient hardware

The VexRiscV project uses SpinalHDL, a new scala-based language similar to Chisel. As shown in Table 5.1, the VexRiscV processor is the most competitive RISC-V processor design we surveyed.

5.3.3 ISA Tests

We tested each RISC-V processor with the standard rv32ui (User-level Integer), rv32um (User-level Multiply) ISA tests from the riscv-tests repository. These correspond to all the minumum required implementation level tests for RISC-V processor [30].

Table 5.4. RISC-V ISA Test Results for surveyed processors. Results are reported as: (Pass — Fail — Unknown)

Simple	**rv32ui**	**rv32um**
	P — F — U	P — F — U
ORCA	27 — 12 — 0	3 — 5 — 0
PicoRV32	38 — 0 — 1	8 — 0 — 0
RV12	28 — 11 — 0	5 — 3 — 0
VexRiscV	38 — 0 — 1	7 — 1 — 0
Cached	**rv32ui**	**rv32um**
	P — F — U	P — F — U
Rocket	39 — 0 — 0	8 — 0 — 0
RV12	0 — 0 — 39	0 — 0 — 8
VexRiscV	38 — 0 — 1	7 — 1 — 0
Predicted	**rv32ui**	**rv32um**
	P — F — U	P — F — U
Rocket	39 — 0 — 0	8 — 0 — 0
RV12	26 — 13 — 0	5 — 3 — 0
VexRiscV	38 — 0 — 1	7 — 1 — 0

The results of our tests are shown in Table 5.5. The ISA tests report Pass (P) or Fail (F). We also report Unknown (U) to indicate when a test didn't complete or produced anomalous behavior.

Our ISA Test results demonstrate that RISC-V projects vary widely in their correctness. The Rocket-Chip project implements all expected arithmetic instructions correctly. Conversley, the RV12 Cached design fails to complete any of the ISA tests, and fails a significant number in the Simple and Predicted designs. The RV12 is hardly unique, Orca fails a significant number of the ISA tests. The VexRiscV and PicoRV32 also perform well but do not achieve perfection.

5.3.4 Dhrystone Benchmark

We benchmark our designs with the Dhrystone benchmark[73] and other benchmarks in the riscv-tests repository. The Dhrystone benchmark is a synthetic benchmark used extensively to measure integer performance [68, 74]. We only show Dhrystone results for space.

We modified the the riscv-tests repository to handle RISC-V ISA version-specific and

project-specific performance counter variations shown in Table 5.5. MicroBlaze performance counters were obtained from the MicroBlaze Debug Module IP.

Table 5.5. Implementations of RISC-V Performance Counters. No performance counter is implemented in all projects.

	utime	ucycle	uinstret	mtime	mcycle	minstret
ORCA	0xC01	N/A	N/A	0xF01	N/A	N/A
PicoRV32	0xC01	N/A	0xC02	N/A	N/A	N/A
Rocket	0xC01	0xC00	0xC02	N/A	0xB00	0xB02
RV12	0xC01	N/A	0xC02	N/A	0xB00	0xB02
VexRiscV	N/A	0xC00	N/A	N/A	0xB00	0xB02

Our benchmark results are shown in Table 5.6. We report the metrics: Cycles Elapsed, Instructions Retired, Number of Dhrystone Runs, Cycles per Instruction, and Cycles per Dhrystone. The FPGA fabric ran at 50 MHz for this experiment.

We highlight three interesting results:

RISC-V processors execute less instructions

MicroBlaze processors execute about 33 Million instructions to complete 50000 runs of Dhrystone while RISC-V processors execute approximately 20 Million instructions for 50000 runs of Dhrystone. This is a 40% decrease in instructions executed.

A RISC-V processor has a lower CPI than MicroBlaze

The VexRiscV project obtains a Cycles Per Instruction metric of 8.83, 1.70, and 1.70 for Simple, Cached and Predicted designs respectively. In comparison, MicroBlaze obtains a CPI of 9.77, 1.97, and 1.81 for Simple, Cached, and Predicted.

Small, Sub-scalar RISC-V designs can outperform other designs

The PicoRV32 processor has a higher MIPS @ F_{Max} than any of the other designs in its class, despite being a multi-cycle processor. When memory latencies dominate this processor

Table 5.6. Dhrystone Benchmark Results for RISC-V and Microblaze at 50 MHz

Simple	**Cycles Elapsed**	**Instructions Retired**	**Number of Runs**	**Cycles per Instruction**	**Cycles per Dhrystone**	**MIPS @ F_{Max}**
ORCA	217750055	20250029*	50000	10.75*	4355	8.10
PicoRV32	202200049	20250029	50000	9.99	4044	17.72
RV12	247400063	220700309*	50000	1.12/12.21	4948	9.69
VexRiscV	178750178	20250029	50000	8.83	3575	13.67
MicroBlaze	323900150	33150013	50000	9.77	6478	15.13
Cached	**Cycles Elapsed**	**Instructions Retired**	**Number of Runs**	**Cycles per Instruction**	**Cycles per Dhrystone**	**MIPS @ F_{Max}**
Rocket	52000559	20250029	50000	2.57	1040	52.18
RV12	N/A	N/A	50000	N/A	N/A	N/A
VexRiscV	345000763	202500029	500000	1.70	690	67.91
MicroBlaze	65400368	33150013	50000	1.97	1308	78.41
Predicted	**Cycles Elapsed**	**Instructions Retired**	**Number of Runs**	**Cycles per Instruction**	**Cycles per Dhrystone**	**MIPS @ F_{Max}**
Rocket	51100570	20250029	50000	2.52	1022	37.77
RV12	N/A	N/A	50000	N/A	N/A	N/A
VexRiscV	345000763	202500029	500000	1.70	690	73.69
MicroBlaze	60100404	33150013	50000	1.81	1202	78.41

can outperform other processors because of its significantly higher F_{Max} that was reported in Table 5.3.

5.3.5 Experience

In this final subsection we relate some of our experiences when developing this project. Our feedback is based on the previous open source projects we have contributed to and our observations of the community at large.

We found that RISC-V processors are substantially easier to bring-up, debug, and measure than their MicroBlaze counterparts for several reasons:

First, the MicroBlaze processor has numerous interface ports to local BRAMs (LMB), to IO Peripherals (AXILITE), and next-level memory (AXI4/ACE). How memory requests are assigned to ports can lead to corrupt data and erroneous results. In contrast, the RISC-V designs we surveyed typically provided one shared port, or one instruction and one data port where all memory requests were issued.

Second, we found that the presence of an active open-source community developing RISC-V tools and processors was a huge benefit. For example, when we discovered a bug in MicroBlaze GCC on the PYNQ board we had to resort to compiling MicroBlaze binaries from Vivado 2017.1 on our host computer. When the same thing happened with RISC-V we simply re-compiled the toolchain on the board.

Finally, we found RISC-V easier to measure because performance counters are part of the base specification and implemented by most projects. For contrast, the MicroBlaze performance counters can only be read through an indirect read and write of registers in the MicroBlaze Debug Module.

Even though we found RISC-V projects easier to use we still encountered issues that the RISC-V community should work to improve.

In general, the RISC-V projects we surveyed did a poor job of implementing interface standards. One project provided a non-standard AXI interface where the AXI Address ports

were multiplexed between read and write commands. Two projects provided non-standard AXI naming conventions. Providing standard interfaces that match the interface specification simplifies the process of wrapping a RISC-V processor into a PYNQ Overlay. The standard interfaces provided with the PicoRV32 was the main reason we chose it for our Tutorial in Section 5.2.3.

We also found that most projects did not provide sufficient documentation. All projects documented top-level parameters but few projects provided documentation describing the internals of the RISC-V core, or in-line comments to help users read and understand the underlying HDL. For example, when determining how a particular branch predictor was implemented we often had to read the source code to discover details. The RV12 project from RoaLogic should be commended in this respect for providing extensive top-level and sub-module documentation.

Finally, work needs to be done to expose the top-level parameters of Scala-based HDL languages to vendor tools. This paper is not the first to show the power of new HDL languages like Chisel and SpinalHDL. However, we felt that the need to constantly re-generate the HDL when changing parameters was error-prone and hard to version control. In contrast, changing parameters in traditional HDL languages meant modifying the top-level parameter in the relevant .tcl file where it could be tracked.

5.4 Related Work

We survey related work in three fields: Computer Architecture Education, Web-Based IDEs, and soft-processor projects.

The MIPS32 ISA is pervasive in the computer Architecture curriculum. However, there has been a movement to the RISC-V ISA as demonstrated in [75] which has coincided with the publication of the well-known Patterson and Hennessy book [65] for the RISC-V ISA.

With this movement we are presented with opportunities for re-thinking old tools and integrating new tools into the Computer Architecture curricula. For example, the lowRISC

organization supported a Google Summer of Code project to develop a RISC-V simulator [76] to complement the ubiquitious MIPS-32 simulator, SPIM [77]. Our tutorial would complement these simulators very well.

Browers-based simulation and development environments have also seen adoption in recent years. The most familiar example is the mbed platform [78], though academic examples have been created as well [79]. [80] creates a browser-based simulation environment for MIPS. Our work is derived from the upcoming IPython-Microblaze feature [66], which allows users to program PYNQ Microblazes using Jupyter Notebooks.

RISC-V is not the first soft processor architecture. MIPS32 is the most common soft-processor ISA for education. OpenRISC [81, 82] has also been proposed as the open hardware replacement for closed ISAs.

However, RISC-V is Growing and many groups have released their RISC-V processors online as open-source repositories. A considerable number of projects are academic: Taiga[74, 83], BOOM [84], and rocket [72]. More targeted research projects have tested security extensions [85, 86, 87] while others have sought smallest implementation size [68, 88], or highest performance [74]. Commercial projects address a particular application, like Pulpino [89] or Orca [67], while other commercial companies, like RoaLogic, provide general-purpose RISC-V IP RV12 [69]. Finally, there projects that demonstrate HDL languages like VexRiscV [70] (SpinalHDL) and riscy (Bluespec). This proliferation of open-source RISC-V projects has led to increasinly humorous names, like the YARVI project.

In this paper we present a framework for RISC-V exploration that can potentially be used on all of these soft processor processors. This framework includes a tutorial that can be adapted for any RISC-V curricula. Our framework can be used to explore the RISC-V ISA across languages, across architectures, and across projects in one familiar environment.

5.5 Conclusion

In the previous sections we demonstrated our for evaluating and comparing FPGA soft-processors. Our tool provides an Arduino-like interface for writing, compiling, and running code that allows users to experiment and evaluate processors. In addition we provide a library of soft-processors and a complete tutorial for adding adding new processors to this library.We also demonstrated that this environment is flexible enough to handle ISA specification variations and deviations.

Using this tool we surveyed 13 processor designs: 10 processors implementing the RISC-V ISA and 3 processors implementing the MicroBlaze ISA. Our results showed that RISC-V designs are competitive with the MicroBlaze processor and can beat the MicroBlaze processor in certain situations.

Further work is needed to fully understand whether RISC-V can ever be a categorical win over other soft processor ISAs, but our work is the first step along that path.

5.6 Acknolwedgements

The authors acknowledge the National Science Foundation Graduate Research Fellowship Program, the Powell Foundation, and ARCS foundation for their support.

This chapter, in full, is a reprint of the material as it was submitted to International Conference on Field-Programmable Custom Computing Machines 2018. A revised version appears at International Conference on Field Programmable Logic and Applications 2018. Richmond, Dustin; Barrow, Michael; Kastner, Ryan. The dissertation author was the primary investigator and author of this paper.

Chapter 6

A General-Purpose Pipeline Architecture

6.1 Introduction

The throughput of modern processors has steadily increased with improvements in frequency, transistor density, and multi-core scaling. Processors have continued to improve as these trends have slowed with architectural optimizations and ISA extensions. Vectorized instructions have improved throughput by executing multiple independent operations, and fused instructions execute several dependent operations. Crypto instructions and other application-specific ISA extensions execute entire functions. However, the traditional fetch-execute-store architecture of processors has inherent inefficiencies in modern applications like video processing, machine learning, and linear algebra [41]. Data movement and re-use in these applications does not map well to modern processor register files and instruction sets.

This has driven computer architects to develop custom hardware pipelines and other non-traditional architectures to overcome these inefficiencies with reuse buffers and deep pipelines. Such architectures improve metrics in video processing [90], signal processing [91], machine learning [92, 10], text processing [8], and genomics [55]. These architectures have demonstrated substantial power savings, increases in throughput, and latency decreases. However, many architectures are highly-customized for a single application and this forces engineers to develop a new pipeline for each application.

Unfortunately hardware development is widely accepted as tedious [16], inabstract [93],

and slow [44]. Hardware development languages require designers to implement clocks, registers, and schedule operations. High-level synthesis tools abstract these in C/C++ [29, 36, 25] but still require detailed understanding of the underlying hardware architecture [94]. Optimized application libraries such as OpenCV [95, 96] can provide more abstraction but still require long compilation iterations. These challenges have limited hardware development to select companies like Google [42], Microsoft [8, 10], and Audi [13] that can afford the associated costs [35]. What is needed is a pipeline architecture that overcomes the inefficiencies of modern processors and is general enough to eliminate hardware development.

In this chapter we propose a pipeline architecture for accelerating general computations on System-on-Chip devices. Our accelerator is implemented with interconnected RISC-V processors in a map-reduce parallel structure [30]. Our architecture allows us to build efficient hardware pipelines, and reprogram them from high-level software tools using the RISC-V ISA. We study our architecture to understand its performance relative to optimized libraries on an ARM processor and hardware pipelines on twelve common functinos. We demonstrate that our architecture out performs an ARM processor with NEON instructions.

This work has four main contributions:

- A study of the trade-offs between processor and pipeline computations
- A set of RISC-V extensions for implementing interconnected RISC-V processors
- A proposal for a general-purpose pipeline accelerator built from interconnected RISC-V processors
- A quantitative comparison between our architecture, and an optimized library implemented in software and hardware pipelines

This chapter is organized as follows: Section 6.2 describes the background for our work. Section 6.3 describes our proposed architecture. Section 6.4 presents our results. We describe related work in Section 6.5. Finally, we conclude in Section 6.6.

6.2 Background

Data-Flow Graphs (DFG) like the one shown Figure 6.1 represent branch-less computations and are useful representations of loop bodies. Each node in a DFG represents an operation, while each edge represents data dependency. We define N_E as the number of entry nodes in a DFG, Figure 6.1 and N_D as the maximum number of nodes from an entry node to the output node, and N_T as the total number of nodes in a DFG. Studying DFGs and their execution can reveal information about the performance of different architectures.

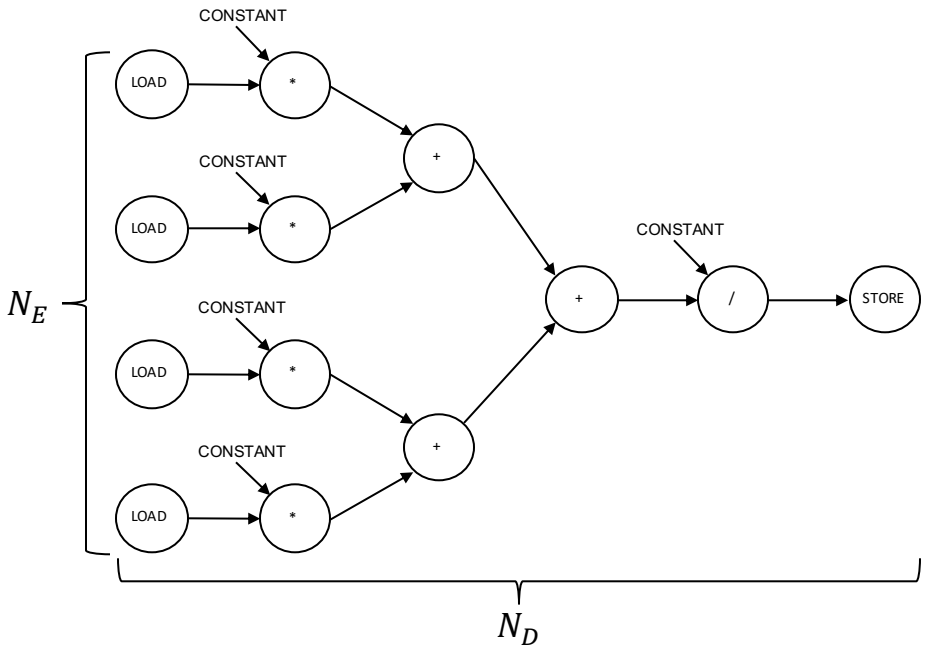


Figure 6.1. A simple Data-Flow Graph (DFG) with $N_E = 4$, $N_D = 6$, and $N_T = 13$.

Data-Flow Graphs are implemented as instruction sequences by a compiler. Figure 6.2 demonstrates a sequence of RISC-V instructions that implements the DFG in Figure 6.1. The memory pointer is in register x1, the multiplication constants are in registers x8-x11, and the division constant is in register x12. In Figure 6.1 the load operations are on Lines 2-5, multiply operations are on Lines 6-9, the addition operations are on Lines 10-12, followed by the division operation on Line 13, and store operation on Line 14.

```
1  dfg_impl:
2          lw   x2, 0(x1)
3          lw   x3, 4(x1)
4          lw   x4, 8(x1)
5          lw   x5, 12(x1)
6          mul  x2, x2,  x8
7          mul  x3, x3,  x8
8          mul  x4, x4, x10
9          mul  x5, x5, x11
10         add  x2, x2,  x3
11         add  x4, x4,  x5
12         add  x2, x2,  x4
13         div  x2, x2, x12
14         sw   x2, 0(x1)
```

Figure 6.2. The instruction sequence from Figure 6.1 in the RISC-V ISA

The Initiation Interval (II) of a computation is the number of cycles between iterations. The Initiation Interval of a processor (II_p) depends on the number instructions executed, which is related to the number of operations (N_T) in the DFG and how the operations map onto instructions provided by the processor Instruction Set Architecture (ISA).

For example, if a processor provides a Fused-Multiply-Add (`fma` instruction the sequence in Figure 6.2 can be shortened by combining `mul` and `add` instructions into the `fma` instruction as shown in Figure 6.3.

```
1  fma_impl:
2          lw   x2, 0(x1)
3          lw   x3, 4(x1)
4          lw   x4, 8(x1)
5          lw   x5, 12(x1)
6          ;; fma rd, rs1, rs2, rs3 ->
7          ;;     rd = ((rs1 * rs2) + rd)
8          fma  x2, x2,  x8, x0
9          fma  x2, x3,  x9, x2
10         fma  x2, x4, x10, x2
11         fma  x2, x5, x11, x2
12         div  x2, x2, x12
13         sw   x2, 0(x1)
```

Figure 6.3. The instruction sequence from Figure 6.1 in the RISC-V ISA with a hypothetical Fused-Multiply-Add (FMA) instruction that computes $(A * B) + C$

Likewise, a processor can provide vectorized instructions to execute multiple unconnected operation nodes in the same instruction as in Figure 6.4. The hyopthetical `vld` instruction on Line 2 simultaneously loads four registers. The `vmul` instruction performs four simultaneous multiplication operations. Vectorized instructions reduce II_p, but they can also be used to produce

P parallel outputs by executing multiple DFGs.

```
1  vec_impl:
2          vld x2:x5, 0(x1)
3          vmul x2:x5, x2:x5, x8:x11
4          add x2, x2,  x3
5          add x4, x4,  x5
6          add x2, x2,  x4
7          div x2, x2, x12
8          sw  x2, 0(x1)
```

Figure 6.4. The instruction sequence from Figure 6.1 with hypothetical vectorized instructions `vld`, which simultaneously loads four registers, and `vmul`, which simultaneously performs four multiplication operations.

Initiation Interval can be used to derive the throughput of a computation as shown in Equation 6.1. This equation computes the throughput (T) in outputs per second, as a function of the frequency (F), output parallelism (P), and Initiation Interval (II). This equation assumes that memory bandwidth is unlimited and all operations take one cycle to complete.

$$T = \frac{1}{t} = \frac{F * P}{II} \tag{6.1}$$

Where:

- t Period, time per result
- II Initiation Interval, the number of cycles between sequential iterations
- F Frequency of the underlying hardware (in MHz)
- V Maximum number of parallel operations provided by vectorized instructions
- P Number of parallel outputs produced by an iteration ($P \leq V$)
- T Throughput in outputs per second

6.2.1 Inefficiencies in Modern Processor

Modern processors have been optimized to reduce the Initiation Interval II using fused instructions that execute multiple dependent DFG nodes in the same instruction or by increasing

P and V with vectorized instructions. Years of silicon process development have maximsed F. Despite this, modern processors still have high Initiation Intervals on important workloads.

These inefficiencies have been studied in numerous papers [41, 97, 91]. These papers have found that processor register files do not support common data movement and re-use patterns. Furthermore, while fused and vectorized instructions reduce the number of instructions issued, they have complex issue rules and often cannot be issued on consecutive cycles [98]. These overheads increase the Initiation Interval (II) and reduce the throughput (T) of these workloads.

These inefficiencies are particularly acute in convolution-like computations that are prevalent computer vision, machine learning, and linear algebra. These computations are slow because their DFGs have a a large number of load instructions relative to useful computation instructions. To address these inefficiencies, many projects have developed custom architectures specifically tailored to to these issues [92, 91, 99, 100, 101, 102]. These projects develop efficient memory buffers that leverage memory re-use to minimize overheads and deep pipelines to improve performance.

6.2.2 The Benefits of Pipelines

Pipelines overcome the inefficiencies of processors by implementing operations in *stages*. Operation results move between stages on every clock cycle. Stages can read memory, manage efficient re-use buffers, perform arithmetic operations, and execute custom functions concurrently. Stages execute operations concurrently, unlike processors which execute operations serially.

The Initiation Interval for a pipeline II_p is equal to the maximum number of cycles any stage takes to execute. Figure 6.5 demonstrates a 5-stage pipeline implementing the DFG from Figure 6.1. The II of the pipeline is 4, equal to the maximum II of the pipeline in Stage 4.

In [90] the authors propose a convolution engine to address situations when II_p is large for image processing computations. The authors demonstrate a 2-3x efficiency improvement, and a 8-15x improvement in performance over a general-purpose core with vectorized instructions.

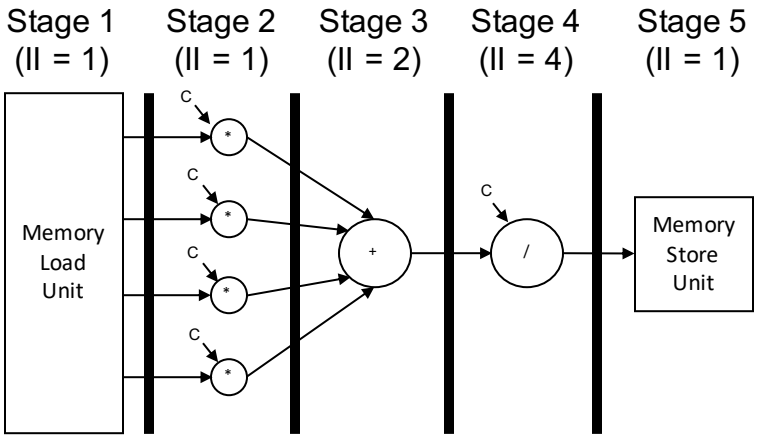


Figure 6.5. A 5-stage pipeline implementing the data-flow graph in Figure 6.1 with $II = 4$

This architecture is integrated as an extension to the Tensillica ISA. Similar architectures have been developed for accelerating convolutional neural networks [42, 92]. These papers fabricate chips, Application-Specific Integrated Circuits (ASIC) to implement their pipelines.

Hardware pipelines can be implemented as ASICs with F_h comparable to F_p but ASICs are time consuming and expensive to manufacture [103]. Increasingly, pipelines are implemented on Field Programmable Gate Arrays (FPGA) [101, 102, 100, 99, 8, 10, 9]. FPGAs have a lower maximum frequency (F_h) and can have high II on complex operations like division. However, theycost substantially less money and provide increased flexibility. This leads to a trade-off.

6.2.3 Processors Versus Pipelines

Figure 6.6 displays the throughput curves for a processor (subscript p), and hardware pipeline (subscript h) using Equation 6.1. The vertical axis measures throughput (T) in results per second and the horizontal axis displays the processor initiation interval (II_p). The blue line charts the performance of a single processor and the orange line charts the performance of a hardware pipeline with a constant initiation interval $II_h = C$. The blue region highlights when a processor has higher performance than a hardware pipeline, and the orange region highlights when a hardware pipeline has higher performance.

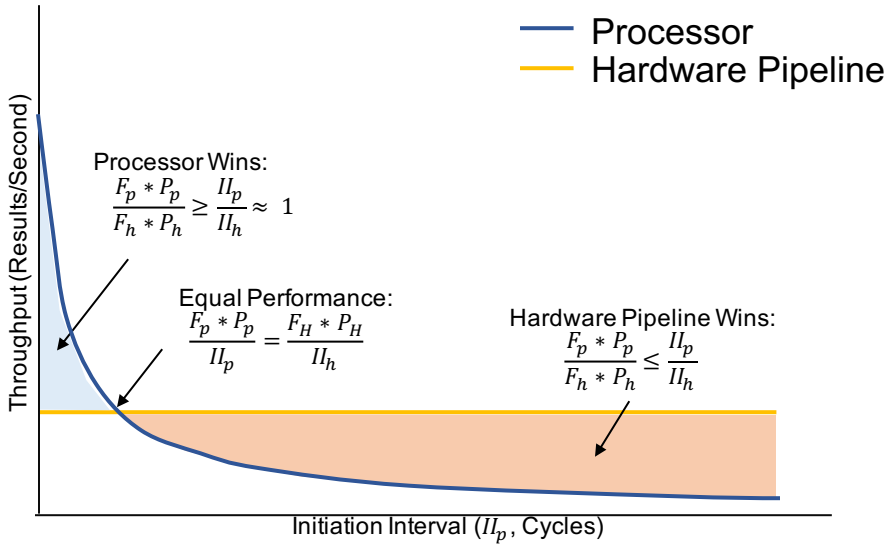


Figure 6.6. Performance curves for a Processor (subscript p) with vector instructions and a Hardware Pipeline (subscript h) with a constant II_h. These are computed using Equation 6.1.

Figure 6.6 demonstrates that pipelines have higher throughput when $F_p * P_p > F_h * P_h$ or when $II_p \gg II_h$. Alternatively, a processor has higher performance when $F_p * P_p > F_h * P_h$, and $\frac{II_p}{II_h} \rightarrow 1$. This is the case in high-performance computing where there are multi-core processors and vectorized processing units. In this space it is more difficult to find applications for hardware pipelines that can overcome the low frequencies of FPGAs.

Hardware pipelines are necessary in mobile environments where $F_p * P_p$ is small due to power constraints and, computer vision, machine learning, and linear algebra applications with high II_p are prevalent. This has led to the growth of System-on-Chip (SoC) devices like Zynq. SoCs provide low-power ARM processors for high-level integration, and FPGAs for building efficient hardware pipelines.

However, FPGA hardware development is substantially more time consuming, difficult, and costly than software development. Software compilation takes seconds, while FPGA compilation takes minutes or hours [44, 43]. Development languages are inabstract, forcing designers to consider low-level details of FPGA architecture [93]. Finally, developing hardware requires experienced engineers [35]. What is needed, is an architecture for exploiting pipeline

parallism in mobile devices that can be programmed quickly, and at a high-level.

6.3 Proposed Architecture

We propose an architecture of interconnected RISC-V processors, shown in Figure 6.7. In our architecture N_H Head (H) processors read data from the L2 cache. N_H Map processors operate on data fetched by the Head processors. $N-1$ Reduce (R) processors form a 2:1 reduction network. Spine (S) processors operate on the result of the Reduce processors. Finally, a Tail (T) processor writes results back to memory. Data is moved between processors with elastic First-In-First-Out (FIFO) buffers.

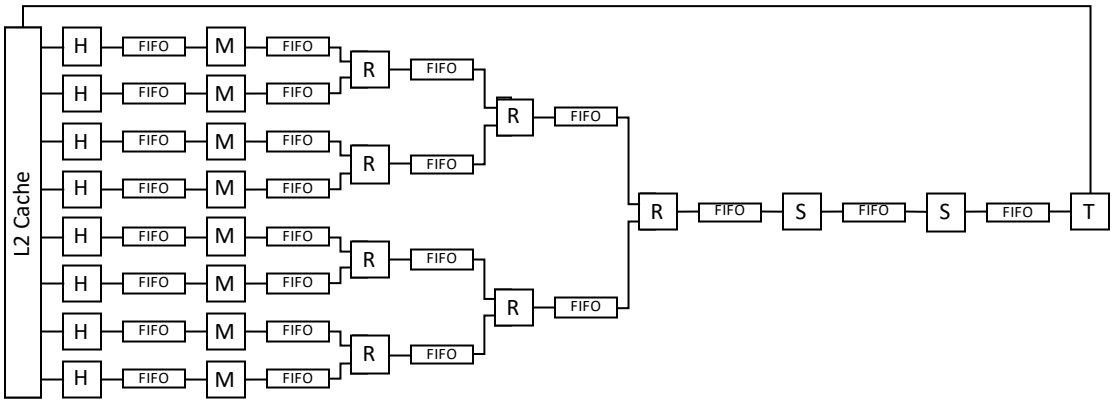


Figure 6.7. Our proposed multi-processor architecture of heterogeneous interconnected RISC-V processors.

The processors are organized in a map-reduce parallel structure that is common in video processing, machine learning, and linear algebra applications [104, 90]. This architecture overcomes the inefficiencies of modern processors: the L2 cache increases data-reuse, and the interconnect allows deep pipelines of interconnected arithmetic units.

Each processor implements the RISC-V ISA with a set of RISC-V extensions described in Section 6.3.1. We use standard extensions to implement the base architecture and functionality. We augment the standard RISC-V extensions with our own custom extensions that enable inter-processor communication, and efficiently execute common patterns. The extensions selected for

each processor are described in Section 6.3.3.

6.3.1 RISC-V Implementation

Figure 6.8 demonstrates our proposed RISC-V processor implementation. Our proposed processor has seven stages: Instruction Fetch (IF), Instruction Decode (ID), FIFO Read and Configuration (FIFO/CSR), Execute 1 (EX1), Execute 2 (EX2), Memory & Look-Up-Table (Mem/LUT), and Write-Back (WB). We have added the FIFO/CSR stage to avoid large, complex multiplexers in the decode stage. The traditional 1-cycle Execute stage has been split into two cycles to maximize frequency.

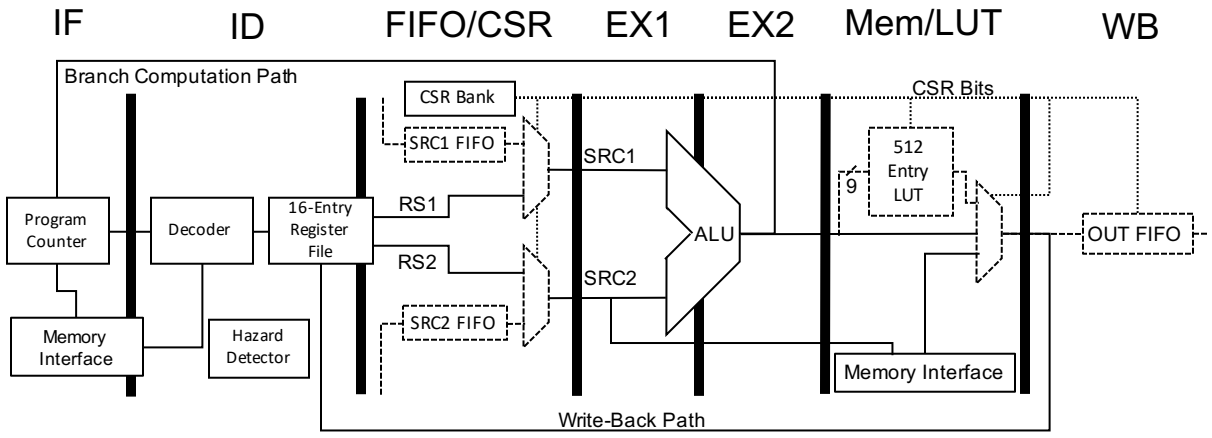


Figure 6.8. Our proposed RISC-V architecture. Dashed lines highlight our custom extensions. Dotted lines indicate configuration bits

Our processor architecture is designed to operated as part of a feed-forward pipeline and implement limited data forwarding logic for Read-After-Write (RAW) hazards. Each processor implements a Hazard unit that is responsible for detecting data or control hazards and causing flushes or forwards.

Each processor in the architecture uses the RISC-V 32-bit ISA with the standard Integer (I) instructions. We select the Embedded (E) extension which provies 16 general-purpose architectural registers instead of 32. This reduces the size of each core by an estimated 25% in

ASIC designs [30]. Each processor operates only at machine level and implements a subset of the Configuration Status Registers (CSR) as shown in Table 6.1. The MCYCLE[H] and MINSTRET[H] registers monitor performance. The remaining registers are used for performance analysis and exceptions.

Table 6.1. Standard RISC-V Configuration Status Registers (CSR) used in our work.

Register Name	Purpose
MCYCLE[H]	Machine Cycle Counter
MINSTRET[H]	Machine Instruction Counter
MSCRATCH	Machine Scratch Register
MTVEC	Machine Trap Vector
MEPC	Machine Exception Program Counter
MCAUSE	Machine Cause Register
MBADADDR	Machine Bad Address Register

6.3.2 Extensions

Each processor is augmented with a selection of extensions. These extensions allow us to efficiently communicate between multiple processors, avoid branch instructions, and implement other non-linear logic that would increase the *II* of our processor designs.

Multiplication

We provide 16-bit by 16-bit multiplication with the MUL[S[U]] instructions to fit in a single FPGA DSP slice. Supporting 32-bit multiplication multiplication and larger doubles and quadruples the DSP cost on FPGAs. We feel that the costs outweigh the benefits in this situation: The 16-bit multiplication is sufficient for 8-bit pixel processing and fixed-point computations that are prevalent in many image processing applications.

Atomics

We implement the Load-Reserved (LR) and Store-Conditional ((SC)) instructions from the RISC-V Atomic (A) extension. This allows data to be safely written back to memory by the

Tail processor and read by the Head processor which is necessary for some image processing kernels.

FIFO Extensions

We define three FIFO extensions for the RISC-V ISA that are shown in Figure 6.8: the SRC1 extension adds the "SRC1 FIFO" that is connected to the SRC1 input of the ALU; the SRC2 extension adds the "SRC2 FIFO" that is connected to the SRC2 input of the ALU; the the OUT extension adds the "OUT FIFO" connected to the WB stage. Each FIFO extension operates independently and they can be implemented selectively.

The FIFO extensions allow processors within our architecture to communicate data efficiently. For example, Head processors in Figure 6.7 load data and transfer it to Map processors through the OUT FIFO extension. Map processors read data from SRC1 FIFO, apply a transformation, and transmit it to a Reduce processor. Reduce processors transmit data to the Spine processors and eventually to the Tail processor for writeback. This effectively allows us to build deep pipelines where each processor is a programmable pipeline stage.

The FIFO extensions add a CSR named `fifoctrl` at address `0xFF0`. The bit-map is shown in Table 6.2. The function of these bits are described below with the corresponding extension.

Table 6.2. CSR `fifoctrl` bit map

Bit	Name	Capability	Description
0	SRC1_EN	WR	SRC1 Implicit Enable Bit
1	SRC1_VALID	RD	SRC1 Input Valid Signal
2	SRC1_LAST	RD	SRC1 Input Last Flag
3	SRC2_EN	WR	SRC2 Implicit Enable Bit
4	SRC2_VALID	RD	SRC2 Input Valid Signal
5	SRC2_LAST	RD	SRC2 Input Last Flag
6	OUT_EN	WR	Output Enable Bit
7	OUT_READY	RD	Output Ready Signal
8	OUT_LAST	WR	Output Last Flag

SRC1 and SRC2 Extensions

The SRC1 and SRC2 extensions add the "SRC1 FIFO" and "SRC2 FIFO" shown in the CSR/FIFO stage of Figure 6.8.

The "SRC1 FIFO" and "SRC2 FIFO" can be explicitly read with the `fsrc1rd` and `fsrc2rd` instructions shown in Table 6.3.

Table 6.3. FIFO read instructions added by SRC1 and SRC2 Extensions

Name	Use	Extension	Description
`fsrc1rd`	`fsrc1rd rd`	SRC1	Read SRC1 FIFO into Register `rd`
`fsrc2rd`	`fsrc2rd rd`	SRC2	Read SRC2 FIFO into Register `rd`

SRC1 FIFO and SRC2 FIFO can be implicitly read by arithmetic, load, and store instructions by setting bits of the `fifoctrl` CSR in Table 6.2. When `SRC1_EN` is set, all set all load and arithmetic that instructions use `rs1` will read and use data from the SRC1 FIFO instead. When `SRC2_EN` is set, all store and arithmetic instructions that use `rs2` will read and use data from the SRC2 FIFO instead. The processor stalls when data is not available during implicit or explicit reads from either FIFO. Implicit reads improve the throughput of computations on the processors by reducing the *II* of each stage.

Figure 6.9 shows a sequence of instructions using the functionality described above. Lines 2 and 3 demonstrate the `fsrc1rd` and `fsrc2rd` instructions from Table 6.3. Lines 5-9 demonstrate implicit reads of the SRC1 FIFO on arithmetic commands, and Lines 11-18 demonstrate corresponding reads for the SRC2 FIFO. Finally, Lines 20 and 21 demonstrate simultaneous implicit reads from SRC1 and SRC2 FIFOs.

OUT Exension

The OUT extension adds the output "OUT FIFO" in the Write-Back (WB) stage of Figure 6.8. This FIFO can be explicitly written with the `foutwr` instruction shown in Table 6.4 The processor stalls when no space is available during writes.

```
1  input_demo:
2          fsrc1rd x1              ;; Read a value from SRC1 FIFO into x1
3          fsrc2rd x2              ;; Read a value from SRC2 FIFO into x2
4
5          csrwi fifoctrl, 0x1     ;; Set SRC1_EN
6          add x3, x4, x0          ;; Implicit read of SRC1 FIFO
7                                  ;; (x4 data is not changed)
8          add x5, x0, x1          ;; The SRC1 FIFO is not read when rs1 is x0
9          addi x6, x1, 1          ;; Read SRC1 FIFO and add 1
10
11         csrwi fifoctrl, 0x8     ;; Set SRC2_EN
12         add x6, x0, x4          ;; SRC2_EN causes the instruction to read
13                                 ;; SRC2 FIFO and replace the data from x4
14                                 ;; (x4 data is not changed)
15
16         add x7, x2, x0          ;; The SRC2 FIFO is not read when rs2 is x0
17         addi x6, x1, 1          ;; The SRC2 FIFO is not read
18                                 ;; (addi does not use rs2)
19
20         csrwi fifoctrl, 0x9     ;; Set SRC1_EN and SRC2_EN
21         add x8, x4, x0          ;; Add values from SRC1 FIFO AND SRC2 FIFO
22
23         csrwi fifoctrl, 0x0     ;; Clear all bits
```

Figure 6.9. A sequence of input FIFO operations using a processor with SRC1 and SRC2 FIFO extensions of our FIFO extensions

Table 6.4. Instructions added by the OUT extension

Name	Use	Extension	Description
`foutwr`	`foutwr rs`	OUT	Write Register `rs` into OUT FIFO

OUT FIFO is implicitly written by arithmetic and load instructions when `OUT_EN` bit of the `fifoctrl` CSR in Table 6.2 is set.

Figure 6.10 demonstrates the functionality of the OUT FIFO. Line 3 demonstrates how the `foutwr` instruction explicitly writes data to OUT FIFO. Lines 5-13 demonstrate how the OUT FIFO is written when the `OUT_EN` bit of the `fifoctrl` register is written. `OUT_EN` is set on Line 5. Line 6 adds the register `x0` and the immediate value 0x2C, and writes the result to register `x2` and OUT FIFO. Line 10 demonstrates that OUT FIFO is not written when the destination register is `x0`.

SRC1/SRC2 and OUT FIFO Co-Operation

SRC1/SRC2 and OUT extensions can be enabled concurrently when multiple extensions are present. This allows each processor to act like a programmable pipeline stage, where data

```
output_demo:
        li x2, 42               ;; Load 42 into x1
        foutw x1                ;; Write the value from x1 into the OUT FIFO

        csrwi fifoctrl, 0x40    ;; Set OUT_EN
        addi x3, x0, 0x2C       ;; Add register x0 (0) and 0x2C and implicitly
                                ;; write it to OUT FIFO in addition to storing
                                ;; it in x2

        addi x0, x0, 0xFF       ;; No Operation, OUT FIFO is not written when
                                ;; rd is x0

        csrwi fifoctrl, 0x0     ;; Clear all bits
```

Figure 6.10. A sequence of FIFO operations using a processor with the OUT extension present

is read from and written to FIFOs by arithmetic instructions. Connecting multiple arithmetic units in series or in a tree like in Figure 6.8 allows programmable multi-stage pipelines to be built from simple RISC-V processors.

Figure 6.11 The FIFO extension allows us to build piplines and program them using the RISC-V ISA. These pipelines meld the high-level programmability of a RISC-V processor with the performance of hardware pipelines, and avoids the drawbacks of both.

```
inout_demo:
        li x1, 0x4000           ;; Load 0x4000 into x1

        csrwi fifoctrl, 0x49    ;; Set SRC1_EN, SRC2_EN, OUT_EN

        add x3, x1, x2          ;; Implicitly read values from RS1_FIFO and RS2_FIFO,
                                ;; Add them together and store the result in OUT_FIFO
                                ;; (x2 is modified as well)

        csrwi fifoctrl, 0x0     ;; Clear all bits
```

Figure 6.11. A sequence of simultaneous Input and Output FIFO Operations.

Min/Max Extension

We extend the RISC-V ISA with minimum and maximum instructions defined in Table 6.5. These instructions compare `rs1` and `rs2`, and write the minimum or maximum into `rd`.

Hazards and branching are expensive in our architecture and the `min` and `max` instructions are useful for transforming branch instructions (control dependencies) into `min`/`max` instructions (data dependencies). In Section 6.4 we use `min` and `max` to saturate pixel values when adding

Table 6.5. `min` and `max` instructions added by the Min/Max extension

Name	Use	Description
`min`	`min rd, rs1, rs2`	Compare `rs1` and `rs2` and write the minimum into `rd`
`max`	`max rd, rs1, rs2`	Compare `rs1` and `rs2` and write the maximum into `rd`

images, implement median filters, and perform non-maxima suppresion in non-linear filters.

Figure 6.12 shows three examples of using `min` and `max`. Lines 3-6 demonstrate how `min` and `max` can be used to find the minimum and maximum of 42 and 13 without branching. Lines 9-12 and 14-18 and demonstrate how `min` and `max` can be used to "clamp" a value to the range 0 to 255.

```
1  minmax_demo:
2          ;; Find the min and max of two values
3          li x1, 42               ;; Load 42 into x1
4          li x2, 13               ;; Load 13 into x2
5          min x3, x2, x1          ;; min stores 13 into x3
6          max x4, x2, x1          ;; max stores 42 into x4
7
8          ;; Clamp x6 to the range (0,255) using Min/Max
9          li x5, 255              ;; Load the saturation value 255 into x5
10         li x6, 273              ;; Load the data value of 273 into x5
11         min x7, x6, x5          ;; min stores 255 into x7
12         max x8, x7, x0          ;; max stores 255 into x8
13
14         ;; Clamp x6 to the range (0,255) using Min/Max
15         li x8, 255              ;; Load the saturation value 255 into x8
16         li x9, -128             ;; Load the data value of -128 into x9
17         min x10,  x6, x5        ;; min stores -128 into x10
18         max x11, x10, x0        ;; max stores 0 into x11
```

Figure 6.12. A sequence of instructions using `min` and `max`.

Look-Up Table (LUT)

We extend RISC-V ISA by adding a Look-Up-Table (LUT). The LUT is located in the LUT/MEM stage in Figure 6.8. When the Look-Up-Table is enabled, the result of any arithmetic operation is replaced by a value from the Look-Up-Table.

The Look-Up-Table is useful for computing non-linear or complex functions. For example, the Look-Up-Table can be used to compute Gaussian coefficients on differences in image intensity for a Bilateral filter, computing the Arctangent function for edge detection, or

computing absolute differences.

The Look-Up-Table is written using the `lutwr` instruction shown in Table 6.6. The register `rs1` specifies the 9-bit address, while `rs2` specifies the 9-bit value. We do not provide a readback instruction. The Look-Up-Table is enabled by writing to the CSR address 0xFC0 (`luten`).

Table 6.6. Definition of the `lutwr` instruction

Name	Use	Description
`lutwr`	`min rs1, rs2`	Write the lower 9-bits of `rs2` into the LUT RAM at `rs1`

Figure 6.13 shows an example using our Look-Up-Table. Lines 2-4 load the value 0xFF into index 42. Line 6 enables the Look-Up-Table. Line 8 adds register `x0` (0), and the immediate value 42. This result is used as the Look-Up-Table index and exchanged for the value 0x1FF, which is written back to register `x3`.

Figure 6.13. A sequence of instructions demonstrating the Look-Up-Table (LUT).

```
1  lut_demo:
2          li x1, 0x1FF
3          li x2, 42
4          lutwr x2, x1            ;; Write 0x1FF to Index 42 of LUT
5
6          csrwi luten, 1          ;; Enable the LUT
7
8          addi x3, x0, 42         ;; x3 is equal to 0x1FF
9                                  ;; The value of x0 + 42 is 42.
10                                 ;; The index 42 is exchanged for 0x1FF
11
12         csrwi luten, 0          ;; Disable the LUT
```

6.3.3 Processor Variants

Table 6.7 summarizes the selected features of each processor. This heterogeneity allows us to build an efficient reprogrammable pipeline. The following sections summarize the features of each processor and motivates their selection.

Table 6.7. Summary of the features of each processor in Figure 6.7

Name	ICache	DCache	Forwarding	Atomic	Mult.	FIFO Extensions	Min/Max	LUT
Head (H)	Y, 2KB	Y, 4KB	Y	Y	N	OUT	N	N
Map (M)	N	N	N	N	Y	SRC1, OUT	Y	Y
Reduce (R)	N	N	N	N	Y	SRC1, SRC2, OUT	Y	Y
Spine (S)	N	N	`rs2` Only	N	Y	SRC1, OUT	Y	Y
Tail (T)	Y, 2KB	Y, 4KB	`rs1` Only	Y	N	SRC2	N	N

Head Processor (H)

The Head processors are annotated with an (H) in Figure 6.7. The Head processors are are optimized for reading data and transferring it to the Map processors using the OUT FIFO extension.

Each Head processor has a 4KB direct-mapped data cache with a 1024-byte line size that amortizes cache miss latency for linear access patterns, and a 4KB direct-mapped instruction cache with a 16-byte line size. Both caches are connected to main memory via the L2 cache. The Head processor implements the OUT FIFO extension to send data to the Map processors, and implements the atomic extension to coordinate with the Tail processor. It does not have Multiply or Min/Max extensions since it is not expected to do computation. The processor supports full forwarding to avoid RAW hazards and improve the penalty of pointer updates and loop-branch calculations.

Figure 6.14 demonstrates how the Head processors act like a streaming direct memory access engine for the architecture in Figure 6.7. On Line 2 the address `0x4000` is loaded into `x1`. The `OUT_EN` bit is set on Line 4. Lines 6-9 load bytes into `x2`-`x5`, and store those bytes in OUT FIFO. These bytes are read by a Map processor.

The Head Processor and L2 Cache are intended to replicate the 2D Buffer and shift register structures that are common in prior work [90, 91]. The multiplexers necessary to implement these schemes are costly on FPGAs. We believe a collection of highly-optimized soft processors fetching data will provide more flexibility in memory access patterns without the

```
1  head_demo:
2          li x1, 0x4000           ;; Load 0x4000 into x1
3
4          csrwi fifoctrl, 0x40    ;; Set OUT_EN
5
6          lb x2, 0(x1)            ;; Read byte 0(x1) and write it to OUT FIFO
7          lb x3, 1(x1)            ;; Read byte 1(x1) and write it to OUT FIFO
8          lb x4, 2(x1)            ;; Read byte 2(x1) and write it to OUT FIFO
9          lb x5, 3(x1)            ;; Read byte 3(x1) and write it to OUT FIFO
10
11         csrwi fifoctrl, 0x0     ;; Clear all bits
```

Figure 6.14. An example demonstrating `lb` instructions and OUT FIFO on the Head processor. Each `lb` writes the loaded data into OUT FIFO and `rd`

added comlexity of custom instructions or hardware multiplexers.

Map Processor (M)

The Map processors are annotated with an (M) in Figure 6.7. The Map processors are are optimized for applying an operation to data from the Head processor via SRC1 FIFO and writing the result to OUT FIFO for the Reduce processors.

Each Map processor has a 4KB memory space with a 1-cycle access latency. The memory space is shared between instructions and data and is separate from host memory. The Map processor implements multiplication, Min/Max, LUT, and the SRC1 and OUT extensions. The processor does not implement forwarding since it is not useful for the computations we expect.

Figure 6.15 demonstrates how the Map processor can be used to read data from the SRC1 FIFO, apply arithmetic operations, and write the data to OUT FIFO. Line 2 loads the constant 42 into `x1`. Line 4 sets the `SRC1_EN` and `OUT_EN` bits so that arithmetic instructions read from SRC1 FIFO and write to OUT FIFO. Line 5 demonstrates this behavior – instead of reading the undefined value in `x2`, it adds 1 to the value in SRC FIFO. Similar behavior is demonstrated on Line 9, where the `mul` instruction multiplies the next value in SRC1 FIFO by the value in `x1` (42). The results of both instructions are implicity written to OUT FIFO.

Map processors perform operations on data from the Head processors using immediates and arithmetic instructions, a useful operation for implementing convolution-like computations.

```
1  map_demo:
2          li x1, 42
3
4          csrwi fifoctrl, 0x41     ;; Set SRC1_EN and OUT_EN
5          addi x3, x2, 1           ;; Read SRC1 FIFO and add 1
6                                   ;; Implicitly read SRC1 FIFO and add 1
7                                   ;; Write the result to OUT FIFO
8
9          mul x5, x4, x1           ;; Implicitly read SRC1 FIFO and
10                                  ;; multiply it by 42 (x1)
11                                  ;; Write the result to OUT FIFO
12
13         csrwi fifoctrl, 0x00     ;; Clear SRC1_EN and OUT_EN
```

Figure 6.15. An example demonstrating the SRC1 and OUT FIFOs on the Map processor to apply two arithmetic operations.

Reduce Processor (R)

The Reduce processors are annotated with an (R) in Figure 6.7. Reduce processors are implemented to "reduce" results from the Map processors. The result is written to the Spine processors for final processing.

Each Reduce processor has a 4KB memory space with a 1-cycle access latency. The memory space is shared between instructions and data. The Reduce processor implements the RISC-V Multiply extension, our Min/Max extension, our LUT extension, and the the SRC1, SRC2, and OUT FIFO extensions. The Reduce processor does not have forwarding logic; accumulates are handled in the Spine processors described next.

Figure 6.16 demonstrates how Reduce processors can "reduce" inputs from SRC1 and SRC2 FIFOs and write the result to OUT FIFO. Line 2 enables the SRC1, SRC2, and OUT FIFOs so that arithmetic instructions read from SRC1 and SRC2 FIFOs and the result is written to OUT FIFO. Line 5 adds together two values from SRC1 and SRC2 FIFO and writes the result to OUT FIFO and `x3`. Similar behavior is shown on Line 8, where the `or` instruction reads from SRC1 and SRC2 FIFOs, applies a logical or operation and writes the result to OUT FIFO and `x6`

The Reduce processors can perform a variety of useful operations to collectively transform an array of parallel inputs into a single output. They can find the local minimum or maximum in a window for non-linear filters, they can compute an arctangent approximation for edge detection,

```
reduce_demo:
        csrwi fifoctrl, 0x49    ;; Set SRC1_EN, SRC2_EN, OUT_EN

        add x3, x2, x1          ;; Implicitly read values from SRC1 FIFO and
                                ;; SRC2 FIFO, add them together and store the
                                ;; result in OUTFIFO (x3 is modified as well)

        or x6, x5, x4           ;; Implicitly read values from SRC1 FIFO and
                                ;; SRC2 FIFO, or them together and store the
                                ;; result in OUT FIFO (x6 is modified as well)


        csrwi fifoctrl, 0x0     ;; Clear all bits
```

Figure 6.16. An example demonstrating how the Reduce processor can add two values from SRC1 FIFO and SRC2 FIFO, and write the result to OUT FIFO

and they can produce the sum of a window for linear filters.

Spine Processor (S)

The Spine processors are annotated with a (S) in Figure 6.7. Spine processors are designed to perform streaming 1-D computations to transform data output by the network of Reduce processors.

Each Spine processor has a 4KB memory space with a 1-cycle access latency. The memory space is shared between instructions and data. The Spine processor implements the RISC-V Multiply extension, our Min/Max extension, our LUT extension, the SRC1, and OUT FIFO extensions. The Spine processor implements forwarding logic for `rs2` operand RAW hazards. This allows it to implment multi-cycle accumulates.

Spine processors are useful for performing any final computations before data is stored in memory. For example, the Spine processors can apply saturation, or round fixed point values.

Tail Processor (T)

The Tail processors are annotated with a (T) in Figure 6.7. Tail processors are designed to store computation results to memory.

The tail processor has a 2KB direct-mapped, write-back data cache with a 256-byte line size and 2KB direct-mapped instruction cache. Both caches are connected to main memory via the L2 cache. The Tail processor implements the SRC2 FIFO and atomic extensions. The

tail processor uses the SRC2 FIFO extension because RISC-V store instructions store data from register `rs2` and use `rs1` for the base address. When when the `SRC2_EN` bit of `fifoctrl` CSR is set data from the SRC2 FIFO is read and stored using storage instructions. It does not implement multiply, LUT, or Min/Max extensions since those are not related to data storage. The Hazard unit implements forwarding on the `rs1` operand to improve the throughput of pointer calculations, but does not predict branches.

Figure 6.17 demonstrates how Tail processors act like a streaming DMA write engine by reading results from SRC2 FIFO and storing the result in host memory. On Line 2 the `x1` loads the address `0x4000`. Line 4 sets the `SRC2_EN` bit which causes store instructions to implicitly read SRC2 FIFO. Lines 6-13 store a series of bytes from SRC2 to addresses `0x4000`-`0x4003`.

```
1  tail_demo:
2          li x1, 0x4000            ;; Load 0x4000 into x1
3
4          csrwi fifoctrl, 0x8      ;; Set SRC2_EN
5
6          sb x2, 0(x1)             ;; Implicitly read SRC2 FIFO and
7                                   ;; write the first byte to address 0x4000 (0 + x1)
8          sb x3, 1(x1)             ;; Implicitly read SRC2 FIFO and
9                                   ;; write the first byte to address 0x4001 (1 + x1)
10         sb x4, 2(x1)             ;; Implicitly read SRC2 FIFO and
11                                  ;; write the first byte to address 0x4002 (2 + x1)
12         sb x5, 3(x1)             ;; Implicitly read SRC2 FIFO and
13                                  ;; write the first byte to address 0x4003 (3 + x1)
14
15         csrwi fifoctrl, 0x0      ;; Clear SRC2_EN
```

Figure 6.17. An example demonstrating the use of SRC2 FIFO on the Tail processor

6.3.4 Area Estimate

Table 6.8 shows our estimated resource consumption per processor for Block RAMS (BRAMS) and DSPs on a Xilinx FPGA. The Xilinx FPGA architecture provides 36 KBit BRAMs that can be fractured into two independent 18 KBit BRAMs. A 36 KBit BRAM is counted as 2 BRAM, and a 18 KBit BRAM is counted as 1. We estimate BRAM and DSP consumption using which is derived from our architecture description in 6.3.1.

Table 6.9 demonstrates the processor composition of six architecture sizes from 2 Head processors ($N_H = 2$) to 64 Head processors ($N_H = 64$). We have chosen to use 5 Spine processors

Table 6.8. Estimated DSP and BRAM consumption for all processor types

Processor Type	Instr. BRAMs	Data BRAMs	FIFO BRAMs	DSPs
Head	1	2	1	0
Map	1	1	1	1
Reduce	1	1	1	1
Spine	1	1	1	1
Tail	1	2	1	0

to implement an efficient `medianBlur` function as described in our Results. The right-most column in the table displays the total number of processors in each architecture.

Table 6.9. Processor counts for six architecture sizes, from 2 Head processors ($N_H = 2$) to 64 Head processors ($N_H = 64$).

N_H	Head	Map	Reduce	Spine	Tail	Total Processors
2	2	2	1	5	1	11
4	4	4	3	5	1	14
8	8	8	7	5	1	22
16	16	16	15	5	1	38
32	32	32	31	5	1	70
64	64	64	63	5	1	134

Table 6.10 estimates the resource consumption of BRAMs and DSPs for the same six architecture sizes. The data displayed in this table is the product of the processor estimates in Table 6.8 and the processor counts in Table 6.9. As described above the Xilinx FPGA architecture contains 36 KBit BRAMs that can be fractured into two 18 KBit BRAMs. Column 2 shows the maximum possible BRAM consumption where all 18 KBit BRAMs use a 36 KBit BRAM. Column 3 reports the minimum BRAM consumption where all 18KBit BRAMs share a 36 KBit BRAM. The actual BRAM consumption will be somewhere between the two numbers.

Table 6.10. Estimated DSP and BRAM resource consumption for 6 architecture sizes

N_H	Total BRAMs (Max.)	Total BRAMs (Min.)	Total DSPs
2	68	36	8
4	86	44	12
8	134	68	20
16	230	116	36
32	422	212	68
64	806	404	132

6.4 Results

We report quantitative results for 12 OpenCV functions implemented on an ARM Processor (with and without NEON instructions), an FPGA Pipeline, and our proposed architecture. The experimental setup for each architecture is described in Section 6.4.1. OpenCV functions and implementations are described in Section 6.4.2. Finally, we report performance results with analysis in Section 6.4.3

6.4.1 Experimental Setup

We have gathered processor results two devices: a Pynq Z1 with a Zynq-7000 SoC, and an Ultra96 board with a Zynq Ultrascale+ SoC. The Zynq-7000 device features a dual-core Cortex-A9 32-bit processor with NEON instructions running at 650 MHz. The Pynq-Z1 provides 512 MB of DDR3 operating at 533 MHz and a 16-bit DDR interface for a theoretical maximum throughput of of 2 GBps. The Zynq Ultrascale+ chip features a quad-core Cortex-A53 64-bit processor with NEON instructions operating at 1200 MHz. The system provides 2 GB of LPDDR4 memory operating at 533 MHz with a 32-bit DDR interface for a theoretical maximum throughput of 4 GBps. Both devices have on-die FPGA fabrics with characteristics shown in Table 6.11.

We collect results from 12 functions in the OpenCV library: `meanStdDev`, `integral`, `minMaxLoc`, `add (8-bit)`, `multiply (16-bit)`, `cvtColor`, `gaussianBlur`, `medianBlur`,

Table 6.11. Device resources and maximum frequencies for Zynq-7000 and Zynq Ultrascale+

Device	Total BRAMs	Total DSPs	DSP Max Freq.	BRAM Max Freq.
Zynq-7000	140	220	463 MHz	400 MHz
Zynq Ultrascale+	216	360	571 MHz	645 MHz

`Canny`, and `bilateralFilter`. We ran these functions on our test set of 10 images shown in Figure 6.18. Each image measures 1280 pixels by 960 pixels, larger than the 512 KB L2 cache on the Zynq-7000, and larger than the 1MB L2 on the Zynq Ultrascale+ device. All functions operate on 8-bit grayscale pixels except `multiply` which operates on 16-bit pixel values.

Figure 6.18. Test images used in our experiments

Measurements were gathered in Python 3.6 using the `timeit` library. The library records execution times for each run. We ran each function 200 times for each image, and the first 100 iterations were discared. We report the maximum throughput (minimum execution time) from the last 100 iterations.

ARM Processors

Arm processor results were gathered running from an unmodified version of OpenCV 3.4 compiled locally on each board. The Zynq-7000 device uses g++ 5.2.1 and Ubuntu 15.10. The Zynq MPSoC uses g++ 5.3.1 and Ubuntu 16.04 LTS. For each device, we report two OpenCV results: **ARM With Neon** are results from OpenCV compiled with NEON SIMD instructions;

ARM Without Neon are results from OpenCV compiled without NEON SIMD instructions. We report our results in Section 6.4.2.

FPGA Pipelines

We report hardware pipeline performance results using implementations from the Xilinx XFOpenCV 2017.4 library. The XFOpenCV Library provides two throughput options: 1 pixel per cycle (**FPGA** $P = 1$) and 8 pixels per cycle (**FPGA** $P = 8$). The functions `integral`, `cvtColor`, and `bilateralFilter` are restricted to $P = 1$ in the XFOpenCV library.

We report maximum frequencies for each function in Table 6.12. We implemented each throughput option in Vivado 2017.4 with default synthesis and implementation goals and used a binary search pattern to determine the maximum frequency. The throughput results in Section 6.4.2 use these maximum frequency bitstreams.

Table 6.12. Compilation Results for our 12 selected OpenCV Functions on Zynq-7000 and Zynq Ultrascale+ devices

	Zynq-7000 Max Frequency		Zynq Ultrascale+ Max Frequency	
Function Name	$P = 1$	$P = 8$	$P = 1$	$P = 8$
`meanStdDev`	179.69 MHz	177.34 MHz	351.95 MHz	318.75 MHz
`minMaxLoc`	192.97 MHz	210.55 MHz	312.11 MHz	317.97 MHz
`add`	192.63 MHz	178.13 MHz	366.02 MHz	394.53 MHz
`multiply`	187.50 MHz	201.56 MHz	309.77 MHz	303.52 MHz
`integral`	183.50 MHz	N/A	363.28 MHz	N/A
`cvtColor`	184.38 MHz	N/A	359.38 MHz	N/A
`gaussianBlur` (3x3)	145.70 MHz	141.41 MHz	262.11 MHz	246.88 MHz
`gaussianBlur` (5x5)	189.06 MHz	201.56 MHz	297.60 MHz	275.78 MHz
`gaussianBlur` (7x7)	191.40 MHz	200.00 MHz	301.55 MHz	264.85 MHz
`medianBlur`	191.41 MHz	176.56 MHz	357.42 MHz	344.14 MHz
`Canny`	177.34 MHz	193.75 MHz	350.39 MHz	378.52 MHz
`bilateralFilter`	155.08 MHz	N/A	300.00 MHz	N/A
Mean	180.89 MHz	186.76 MHz	327.38 MHz	315.87 MHz
Standard Deviation	15.23 MHz	21.07 MHz	33.78 MHz	53.51 MHz

Table 6.12 demonstrates that most Zynq designs achieve 180 MHz, and nearly all Zynq Ultrascale designs achieve 300. The exceptions are `gaussianBlur` (3x3) on all architectures, `gaussianBlur` (5x5) and (7x7) at $P = 8$ on Zynq Ultrascale+, and `bilateralFilter` on the

Zynq-7000 at $P = 1$. We traced these deviations back to the projected timing in Vivado HLS, but we were unable to determine the root cause. These anomalies persist across computers, compilation runs, and sythesis goals.

RISC-V Tree

We estimate results for our architecture at $N_H = 8$, $N_H = 16$, and $N_H = 32$, where N_H is the number of head processors in our architecture. We estimate that these are feasible architectures based on the resources available in Table 6.11 and the resource consumption estimates in Table 6.10. Such results based on the 80-processor result on the same Zynq-7000 device in [88].

We report results for our architecture at 150 MHz for the Zynq-7000 device, and 300 MHz for the Zynq Ultrascale device. These estimates are 30 MHz below the mean maximum frequency for the Zynq-7000 device and 27 MHz below the mean maximum frequency for the Zynq Ultrascale+ device from Table 6.12, and correspond well to [88].

We compute performance results using a modified version of Equation 6.1 for throughput:

$$T = \frac{1}{t} = \frac{E * F * P}{II} \tag{6.2}$$

And, the period:

$$t = \frac{II}{E * F * P} \tag{6.3}$$

Where E is the the efficiency, a value between 1.0 and 0. Efficiency of computations on our architecture is determined by how often the Head processor stalls when read data from memory, and the cache miss penalty. We have determined that the memory read latency is between 30 and 50 cycles on the Zynq-7000 and Zynq Ultrascale+ devices using the XFOpenCV Functions. Assuming memory is read linearly with a cache size of 1024 bytes, $E = 95\%$ for 1-byte reads, $E = 90\%$ for 2-byte reads, and $E = 80$ for 4-byte reads. Our benchmarks mainly read 1-byte and 2-byte values so we assume $E = 90\%$.

Some OpenCV functions are composed from multiple sub-functions, called *Passes*, and each pass may have a uniqe Initiation Interval (II) and parallel output (P). To compute the total throughput for a multi-pass function we use Equation 6.4, derived from Equation 6.2.

$$T_{function} = \frac{1}{\sum_{i=1}^{i=\text{Passes}} t_i} = \frac{1}{\sum_{i=1}^{i=\text{Passes}} \frac{II_i}{P_i * E * F}} \tag{6.4}$$

This equation states that the throughput of an OpenCV function on our architecture is equal to the inverse of the sum of the periods of the passes. The period of each pass is computed using Equation 6.3 using the tables in Section 6.4.2.

6.4.2 OpenCV Functions

We have selected 12 common OpenCV functions for implementation. The functions have diverse computational patterns so we can understand how our architecture compares to custom hardware pipelines, and an ARM ISA. In the following sections we briefly describe each function, and provide a detailed description of how the function and its passes map to our proposed architecture.

Image Mean & Std. Deviation (`meanStdDev`))

The `meanStdDev` function computes the Mean and Standard Deviation of the pixels in an image.

Our proposed architecture computes the result in two passes, μ and *sigma*, as shown in Table 6.13. The first pass computes the sum of all pixels in an image using a sliding column window, where the height of the column is equal to the number of head processors. The ARM processor is used to compute the quotient to determine the mean since our architecture does not support divide. The second pass computes the total variance. Two Map processors simultaneously compute the deviation of each pixel, and the first Reduce stage computes the product to square the result. The remaining Reduce processors compute the sum. Finally, the ARM processora is used to compute the quotient and root.

Table 6.13. Breakdown of images passes to compute the function `meanStdDev` on different sizes of our architecture

N_H	Passes	II_μ	P_μ	II_σ	P_σ
8	2	1	8	1	2
16	2	1	16	1	4
32	2	1	32	1	8

Image Min/Max Location (`minMaxLoc`)

The `minMaxLoc` function determines the value and location of the minima and maxima in an image.

Table 6.14 shows how our architecture computes the result in three passes, `min`, `max`, and `location`. The first and second pass compute the maximum and minumum value in a sliding column where the height of the column is equal to N_H. On the third and final pass, the architecture iterates through the image to find the minumum and maximum locations.

Table 6.14. Breakdown of images passes to compute the function `minMaxLoc` on different sizes of our architecture

N_H	Passes	II_{min}	P_{min}	II_{max}	P_{min}	$II_{location}$	$P_{location}$
8	3	1	8	1	8	1	1
16	3	1	16	1	16	1	1
32	3	1	32	1	32	1	1

Image Add (`add`)

The `add` function computes the pixel-wise sum of two images. In our tests, each input is an 8-bit greyscale image, and the output is a saturated 8-bit pixel value.

Table 6.15 demonstrates how our architecture computes the result in one pass. The head processors load 2 or 4 pixels from each of the two input images. The first Reduce processor computes the summation and saturation using the output Look-Up-Table. The second level of

reduce processors shift the result. The remaining Reduce processors form an or-reduction tree to create a 2-result, or 4-result output.

Table 6.15. Breakdown of images passes to compute the function `add` on different sizes of our architecture

N_H	Passes	II_{add}	P_{add}
8	1	1	2
16	1	1	4
32	1	1	4

Image Multiply (`multiply`)

The `multiply` function computes the pixel-wise product of two images. In our tests, each input is an 8-bit greyscale image, and the output is a 16-bit pixel value that does not need to be checked for saturation.

Table 6.16 demonstrates how our architecture computes the result in one pass. In all architectures, two pairs of Head processors load corresponding pixels from the source images. The first Reduce processor computes the product. The second level of Reduce processors shift the result by 16 bits if neccsary. The remaining Reduce processors create an or-reduction tree.

Table 6.16. Breakdown of images passes to compute the function `multiply` on different sizes of our architecture

N_H	Passes	$II_{multiply}$	$P_{multiply}$
8	1	1	1
16	1	1	2
32	1	1	2

Image Integral (`integral`)

`integral` computes the integral image of a function, where each pixel in the integral image is equal to the sum of all pixels to the left and above the current pixel.

Our architecture computes the result in one pass, as shown in Table 6.17. Two Head processors load the left-up-diagonal pixel of the previous result row, and the up pixel of the previous result row. The Reduce processors add these values. The first Spine processor simultaneously accumulates the result and sends the current result to the tail processor.

Table 6.17. Breakdown of images passes to compute the function `integral` on different sizes of our architecture

N_H	Passes	$II_{integral}$	$P_{integral}$
8	1	1	1
16	1	1	1
32	1	1	1

Color Conversion (`cvtColor`)

The `cvtColor` function computes colorspace transformation from the RGB colorspace to the YUV colorspace.

Our architecture computes the result in three passes as shown in Table 6.18. Each pass computes a single output channel. The computations are fixed point and normalized by the Spine processors.

Table 6.18. Breakdown of images passes to compute the function `cvtColor` on different sizes of our architecture

N_H	Passes	II_Y	P_Y	II_U	P_U	II_V	P_V
8	3	1	1	1	1	1	1
16	3	1	1	1	1	1	1
32	3	1	1	1	1	1	1

Gaussian Filter (`gaussianBlur`)

`GaussianBlur` computes the result of a 2-D Gaussian kernel convolved with an image. We report results for a 3x3 kernel, a 5x5 kernel, and a 7x7 kernel.

Our architecture computes the result of this function in one pass, as shown in Table 6.19. However, different sizes of our architecture have different initiation intervals. For example, the architecture $N_H = 8$ is not large enough to execute the full 9-element stencil multiply on a single cycle and must use two cycles for $II_G = 2$.

Table 6.19. Breakdown of images passes to compute the function `gaussianBlur` on different sizes of our architecture

Gaussian 3x3

N_H	Passes	II_G	P_G
8	1	2	1
16	1	1	1
32	1	1	1

Gaussian 5x5

N_H	Passes	II_G	P_G
8	1	4	1
16	1	2	1
32	1	1	1

Gaussian 7x7

N_H	Passes	II_G	P_G
8	1	8	1
16	1	4	1
32	1	2	1

Median Filter (`medianBlur`)

`medianBlur` computes the result of a 2-D non-linear median filter applied to an image. The result at each pixel is the median of all of the pixels in a 3x3 window surrounding the pixel.

Our architecture computes the result of this function in one pass, as shown in Table 6.20. Our architecture computes the median by implementing an insertion sort network [54] using 5 spine processors. However, this means an II of 9 for a 3x3 window since each pixel has to be set into the "Sorting Network" on each iteration.

Table 6.20. Breakdown of images passes to compute the function `medianBlur` on different sizes of our architecture

N_H	Passes	II_{median}	P_{median}
8	1	9	1
16	1	9	1
32	1	9	1

Canny Filter (`Canny`)

The `Canny` function is a popular non-linear edge detection algorithm [105]. The function highlights "edges" by detecting regions of with changes in image intensity.

The canny function is computed in six passes: *Gauss* executes a 5x5 `gaussianBlur` filter, G_x, G_y computes the X and Y gradients using 3x3 derivative kernels, *atan* computes the angle between the G_x and G_y gradients, $\|G\|$ computes the magnitude of the gradient, *NM* performs non-max supression, and finally θ performs thresholding.

The subfunctions of the canny filter are quite complex, as shown in Table 6.21.

Table 6.21. Breakdown of images passes to compute the function `Canny` on different sizes of our architecture

N_H	Passes	II_{Gauss}	P_{Gauss}	II_{G_x,G_y}	P_{G_x,G_y}	II_{atan}	P_{atan}	$II_{\|G\|}$	$P_{\|G\|}$	II_{NM}	P_{NM}	II_{θ}	P_{θ}
8	6	4	1	4	1	2	1	1	1	2	1	1	1
16	6	2	1	2	1	1	1	1	1	1	1	1	1
32	6	1	1	1	1	1	1	1	1	1	1	0	1

Bilateral Filter (`bilateralFilter`)

`bilateralFilter` is a popular non-linear, edge-preserving, de-noising filter developed by [106]. For each pixel, the function computes the following:

$$O(x) = \frac{1}{W_p} \sum_{x_i \in \Omega} I(x) f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|) \tag{6.5}$$

Where the normalization term W_p is:

$$W_p = \sum_{x_i \in \Omega} f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|) \tag{6.6}$$

And:

- O is the output image
- I is the input image
- x is the coordinate of the current pixel in the image
- Ω is the window centered on x
- f_r is a gaussian function for differences in pixel intensity
- g_r is a gaussian function for differences in pixel coordinates

In our results we test performance of a 3x3 `bilateralFilter`. Our architecture can compute this function in three passes as shown in Table 6.22. The first pass computes the numerator: Head processors load image pixels in the 3x3 window. Two Head procesors feed $I(x_I)$ and $I(x)$ to a Reduce processor, which subtracts them and uses the LUT to compute $f_r(\|I(x_i) - I(x)\|)$. Adjacent head processors load I_x and compute $g_s(\|x_i - x\|)$, and pass them to the first Reduce stage which multiplies them. The subsequent reduce stages produce the sum. $II_{Numerator}$ is multiple cycles because each window input requires 4 processors meaning $N_H \geq 36$ processors for the whole window to be computed with an II of 1. The denominator W is computed in similar fashion. The final pass computes the quotient of the numerator and the denominator by inverting the denominator and multiplying.

6.4.3 Results

Zynq-700 MPSoC

Performance results for the Zynq-7000 device are displayed in Figure 6.19. The vertical axis measures throughput in Frames per Second (FPS), and the horizontal axis shows the function

Table 6.22. Breakdown of images passes to compute the function `bilateralFilter` on different sizes of our architecture

N_H	Passes	$II_{Numerator}$	$P_{Numerator}$	II_W	P_W	II_{Divide}	P_{Divide}
8	3	5	1	5	1	1	1
16	3	3	1	2	1	1	1
32	3	2	1	2	1	1	1

name sorted by `ARM without NEON` performance. Processor performance results for the ARM core are shown in two shades of blue, for our architecture in three shades of green, and for two FPGA pipelines in two shades of orange.

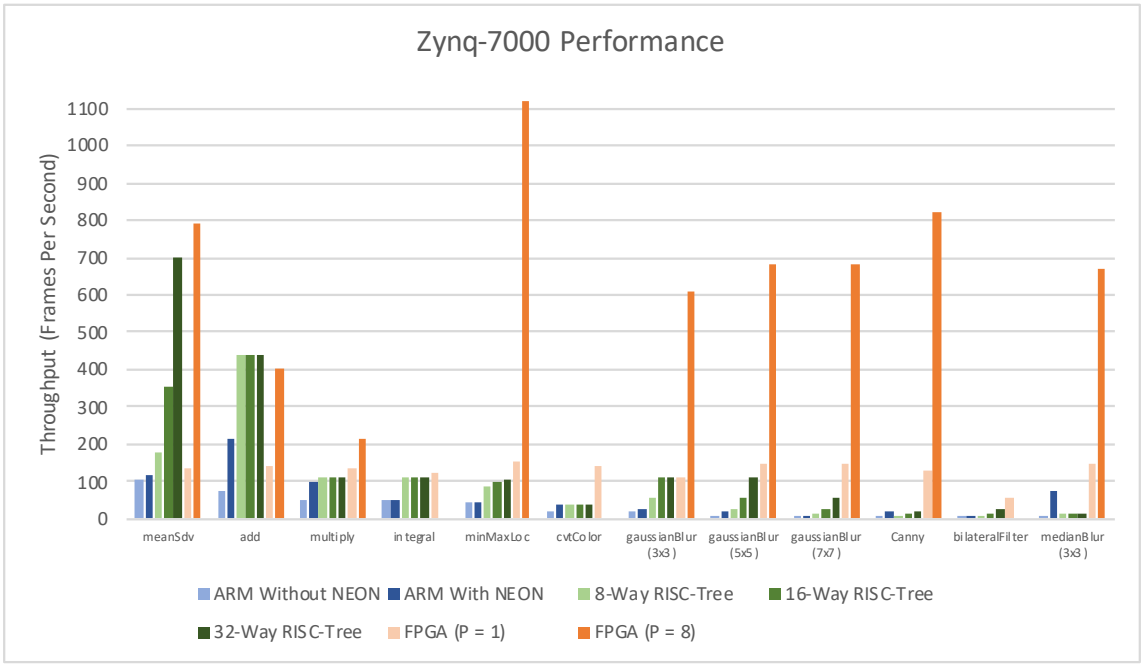


Figure 6.19. Throughput results for 12 OpenCV Functions on a Zynq-7000 device. The horizontal axis is sorted by the ARM Without Neon Result

Table 6.23 demonstrates the same results as Figure 6.19 in table form. The first two columns record ARM processor results without and with NEON instructions, the third and fourth columns show FPGA results with $P = 1$ and $P = 8$, and the final columns present results for our architecture with $N_H = 8$, $N_H = 16$, and $N_H = 32$ where N_H is the number of head processors in

our proposed architecture.

Table 6.23. Throughput results for our 12 selected OpenCV Functions on a Zynq-7000 device.

Function Name	ARM w/o NEON	ARM w/ NEON	FPGA ($P = 1$)	FPGA ($P = 8$)	$N_H = 8$	$N_H = 16$	$N_H = 32$
`meanStdDev`	102 FPS	116 FPS	138 FPS	792 FPS	176 FPS	352 FPS	703 FPS
`add`	76 FPS	215 FPS	141 FPS	404 FPS	439 FPS	439 FPS	439 FPS
`multiply`	53 FPS	102 FPS	133 FPS	216 FPS	110 FPS	110 FPS	110 FPS
`integral`	48 FPS	48 FPS	121 FPS	N/A	110 FPS	110 FPS	110 FPS
`minMaxLoc`	46 FPS	46 FPS	154 FPS	1122 FPS	88 FPS	98 FPS	103 FPS
`cvtColor`	19 FPS	37 FPS	140 FPS	N/A	37 FPS	37 FPS	37 FPS
`gaussianBlur` 3x3	19 FPS	27 FPS	112 FPS	609 FPS	55 FPS	110 FPS	110 FPS
`gaussianBlur` 5x5	10 FPS	20 FPS	144 FPS	682 FPS	27 FPS	55 FPS	110 FPS
`gaussianBlur` 7x7	7 FPS	7 FPS	146 FPS	681 FPS	14 FPS	27 FPS	55 FPS
`Canny`	6 FPS	17 FPS	131 FPS	823 FPS	8 FPS	14 FPS	22 FPS
`bilateralFilter`	4 FPS	4 FPS	59 FPS	N/A	10 FPS	16 FPS	27 FPS
`medianBlur`	3 FPS	73 FPS	146 FPS	668 FPS	12 FPS	12 FPS	12 FPS

We highlight several trends in our Zynq-7000 data:

First, the throughput of ARM (Without Neon) decreases as the function complexity increases. Functions like `add` and `meanSdv` have low *II* loops. Multi-stage filters like `bilateralFilter` and `Canny` have high *II* loops. This is consistent with our graph in Figure 6.6

Second, ARM (With Neon) throughput is comparable to ARM (Without NEON) throughput on convolution-like functions. Convolution functions like `gaussianBlur`, `Canny` and `bilateralFilter` do not benefit from NEON instructions. In contrast, functions like `add` and `multiply` that have low *II* loops with obvious parallelism benefit from vectorized instructions. This is consistent with analysis in previous work described in Section 6.2

Fourth, our architecture beats ARM throughput in most cases. The exception is `medianBlur`, which uses NEON instructions to create a sorting network.

Third, the throughput of our architecture decreases with the number of *stages* in a function. Single stage functions like `meanSdv`, `add`, and `multiply` have throughput that is comparable to the XFOpenCV. Multi-stage functions like `Canny`, `bilateralFilter` `cvtColor` have reduced throughput. Our definition of a *stage* is described in Section 6.4.2.

Fifth, the performance of XFOpenCV hardware is constant across OpenCV functions when $P = 1$. The only exception is `bilateralFilter`, which has a high initiation interval for the division operation shown in Equation 6.5. This trend is consistent with our graph in Figure 6.6 where hardware performance is a function of the operation with the largest II in a function.

Sixth, the performance of XFOpenCV hardware is bandwidth limited when $P = 8$. The maximum theoretical bandwidth of the Zynq-7000 device is 2 GBps but is shared between ports of an OpenCV function. `minMaxLoc` and `meanSdv` have one read port, and have the highest performance. `meanSdv` has lower performance because it is written (unnecessarily) as two nested loops which causes the HLS compiler to introduce overheads that we can see on an ILA core. `gaussianBlur`, `Canny`, and `medianBlur` each have one read port and one write port and have approximately half the frame-rate of `minMaxLoc`. The `add` function has three 8-bit ports, and the `multiply` has three 16-bit ports. The number of ports increases contention and lowers performance.

Zynq Ultrascale+

Performance results for the Zynq Ultrascale+ device are displayed in Figure 6.19. The vertical axis measures throughput in Frames per Second (FPS), and the horizontal axis shows the function name sorted by baseline ARM performance (without NEON). Processor performance results for the ARM core are shown in two shades of blue, for our architecture in three shades of green, and for two FPGA pipelines in two shades of orange.

Table 6.24 records the results shown in Figure 6.20. The first two columns record ARM processor results without and with NEON instructions, the third and fourth colum show FPGA results with $P = 1$ and $P = 8$, and the final columns present results for our architecture with $N_H = 8$, $N_H = 16$, and $N_H = 32$ where N_H is the number of head processors in our proposed architecture.

We highlight several trends in our Zynq Ultrascale+ data:

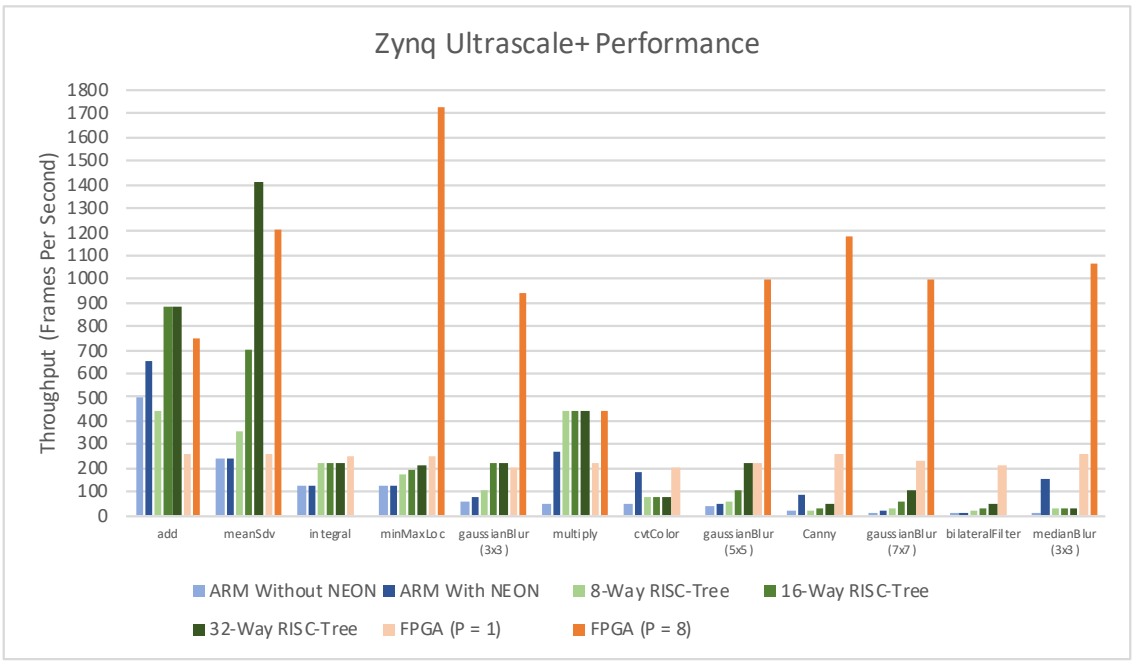


Figure 6.20. Throughput results for 12 OpenCV Functions on a Zynq Ultrascale+ Chip.

Table 6.24. Throughput results for our 12 selected OpenCV Functions on a Zynq Ultrascale+ device.

Function Name	ARM w/o NEON	ARM w/ NEON	FPGA ($P = 1$)	FPGA ($P = 8$)	$N_H = 8$	$N_H = 16$	$N_H = 32$
`add`	495 FPS	656 FPS	259 FPS	752 FPS	439 FPS	879 FPS	879 FPS
`meanStdDev`	239 FPS	239 FPS	261 FPS	1204 FPS	352 FPS	703 FPS	1406 FPS
`integral`	128 FPS	128 FPS	248 FPS	N/A	220 FPS	220 FPS	220 FPS
`minMaxLoc`	121 FPS	121 FPS	248 FPS	1725 FPS	175 FPS	195 FPS	207 FPS
`gaussianBlur` 3x3	57 FPS	75 FPS	197 FPS	941 FPS	110 FPS	220 FPS	220 FPS
`multiply`	52 FPS	270 FPS	218 FPS	437 FPS	110 FPS	220 FPS	220 FPS
`cvtColor`	51 FPS	185 FPS	198 FPS	N/A	73 FPS	73 FPS	73 FPS
`gaussianBlur` 5x5	40 FPS	53 FPS	221 FPS	995 FPS	55 FPS	110 FPS	220 FPS
`Canny`	15 FPS	83 FPS	258 FPS	1178 FPS	16 FPS	27 FPS	44 FPS
`gaussianBlur` 7x7	13 FPS	15 FPS	231 FPS	994 FPS	27 FPS	55 FPS	110 FPS
`bilateralFilter`	7 FPS	11 FPS	208 FPS	N/A	20 FPS	31 FPS	44 FPS
`medianBlur`	6 FPS	154 FPS	263 FPS	1064 FPS	24 FPS	24 FPS	24 FPS

First, ARM results are consistent with Section 6.4.3. Similar to above, the throughput of ARM (Without Neon) decreases as the function complexity increases. In addition ARM (With Neon) througput is comparable to ARM (Without NEON) throughput on convolution-like

functions. The one exception is `Canny` which greatly out-performs all hardware except FPGA pipelines.

Second, our architecture outperforms fewer ARM (With NEON) functions. In addition to `medianBlur`, `Canny` and `cvtColor` now outperform our architecture as well. In addition, ARM functions like `add` and `multiply` have higher performance.

Finally, FPGA hardware performance is characteristically simliar to the Zynq-7000. The performance of XFOpenCV hardware is constant across OpenCV functions when $P = 1$ and bandwidth limited when $P = 8$. The Ultra96 board has double the memory bandwidth so all kernels improve but only `minMaxLoc` increases by a factor of 8 from $P = 1$ to $P = 8$.

6.4.4 Analysis

We draw several conclusions from our results:

First, we corroborate results in [41, 91] and confirm that modern architectures are inefficient on convolution operations. ARM (With NEON) throughput and ARM (Without NEON) throughput is approximately equal for all convolution operations in Figure 6.19 and Figure 6.20. Our results We also confirm that hardware accelerators can be used to out-perform modern processors even with $P = 1$.

ARM (With NEON) throughput is improving across generations for simple functions. The throughput for low *II* operations like `add` and `meanSdv` and `minMaxLoc` improved relative to our architecture from the Zynq-7000 to the Zynq Ultrascale+ generation.

Our proposed architecture is more efficient on convolution functions than an ARM processor. We outperform the ARM processor on `gaussianBlur`, `bilateralFilter` functions on both devices, and `Canny` on the Zynq-7000. Our architecture cannot efficiently implement `medianBlur`.

Further increasing N_H may not greatly improve performance on these kernels. N_H is similar to increasing the width of vector instructions, and may help support larger convolution sizes (5x5, 7x7, and even 9x9). Increasing N_H exponentially increases the number of processors,

and increases memory contention even at $N_H = 8$.

Instead, The largest benefits may come from multiple small N_H architectures cooperating. Deep pipelines like `Canny` would benefit more from reducing the number of passes over increasing N_H. Trading one $N_H = 32$ architecture for 4 N_H architectures could double the performance of Canny and other multi-stage functions. This would bring the performance of our architecture closer to single-pass custom hardware pipelines. Shared-memory architectures would be necessary to minimize main-memory communication and reduce contention.

6.5 Related Work

6.6 Conclusion

In this chapter we proposed a general-purpose hardware architecture using interconnected RISC-V processors. The RISC-V ISA of each processor was extended with FIFO interfaces, Look-up-Table memories, and min/max instructions. The processors were interconnected with FIFOs in a map-reduce parallel pattern structure[30].

We study our proposed architecture and compared it to an optimized software library, and an optimized hardware library. We performed our experiments on two devices from different generations. These experiments gave us insights on our architecture.

We conclude that our architecture out performs an ARM processor on many functions. However, NEON instructions are improving, and memory contention is an issue. Moving forward we propose building multiplecooperating small trees instead of larger trees.

6.7 Acknolwedgements

The authors acknowledge the National Science Foundation Graduate Research Fellowship Program, the Powell Foundation, and ARCS foundation for their support.

Chapter 6, this being prepared for submission for publication of the material. Richmond, Dustin; Kastner, Ryan. The dissertation author was the primary investigator and author of this

material.

Bibliography

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct 1974.

[3] M. Lavasani, H. Angepat, and D. Chiou, "An fpga-based in-line accelerator for memcached," *IEEE Comput. Archit. Lett.*, vol. 13, no. 2, pp. 57–60, Jul. 2014.

[4] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing soc accelerators for memcached," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 36–47.

[5] K. Eguro and R. Venkatesan, "Fpgas for trusted cloud computing," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 2012, pp. 63–70.

[6] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan, "A secure coprocessor for database applications," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–8.

[7] J. W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. Vissers, "A low-latency library in fpga hardware for high-frequency trading (hft)," in *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on*. IEEE, 2012, pp. 9–16.

[8] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, D. Burger, J. Larus, G. P. Gopal, and S. Pope, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA 2014*. IEEE Press, June 2014, pp. 13–24.

[9] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, 2015.

[10] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.

[11] I. Corporation, *Intel Xeon Processor Scalable Family*, https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-scalable-datasheet-vol-1.pdf, Intel Corporation.

[12] J. Saad, A. Baghdadi, and F. Bodereau, "Fpga-based radar signal processing for automotive driver assistance system," in *Rapid System Prototyping, 2009. RSP'09. IEEE/IFIP International Symposium on*. IEEE, 2009, pp. 196–199.

[13] K. Taylor, "Audi selects altera soc fpga for production vehicles with piloted driving capability."

[14] P. Newsome, "The application of fpgas for wireless base-station connectivity," *Xilinx Whitepaper*.

[15] D. F. Bacon, R. Rabbah, and S. Shukla, "Fpga programming for the masses," *Communications of the ACM*, vol. 56, no. 4, pp. 56–63, 2013.

[16] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovi, "Chisel: Constructing hardware in a scala embedded language," in *DAC 2012*, June 2012, pp. 1212–1221.

[17] D. Lockhart, G. Zibrat, and C. Batten, "Pymtl: A unified framework for vertically integrated computer architecture research," in *MICRO 2014*, Dec 2014, pp. 280–292.

[18] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 2017, pp. 1–7.

[19] M. Jacobsen, Y. Freund, and R. Kastner, "Riffa: A reusable integration framework for fpga accelerators," in *Field-Programmable Custom Computing Machines (FCCM), 2012*. IEEE, 2012, pp. 216–219.

[20] M. Jacobsen and R. Kastner, "Riffa 2.0: A reusable integration framework for fpga accelerators," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–8.

[21] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "Riffa 2.1: A reusable integration framework for fpga accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 4, pp. 22:1–22:23, Sep. 2015.

[22] E. S. Chung, J. C. Hoe, and K. Mai, "Coram: an in-fabric memory architecture for fpga-based computing," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 97–106.

[23] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "Platform-based behavior-level and system-level synthesis," in *SOC Conference, 2006 IEEE International*, Sept 2006, pp. 199–202.

[24] A. Papakonstantinou, K. Gururaj, J. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "Fcuda: Enabling efficient compilation of cuda kernels onto fpgas," in *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, July 2009, pp. 35–42.

[25] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 33–36.

[26] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From opencl to high-performance hardware on fpgas," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 2012, pp. 531–534.

[27] S. Neuendorffer, T. Li, and D. Wang, "Accelerating opencv applications with zynq-7000 all programmable soc using vivado hls video libraries," *Xilinx Inc., August*, 2013.

[28] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *2013 23rd International Conference on Field Programmable Logic and Applications*, Sept 2013, pp. 1–4.

[29] V. Kathail, J. Hwang, W. Sun, Y. Chobe, T. Shui, and J. Carrillo, "Sdsoc: A higher-level programming environment for zynq soc and ultrascale+ mpsoc," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 4–4.

[30] K. Asanovic and D. A. Patterson, "Instruction sets should be free: The case for risc-v," EECS Department, Univ. of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, Aug 2014.

[31] K. Eguro, "Sirc: An extensible reconfigurable computing communication api," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2010, pp. 135–138.

[32] J. M. Scott III, "Open component portability infrastructure (opencpi)," MERCURY FEDERAL SYSTEMS INC ARLINGTON VA, Tech. Rep., 2009.

[33] G. Marcus, W. Gao, A. Kugel, and R. Mnner, "The mprace framework: An open source stack for communication with custom fpga-based accelerators," in *2011 VII Southern Conference on Programmable Logic (SPL)*, April 2011, pp. 155–160.

[34] K. Vipin and S. A. Fahmy, "Dyract: A partial reconfiguration enabled accelerator and test platform," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2014, pp. 1–7.

[35] L. Ceze, M. D. Hill, and T. F. Wenisch, "Arch2030: A vision of computer architecture research over the next 15 years," *CoRR*, vol. abs/1612.03182, 2016.

[36] D. P. Singh, T. S. Czajkowski, and A. Ling, "Harnessing the power of fpgas using altera's opencl compiler," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2013, pp. 5–6.

[37] R. Marlow, C. Dobson, and P. Athanas, "An enhanced and embedded gnu radio flow," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. IEEE, 2014, pp. 1–4.

[38] R. Konomura and K. Hori, "Phenox: Zynq 7000 based quadcopter robot," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*. IEEE, 2014, pp. 1–6.

[39] Q. Gautier, A. Shearer, J. Matai, D. Richmond, P. Meng, and R. Kastner, "Real-time 3d reconstruction for fpgas: A case study for evaluating the performance, area, and programmability trade-offs of the altera opencl sdk," in *Field-Programmable Technology (FPT), 2014 International Conference on*. IEEE, 2014, pp. 326–329.

[40] B. Hutchings and M. Wirthlin, "Rapid implementation of a partially reconfigurable video system with pynq," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2017, pp. 1–8.

[41] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 37–47.

[42] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[43] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Are coarse-grained overlays ready for general purpose application acceleration on fpgas?" in *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, Aug 2016, pp. 586–593.

[44] G. Stitt, "Are field-programmable gate arrays ready for the mainstream?" *IEEE Micro*, vol. 31, no. 6, pp. 58–63, Nov 2011.

[45] W. Peck, E. K. Anderson, J. Agron, J. Stevens, F. Baijot, and D. L. Andrews, "Hthreads: A computational model for reconfigurable devices," in *FPL*. IEEE, 2006, pp. 1–4.

[46] J. H. Kelm and S. S. Lumetta, "Hybridos: runtime support for reconfigurable accelerators," in *FPGA*. New York, NY, USA: ACM, 2008, pp. 212–221.

[47] R. Brodersen, A. Tkachenko, and H. K. H. So, "A unified hardware/software runtime environment for fpga-based reconfigurable computers using borph," in *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, Oct 2006, pp. 259–264.

[48] A. Goldhammer and J. Ayer Jr, "Understanding performance of pci express systems," *White Paper: Xilinx Virtex-4 and Virtex-5 FPGAs*, 2008.

[49] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[50] R. Mueller and J. Teubner, "Fpga: What's in it for a database?" in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 999–1004.

[51] L. Josipovic, N. George, and P. Ienne, "Enriching c-based high-level synthesis with parallel pattern templates," in *2016 International Conference on Field-Programmable Technology (FPT)*, Dec 2016, pp. 177–180.

[52] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," *SIGPLAN Not.*, vol. 51, no. 4, pp. 651–665, Mar. 2016.

[53] D. Lee, A. Althoff, D. Richmond, and R. Kastner, "A streaming clustering approach using a heterogeneous system for big data analysis," in *Computer-Aided Design (ICCAD), 2017 IEEE/ACM International Conference on*. IEEE, 2017, pp. 699–706.

[54] J. Matai, D. Richmond, D. Lee, Z. Blair, Q. Wu, A. Abazari, and R. Kastner, "Resolve: Generation of high-performance sorting architectures from high-level synthesis," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: ACM, 2016, pp. 195–204.

[55] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, "Hardware acceleration of short read mapping," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, April 2012, pp. 161–168.

[56] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in haskell," in *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '98. New York, NY, USA: ACM, 1998, pp. 174–184.

[57] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, ser. C++ in-depth series. Addison-Wesley, 2005.

[58] C. Baaij, "Cλash : from haskell to hardware," December 2009.

[59] R. A. Fisher, *The design of experiments*. Oliver And Boyd; Edinburgh; London, 1937.

[60] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[61] R. Nikhil, "Bluespec system verilog: efficient, correct rtl from high level specifications," in *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, June 2004, pp. 69–70.

[62] D. Navarro, O. Lucia, L. A. Barragn, I. Urriza, and O. Jimenez, "High-level synthesis for accelerating the fpga implementation of computationally demanding control algorithms for power converters," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1371–1379, Aug 2013.

[63] F. Vahid, T. Givargis *et al.*, "Timing is everything–embedded systems demand early teaching of structured time-oriented programming," *Proceedings of WESE*, pp. 1–9, 2008.

[64] D. Richmond, J. Blackstone, M. Hogains, K. Thai, and R. Kastner, "Tinker: Generating custom memory architectures for altera's opencl compiler," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 21–24.

[65] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[66] P. Ogden, "Ipython magic for compiling for microblaze," https://github.com/PeterOgden/ipython_microblaze, 2018.

[67] VectorBlox, "Orca," https://github.com/VectorBlox/orca, 2018.

[68] C. Wolf, "Picorv32 - a size-optimized risc-v cpu," https://github.com/cliffordwolf/picorv32, 2018.

[69] RoaLogic, "Rv12," https://github.com/RoaLogic/RV12, 2018.

[70] SpinalHDL, "Vexrisc-v: A fpga friendly 32 bit risc-v cpu implementation," https://github.com/SpinalHDL/VexRiscv, 2018.

[71] B. Review, "Pynq risc-v overlays: A library of rv32im overlas for pynq," https://github.com/Qi7aQtNixPNQVGU/PYNQ-RISC-V-Overlays, 2018.

[72] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, Univ. of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016.

[73] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark," *Commun. ACM*, vol. 27, no. 10, pp. 1013–1030, Oct. 1984.

[74] E. Matthews and L. Shannon, "Taiga: A new risc-v soft-processor framework enabling high performance cpu architectural features," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2017, pp. 1–4.

[75] D. Kehagias, "A survey of assignments in undergraduate computer architecture courses," *International Journal of Emerging Technologies in Learning (iJET)*, vol. 11, no. 06, pp. 68–72, 2016.

[76] B. Landers, "Risc-v simulator for education," summerofcode.withgoogle.com/archive/2017/projects/6309390843904000/, 2017.

[77] D. Page, *SPIM: A MIPS32 Simulator*. London: Springer London, 2009, pp. 561–628.

[78] "The arm mbed iot device platform," www.mbed.com/en/, Jan 2018.

[79] A. van Deursen, A. Mesbah, B. Cornelissen, A. Zaidman, M. Pinzger, and A. Guzzi, "Adinda: a knowledgeable, browser-based ide," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, May 2010, pp. 203–206.

[80] I. Branovic, R. Giorgi, and E. Martinelli, "Webmips: A new web-based mips simulation environment for computer architecture education," in *Proceedings of the 2004 Workshop on Computer Architecture Education*, ser. WCAE '04. New York, NY, USA: ACM, 2004.

[81] J. Tandon, "The openrisc processor: Open hardware and linux," *Linux J.*, vol. 2011, no. 212, Dec. 2011.

[82] OpenRISC, "mor1kx - an openrisc 1000 processor ip core," https://github.com/openrisc/mor1kx, Jan 2018.

[83] E. Matthews, "Taiga - a 32-bit risc-v processor," gitlab.com/sfu-rcl/Taiga, 2018.

[84] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanovic, "Boom v2: an open-source out-of-order risc-v core," EECS Department, Univ. of California, Berkeley, Tech. Rep. UCB/EECS-2017-157, Sep 2017.

[85] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "Flexpret: A processor platform for mixed-criticality systems," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014, pp. 101–110.

[86] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, "Shakti-t: A risc-v processor with light weight security extensions," in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, ser. HASP '17. New York, NY, USA: ACM, 2017, pp. 2:1–2:8.

[87] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: A platform for high-level parametric hardware specification and its modular verification," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 24:1–24:30, Aug. 2017.

[88] J. Gray, "Grvi phalanx: A massively parallel risc-v fpga accelerator accelerator," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2016, pp. 17–20.

[89] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Gurkaynak, and L. Benini, "Pulpino: A small single-core risc-v soc," in *2016 RISC-V Workshop*, 2016.

[90] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 24–35.

[91] J. Balfour, W. Dally, D. Black-Schaffer, V. Parikh, and J. Park, "An energy-efficient processor architecture for embedded systems," *IEEE Computer Architecture Letters*, vol. 7, no. 1, pp. 29–32, Jan 2008.

[92] Y. H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 367–379.

[93] D. Bacon, R. Rabbah, and S. Shukla, "Fpga programming for the masses," *Queue*, vol. 11, no. 2, pp. 40:40–40:52, Feb. 2013.

[94] P. Meng, A. Althoff, Q. Gautier, and R. Kastner, "Adaptive threshold non-pareto elimination: Re-thinking machine learning for system level design space exploration on fpgas," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016, pp. 918–923.

[95] K. Denolf, "Pynq - computer vision overlay," https://github.com/Xilinx/PYNQ-ComputerVision, July 2018.

[96] None, "Xilinx opencv user guide," *Xilinx Inc.*, Jan 2018.

[97] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 205–218.

[98] *Cortex A9 Technical Reference Manual*, ARM, http://infocenter.arm.com/help/topic/com.arm.doc.subset.cortexa.a9/index.html, 2012.

[99] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf, "A massively parallel coprocessor for convolutional neural networks," in *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, July 2009, pp. 53–60.

[100] V. Sriram, D. Cox, K. H. Tsoi, and W. Luk, "Towards an embedded biologically-inspired machine vision processor," in *2010 International Conference on Field-Programmable Technology*, Dec 2010, pp. 273–278.

[101] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 247–257. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815993

[102] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, Oct 2013, pp. 13–19.

[103] Y. Mathys and A. Chatelain, "Verification strategy for integration 3g baseband soc," in *Design Automation Conference*, June 2003, pp. 7–10.

[104] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," TECHNICAL REPORT, UC BERKELEY, Tech. Rep., 2006.

[105] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, Nov 1986.

[106] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Proceedings of the Sixth International Conference on Computer Vision*, ser. ICCV '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 839–. [Online]. Available: http://dl.acm.org/citation.cfm?id=938978.939190