High Speed Software Radio on General Purpose CPUs

by

Christopher William Yarp

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Spring 2022

High Speed Software Radio on General Purpose CPUs

Abstract

High Speed Software Radio on General Purpose CPUs

by

Christopher William Yarp

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Science

University of California, Berkeley

Professor John Wawrzynek, Chair

Real-time, high performance, radio signal processing has traditionally been implemented on custom ASICs or FPGAs. While powerful and efficient, these hardware platforms require extensive engineering effort to create and are relatively inflexible post deployment. Software Radio (SR) presents a compelling alternative to ASIC and FPGA solutions by utilizing the consistently expanding compute power available on CPUs. However, extracting the parallelism available in a radio design and effectively mapping it to the different modes of parallelism available on the CPU (SIMD units, superscalar out-of-order cores, MIMD across cores) to obtain the desired performance is challenging, requiring in-depth knowledge of both the signal processing design and CPU microarchitecture.

This project aims to demonstrate the feasibility of high-performance software radio by identifying causes behind gaps in expected performance vs. achieved performance, developing solutions to address platform and design limitations, and improving designer productivity by providing a flow from signal processing dataflow graphs to optimized multithreaded C applications. To achieve these goals, a variety of tools were developed including a custom written dataflow-graph-to-C compiler (Laminar) which contains optimization passes specifically targeting software implementations of streaming DSP. Performance modeling, benchmarking, and telemetry collection are used to assist DSP co-design and provide insight for future CPU designs.

To my family

Your steadfast love and support helped me reach this point. Thank you for standing by me through the highs and lows.

Contents

List of Figures

List of Tables

List of Code Listings

Acknowledgments

At the risk of inducing eye rolls by using this clichéd adage, the work that I have been able to achieve while a graduate student at the University of California, Berkeley has truly been the result of "standing on the shoulders of giants". I mean this in terms of the wealth of work which has been built upon by generations of students and practitioners, knowledge that has been passed down by professors and colleagues, and the support and friends and family who have lifted me up when I needed it most. So many people have been involved either directly or indirectly in this project that it is likely that I will miss some of them here. To those people, I would like to extend my most heartfelt thanks for your help.

This project grew out of several different efforts at the Berkeley Wireless Research Center (BWRC) under the watchful eye of Prof. John Wawrzynek, Prof. Robert Brodersen, and Dr. Gregory Wright (of Nokia Bell Labs). John, who has been my advisor since I set foot on the Berkeley campus, has provided me with constant support over the years. His critical eye, architecture expertise, helpful suggestions, and flexibility to let me explore the different aspects of this topic have helped lead me and this project to its exciting conclusion. Prof. Brodersen and Dr. Wright, while not my formal advisors, have played such critical roles in this project, along with Prof. Wawzeynek, that I consider them my informal advisors. Prof. Brodersen's knowledge and passion for radio, willingness to debate the merits of different implementations, and dig deep into challenges has helped shape this project into what is has become. Dr. Wright's expertise in modern radio systems, perspective on what matters most, and willingness to answer esoteric radio questions have been invaluable throughout this project. His advice has helped focus this work and pulled me out of holes I had dug myself too deeply into. His willingness to review and provide helpful suggestions and edits to this manuscript as well as presentations delivered throughout my graduate school journey have also been godsends. Prof. Robert Leachman of Industrial Engineering and Operations Research (IEOR) has been kind enough to serve on both my Qualifying Exam Committee and this Dissertation Committee. I cannot thank him enough for his feedback and willingness to wade through this specialty work on software radio.

The motivation for this project came out of my experience after participating in the Berkeley effort for the DARPA Spectrum Collaboration Challenge (SC2). Through this project, I had the opportunity to interact and learn from faculty and students alike. I would like to thank Profs. Anant Sahai, Prof. Jean Walrand, and Prof. Adam Wolisz for sharing their wisdom and providing feedback on early versions of this project. I would also like to extend a special thank you to Prof. Sahai for serving on my Qualifying Exam Committee and providing valuable feedback on the potential merits and directions for this project. The SC2 effort extended over two years during which time I had the privilege of collaborating with fellow students and visiting researchers including James Dunn, Josh Sanz, Laura Brink, Colin de Vrieze, Arya Reais-Parsi, Vasuki Narasimha Swamy, Vidya Muthukumar, Eleanor Cawthon, Vignesh Subramanian, Nikunj Jain, Anish Prabhu, Jason Hongzhen Shan, Gregory Wright, Douglas Chan, and Meryem Simsek.

Industry collaboration has been a cornerstone of the BWRC research activities since its inception, bringing perspective on the biggest problems facing the world outside academia and providing resources which can make all the difference. I would like to thank Tom Burd, Anil (Gohan) Harwani, and Keesang Song of AMD for their help understanding the best ways to interact with modern AMD CPUs, their invaluable feedback and interest, and their help obtaining platforms which were used for testing in the project. I would also like to thank Farhana Sheikh, Kamran Azadet, and Tushar Patel of Intel for providing excellent feedback during the early stages of the project and later as results started coming in. Kees Vissers of Xilinx and Alan Gatherer of Futurewei both provided excellent perspective on this work and ideas on where to take it, for which I am very grateful.

I could not be more grateful that I was able to call the BWRC home during my time at UC Berkeley. It has given me the opportunity to interact with the best and brightest in the field. I would like to thank all the students and staff at the BWRC for making my time at Berkeley one which I will treasure. I would like to extend a special thank you to my former lab mates Paul Rigge, John Wright, and Angie Wang who I had the privilege of working with on a few different projects and who have provided me with counsel and perspective more times than I can count. I would also like to thank the members of John Wawrzynek's research group, past and present, including Arya Reais-Parsi, Tan Nguyen, Qijing Huang, James Martin, Simon Scott, and Shaoyi Cheng for the comradery, advice, feedback, and fun. The staff, past and present, of the BWRC including Candy Corpus-Steudeman, Mikaela Cavizo-Briggs, James Dunn, Brian Richards, Fred Burghardt, Ajith Amerasekera, Jeff Anderson-Lee, Greg Pearson, Amber Sanchez, Olivia Nolan, Sarah Jordan help make BWRC the welcoming and supportive environment is. I am truly grateful for all their help. I would also like to thank the faculty members of the BWRC including Profs. John Wawrzynek, Robert Brodersen, Bora Nikolic, Jan Rabai, Ali Niknejad, Vladimir Stojanovic, Ana Arias, Elad Alon, Rikky Muller, and Sophia Shao for fostering an atmosphere of collaborative discovery.

The ethos of collaboration in Berkeley EECS runs deep with students lending each other helping hands as we collectively wade through the sometimes-harrowing journey which is graduate school. To all the students who I have had the privilege to work alongside, thank you for your contributions to this community. I would like to especially thank Josh Sanz, Nathan Pemberton, and Marquita Ellis for their help understanding aspects of my project that strayed outside of the traditional bounds of my sub-specialty and into their own.

I would also like to thank Profs. Dorit Hochbaum and Robert Leachman of IEOR and Prof. Satish Rao of EECS for their willingness to talk about potential optimizations solutions to the challenging scheduling problems early in this project.

Graduate school can be a complex and daunting journey, with many unexpected twists and turns. I would like to express my deep gratitude to Shirley Salanio who has helped me navigate the streets of Berkeley (both literally and figuratively) since my first visit to the campus as a perspective student. Her constant support and willingness to answer the many questions I had throughout the process has helped me stay on course during my time here.

As I reach the end my graduate school career, I have had the opportunity to reflect back on all of the help I have received along the way to even have the opportunity to reach this point.

I would like to thank the faculty of my alma mater, Santa Clara University, and every teacher I have had through the course of my schooling who have devoted their time to educating me and my fellow classmates. Without the knowledge I gained through their efforts, I would not be in the position I am today. I would like to especially thank my undergraduate advisors Profs. Sarah Kate Wilson, Silvia Figueira, and Sally Wood for fostering my interest in Electrical Engineering and Computer Engineering while an undergraduate and encouraging me to pursue graduate studies. My undergraduate experience vastly broadened my horizons and much of the credit for that rests with the efforts of the SCU Engineering faculty and staff.

Graduate school, with all its excitement, can also be tumultuous at times. My family and friends have provided me with the support to get back up when things were looking dark and a patient ear when stress consumed me. I cannot thank my parents, Russ and Mara, and my sister Jennifer enough for their steadfast love and support during this process. I would also like to reflect on the memory of my grandparents who gave so much to help and support me. I wish they were still here to see this journey come to fruition. To my close-knit group of friends including Aaron, Carlos, Mark, and Zach who have kept me going throughout this process, thank you. To the EECS AoE community for providing a welcome place to play some games and chat about the realities of graduate school, thank you. Your help and perspective are most appreciated, especially when the COVID pandemic isolated us from our colleagues and lab mates. I would also like to thank Andrew Mellows who served as my mentor during the formative years of my life. His wisdom and encouragement helped spur my interest in signal processing and for that, I am eternally grateful.

This work would not be possible without the direct and indirect support of organizations. I would like to acknowledge the support of a grant from Nokia and equipment donations from Advanced Micro Devices (AMD) for this project. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Precursor work was supported in part by an SC2 project grant administered by the Air Force Research Laboratory (AFRL) and DARPA as well as by the TerraSwarm Research Center, one of six centers administered by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation (SRC) program sponsored by MARCO and DARPA. This project was also supported in part by the funds and resources of the Berkeley Wireless Research Center (BWRC). Finally, I would like to thank the educational programs from GitHub, Travis CI, and JetBrains for the use of their tools and services during my graduate school career.

Chapter 1

Introduction

1.1 Motivation

Radio systems have become indispensable in modern society. From Wi-Fi to cellular data, radios are ubiquitous and dependence on them is only growing [1]. Underpinning all modern radio systems is a suite of signal processing tasks. Due to the performance required, much of the signal processing workload has traditionally been conducted on specialized hardware such as custom Application Specific Integrated Circuits (ASICs), purpose-built System-on-Chips (SoCs), and Field Programmable Gate Arrays (FPGAs). While these platforms are typically able to provide the required level of performance and efficiency, they have several drawbacks:

- Designers typically require hardware engineering experience, familiarity with the underlying hardware technology, and knowledge of Electronic Design Automation (EDA) tools to achieve quality results.
- Iterating on designs is slow due to the speed of modern EDA tools and the long fabrication times for ASICs and custom SoCs.
- High (ASCIs and custom SoCs) to moderate (FPGAs) fixed development and production costs which can only be amortized with large volume production.
- Low (ASICs) to moderate (SoCs and FPGAs) flexibility after deployment.

Due to their large volume[1] and limited battery capacity, wireless handsets and laptops can justify their use of custom chips. By contrast, wireless infrastructure (including baseband processing for cellular base stations) does not enjoy such large volume.[2] As a consequence,

[1]In 2020 Apple sold just short of 200 million iPhones globally with a global market share of 14.8% [2]. The total number of smart phones, across all vendors, sold during the same period in North America alone was 136 million [3].

[2]In 2020, there were 417,000 cell sites in the US [4]. Note that cell sites are typically not upgraded with new hardware each year.

producing custom ASICs and SoCs for cellular base stations is considerably less appealing. While more affordable than short-run ASICs and SoCs, FPGAs still require a notable Non-Recurring Engineering (NRE) effort to produce the design which is ultimately loaded onto them.

Software-based solutions leveraging general-purpose processors are a promising alternative, particularly for the lower volume wireless infrastructure industry. The potential advantages of software radio include the following.

- General purpose CPUs often have a low per-unit cost due to their wide use in multiple industries.
- Many general purpose high-performance CPUs are generally available and incorporate continued improvements over time (ex. increased core counts[3] and wider SIMD / vector units[4]) thanks in part to demands by a wide range of customers and competition from other vendors.
- Software based solutions are flexible after deployment unlike ASIC-based solutions which must include any flexibility at design time.
- Software development can leverage a mature ecosystem that includes fast compilers and software engineering tools/practices to reduce time to market.

However, creating a high-performance software radio has proven challenging. In the author's experience, implementing simple algorithms using one of the existing Software Defined Radio (SDR) frameworks is often promising at low bandwidths but quickly becomes limited when design complexity or bandwidth requirements increase. Despite modern CPUs operating at clock rates in the GHz range, matching the speeds of hardware implementations requires the CPU to process a sample in under 100 cycles, on average. To put this concretely, if a baseband specified for 3 samples/symbol is targeted a channel bandwidth of 20 MHz, it would need to process complex samples[5] at a rate of 60 Msps. On a CPU running at an aggressive 3.7 GHz, this would require a sample to finish processing every 61.7 cycles on average. Already difficult for a design of significant complexity, this constraint becomes more challenging as wider bandwidths are desired. Given this implementation challenge, it is a reasonable question to ask if general-purpose CPUs can possibly support the specifications required to make a software radio competitive with a hardware implementation.

As a litmus test for the potential of software radio on modern general-purpose CPUs, a simplified model was constructed using an early version of the Cyclops baseband, an FPGA radio design which has been used for several projects at the Berkeley Wireless Research Center (BWRC). The operators in the hardware design, excluding FSMs, were counted as

[3]For example, AMD increased the maximum core count of their Epyc server line from 32 in the Epyc 7001 series [5] to 64 in the Epyc 7002 series [6].

[4]Intel has increased the vector width from 256-bits (AVX/AVX2) to 512-bits (AVX512) in some of their parts [7].

[5]Complex samples including real and imaginary components ($a + bi$).

shown in Table 1.1. Some of these operations involve complex numbers and were converted to give a rough estimate of the number of primitive hardware operations in the design, as shown in Table 1.2.

Required Operations	Real	Complex	Mixed
Mult	57	76	6
Gain	313	18	0
Sum	327	111	0
Lookup Table Address Compute	11	0	0

Table 1.1: Required Hardware Operations (Excluding FSMs) in Early Cyclops Radio

Required Primitive Ops	
Mult	769
Sum	748
Total	1517

Table 1.2: Required Hardware Primitive Operations (Excluding FSMs) in Early Cyclops Radio

At the time, the newly released AMD Ryzen Threadripper 2990WX represented an interesting target CPU featuring a base clock of 3.0 GHz, and 32 high-performance cores [8]. Each of these cores was superscalar, out-of-order, with 128-bit SIMD/vector units [9]. To get an estimated performance upper bound of Cyclops on this CPU, the operators (which were expected to be on single precision floating point values) were assumed to be independent. The hypothetical upper bound execution rates of Cyclops using this simplified execution model on the Threadripper 2990WX under different assumptions on the parallel resources used on the CPU is shown in Table 1.3. Note that some operations run every cycle and some less often (ideally every fourth sample). On the FPGA targets for Cyclops baseband, the worst-case specification for these operators was that they would run every third sample. The execution rates if these operators executed every sample and if they executed every third cycle are both shown in Table 1.3.

These upper bounds are quite promising considering that a less complex version of the Cyclops baseband used for the author's master's thesis [10] operated at a 250 MHz clock rate on a Xilinx Zynq 7000 series FPGA and a more complex version of the baseband developed for the Berkeley effort in the DARPA Spectrum Collaboration Challenge (SC2) operated at 100 MHz on a Xilinx Kintex 7 FPGA. With the baseband spread across all 32 cores on the system and making parallel use of all the floating-point units in scalar mode, the performance upper bound exceeds that of the SC2 FPGA implementation by between two

Scenario	Upper Bound Max Sample Rate (Ms/s)	
	Slow Rx Executes for Each Sample	Slow Rx Executes Every 1/3 Samples (Worst Case)
Serial Execution (No Vectorization, No Parallel use of FP Units)	1.98	2.92
1 Core, Parallel Use of FP Units (Scalar Mode)	7.79	11.5
1 Core, Parallel Use of FP Units (Vector Mode)	30.9	46.2
32 Cores, Serial in Core, Fully Parallel Across Cores	62.5	90.9
32 Cores, Parallel Use of FP Units in all Cores (Scalar Mode)	231	333
32 Cores, Parallel Use of FP Units in all Cores (Vector Mode)	750	1000

Table 1.3: Estimated Upper Bounds for Cyclops Executing on Ryzen Threadripper 2990WX at 3.0 GHz Base Clock Rate

and three times. This illustrates the potential of software radio, but also makes it clear that attaining competitive data rates requires aggressive utilization of the different parallel resources available on the CPU.

1.2 Objectives and Contributions

The primary objective of this project is to evaluate the feasibility of software radio on modern general-purpose CPUs by:

- Demonstrating a fast and efficient software radio system on a generally available CPU.
- Identifying limitations of radio designs and platforms that can impede performance.
- Evaluating and recommending software radio baseband co-design considerations and optimization opportunities.
- Providing automation for optimization passes, facilitating their consistent application across designs and design iterations.

To accomplish these goals, the project is broken into three main components.

- Cyclops: a complete radio baseband DSP design used as the primary demonstrator for the project. It is under the author's full control and allows exploration of software-algorithm co-design considerations.

- Laminar: an optimizing DSP Compiler for dataflow graphs. It provides a framework for experimenting with different optimization techniques. It also automates optimization passes and multicore code generation so that they can be applied consistently across design revisions.
- Platform Characterization: A series of benchmarks which provide insight into the capabilities of the target platform. The information obtained from these benchmarks is used to inform expected performance and guide design tuning.

These components form the basis of a design cycle which involves passing the Cyclops design to Laminar which creates a C implementation that is then compiled with a general-purpose C compiler. Laminar-inserted telemetry collection code along with analysis scripts are used to measure the performance of the executable. Measured performance and platform characterization information are used to inform algorithmic changes in Cyclops and/or change the parameters passed to Laminar. With these changes, the design cycle repeats again.

Through multiple iterations of the design cycle, improvements to the Laminar compiler, and changes to the Cyclops design to better match the CPU target, a high-performance software realization of Cyclops capable of running at 104.5 Msps (with a maximum payload rate of 278.7 Mbps when operating in 256 QAM mode) was achieved on an AMD Ryzen Threadripper 3970X running at its base clock rate of 3.7 GHz. This dissertation details how these results were achieved through logic implemented in the Laminar compiler, algorithm co-design changes to the Cyclops baseband, and the use of platform characterization to partition and map the design to CPU cores.

1.3 Prior and Related Work

There was a concentrated period of work exploring the mapping of signal processing tasks to multiprocessor systems in the mid/late 1980s into the early/mid 1990s. Much of this work was conducted at the University of California, Berkeley with representative works including [11]–[15]. Some ideas developed in these works, including the use of synchronous dataflow graphs [11], [12] and the self-timed strategy under the scheduling taxonomy noted by Edward Lee and Soonhoi Ha in [13] are used in this project. However, the characteristics of the target CPUs have changed since this period of research, with modern general-purpose CPUs exhibiting the following:

- Less control over scheduling within the core with superscalar, out-of-order execution, and potential speculative execution.
- A loss of control and visibility of the interconnect. Inter-core communication and communication to memory is now handled by the cache coherency system.

- Not all cores in the system are guaranteed to see a change to a memory location at the same time (governed by the memory consistency model).
- A proliferation of many high-performance SIMD-enabled processing cores on a single socket.
- Introduction of more complex caching systems and interconnect topologies.
- Increase memory access latency relative to compute latency[6]
- Evolution of C/C++ compilers with advanced optimization passes.

Some of these changes complicate the scheduling approaches used in previous works, as it is now much more expensive to communicate with another core and access memory relative to computing on a local core[7]. The cost of communication on modern systems makes it largely infeasible to transact in single samples. It was also unclear at the onset of this project how capable modern C/C++ compilers are at producing executables with instructions ordered to perform best with the out-of-order scheduler on modern CPUs. This work aims to reevaluate the feasibility of implementing radio DSP on multicore processors with modern software tools.

Related concepts to Synchronous Dataflow include the more general [18] Kahn Process Network (KPN) described in Gilles Kahn's seminal paper [19] and Communicating Sequential Process (CSP) model described by C. A. R. Hoare [20]. Synchronous Dataflow can be viewed as a restriction of these models with the added requirement that the number of tokens consumed and produced by each process in the system is known a priori. Due to the close relationship between these concepts, advances in describing and analyzing each of these models can potentially be used by this project. For example, the *Go* language bases its concurrency model on CSP [21] and could potentially be used as an input language for describing DSP.

There exist several different frameworks for implementing radio and packet processing in software. On the radio side, probably the most well-known is GNURadio [22] which supports several different radio frontends [23] including the popular Ettus Research USRP [24]. GNURadio allows signal processing blocks to be written in multiple different languages, typically C/C++ or Python with GNURadio handling the orchestration between blocks. In conjunction with the RFNoC [25], GNURadio was used for the Berkeley effort

[6]For example, the Intel Pentium processor, released in 1993, has a compute operation latency of 76 ns and a memory latency of 75 ns. The Intel Core i7, released in 2010, has a computation latency of 4 ns and a memory latency of 37 ns [16].

[7]This effect is not isolated to DRAM and is present even when accessing caches. For example, a modern AMD CPU has a L1 load-to-use latency of between 4 and 8 cycles, L2 load-to-use latency of at least 12 cycles, and an average L3 load-to-use latency of 39 cycles [17]. By comparison, the latency of an integer multiply 3-4 cycles and the latency of a floating point multiply is two cycles on the same CPU (assuming the operands are in registers) [17]. On these modern x86_64 CPUs, the only feasible method for cores to communicate is via the cache coherency system. In the best-case scenario, the communicating cores are in the same L3 cache domain. However, if they are apart, the communication requires transactions over the interconnect between L3 domains, increasing latency.

in the DARPA Spectrum Collaboration Challenge (SC2), in which the author was an active participant. While it is clear that frameworks like GNURadio provide an exceptional tool for education and experimentation with radio, extracting the desired level of performance proved challenging in the SC2 project with GNURadio acting as the performance bottleneck rather than the FPGA portion of the design. This project aims to address the challenges experienced with GNURadio by taking more control over the orchestration and optimization of the design while also providing additional insight into what aspects of the algorithms or system architecture are hindering performance.

DPDK [26] is another framework focused on packet processing with a particular focus on the upper layers of the network stack. While these layers are important and are key to the Network Function Virtualization (NFV) goals of network operators [27], they are generally outside the scope of this project, which is focused on the signal processing work in the baseband. There is some support for baseband processing within DPDK, but it appears to be currently focused on accelerating Forward Error Correction (FEC) with the desire to later add support for additional baseband processing blocks [28]. Because of DPDKs focus on the necessary upper layers of the network stack, it is probably a good companion to the work undertaken in this project to produce a complete radio system.

There has recently been a push to open up the core of cellular Radio Access Networks (RANs) to allow operators to select different components from different vendors. Initiatives such as OpenRAN [29] and O-RAN [30] aim to standardize interfaces between components in a typical RAN, allowing interoperability. There is also emphasis on enabling Network Function Virtualization (NFV) and software-based solutions on standard hardware. There are several efforts, including srsRAN (formerly srsLTE) [31], the OpenAirInterface 5G RAN project [32], Intel FlexRAN [33], and NVidia Aerial [34] to run the PHY layer either fully or partially in software either on CPU or GPU platforms. It is important to note that these efforts are focused on 3GPP 4G and 5G standards and are aimed at deployment of networks using those standards. 3GPP and O-RAN both define multiple split points for what resides in hardware and what resides in software [35]. For example, NVidia Aerial uses a 7.2 split [34] which corresponds to the *Low PHY* and *RF* being implemented in hardware [35]. Specifically, with the 7.2 split the “FFT, CP removal, resource de-mapping and possibly pre-filtering functions reside in the DU” on the uplink side and the “iFFT, CP addition, resource mapping and precoding functions reside in the DU” on the downlink side[8]. With the 7.2 split, a significant amount of signal processing, albeit standardized for this protocol, is being conducted outside the scope of the software. While these efforts have produced impressive results, their primary focus is the implementation of specific cellular baseband standards. This project takes a more general view of investigating the problems faced when mapping an entire radio DSP design to software and producing solutions to those problems that can potentially be applied across multiple different baseband designs.

[8]The *DU* is the *distributed unit* while the rest of the functions reside in the *CU* or the *central unit* [36]. For designs seeking to implement the CU in software, there is typically the expectation that the DU is in hardware.

A set of programs with objectives analogous to this project are hardware simulators, such as Synopsys VCS [37] and the open-source Verilator [38]. These tools are indispensable for digital design and are relied upon to test and simulate designs before manufacturing. Due to the high degree of complexity in some modern hardware designs, there is pressure on these simulators to operate as quickly as possible while providing accurate results. Verilator has recently integrated multithreaded execution to accelerate simulations [39]. While these programs share the same general objective of accelerating the simulation of digital designs, which can often be viewed as dataflow (as will be discussed in section 5.1), they face additional constraints on their execution with a need to provide bit and cycle accurate simulations that cover the complex event semantics and signal types (such as *high-z* and *don't care*) that are present in HDL languages like Verilog. Sometimes these constraints can be relaxed to attain higher simulation speed, such as Verilator converting synthesizable constructs to C/C++ and converting *don't care* values to random values [39]. The project at the core of this dissertation has the benefit of working in the more restricted domain of signal processing DSP with opportunities for optimizations that are not possible for general-purpose digital design simulation. This project is also centered on the software implementation being the ultimate target of the design, rather than the software implementation acting a mirror to what will ultimately be implemented in hardware. To this end, types (such as floating-point) and operators which are well supported on modern CPUs but are potentially less efficient when designing custom hardware[9] are used.

MathWorks Matlab [40] and Simulink [41] are de facto programs of choice within the digital communications industry for the design and evaluation of radio DSP designs. This is in part to their convenient programming model that is familiar to many engineers and their extensive libraries[10] (called Toolboxes in Matlab parlance) of gold standard reference models, algorithm implementations, analysis tools, and relatively rich documentation. There is a suite of add-on products to Matlab and Simulink including a tool to convert Matlab m-code and Simulink designs to C/C++ called Matlab and Simulink Coder, respectively [44], [45]. To the author's knowledge, at the start of this project, Simulink Coder only produced single-threaded implementations of a given design. More recently, Simulink has been adding support for multithreaded targets and SIMD. This initially appeared to be focused on multi-model simulations [46] but expanded with the ability to support dataflow domains using DSP toolbox [47] and is continuing to extend multicore support in recent releases [48]. An attempt to evaluate the new DSP Toolbox dataflow support revealed limitations[11] which prevented the generation a C/C++ implementation of the Cyclops Rx baseband demonstrator used in this project. However, the movement of MathWorks to

[9]For example, many hardware implementations of DSP algorithms use fixed point representations which can carry the exact number of bits requited and can easily implement re-normalization logic with almost free masks and shifts. Modern CPUs, by comparison, are generally byte-centric and require explicit masking and shifting operations to perform the same operation.

[10]Including the frequently used DSP Toolbox[42] and Communications Toolbox [43].

[11]Limitations encountered at the time including the use of switches (ie. muxes), enabled subsystems, and stateflow diagrams (FSMs).

support multicore and additional SIMD use underscores the commercial interest in producing high-performance software implementations of dataflow designs such as DSP.

One method to accelerate applications including, but not limited to, DSP is the use of libraries and generators that produce optimized implementations for commonly used and problematic compute kernels. These libraries and generators can be produced by third-party researchers or by CPU/GPU vendors with expert knowledge of their hardware implementation. Examples of such libraries include Automatically Tuned Linear Algebra Software (ATLAS) [49], FFTW [50], Spiral [51], Intel Math Kernel Library (MKL) / oneMKL [52], AMD Optimizing CPU Libraries (AOCL) [53] (including AMD BLIS [54] and AMD FFTW [55]), and NVidia CUDA-X Accelerated Libraries [56] (including cuBLAS [57] and cuFFT [58]). These specialized libraries can provide substantial benefit, especially for complex and well-researched problems such as General Matrix Matrix Multiplication (GEMM) and Fast Fourier Transform (FFT). As such, these libraries could be used to augment the emitted operations provided by this project's DSP compiler, Laminar. This dissertation more broadly considers the entire radio system, including inter-core communication concerns and baseband co-design, and rather solely focusing on specific computational kernels[12].

The difficulty of the general software optimization problem has led to the creation of Domain-Specific Languages (DSLs), frameworks, and compilers which provide performance benefits within a restricted domain that would be challenging for a general-purpose compiler to achieve on its own. Examples include Halide [59] for image processing and TensorFlow [60], Caffe [61], and PyTorch [62] for machine learning. Halide pays special attention to the structure and locality of computation for its chosen domain of image processing. TensorFlow, Caffe, and PyTorch all provide functions and optimizations that are important to machine learning developers. This project can be viewed in a similar vein as a domain-specific compiler, Laminar, combined with design recommendations and evaluations for radio digital signal processing on general-purpose CPUs.

While this project implements a domain-specific compiler (Laminar) for radio signal processing, it is a source-to-source compiler which emits C code. General-purpose C compilers, such as Clang/LLVM [63], gcc [64], or vendor-supplied compilers such as AMD Optimizing C/C++ Compiler (AOCC) [65] are used to compile the final executable file. The compilers include many optimizations [66], [67] such as auto-vectorization which can provide significant performance gains for DSP algorithms that execute on CPUs equipped with SIMD/vector units. Instead of trying to reimplement these optimizations, the Laminar compiler developed for this project seeks to use them by crafting C code which can easily be analyzed and optimized by existing C compilers. This general strategy is discussed in detail in section 6.1. Additional libraries and frameworks such as the Message Passing Interface (MPI) [68] and the OpenMP parallel framework [69] can be used with compilers but are not used in this project. This project targets single socket shared memory operation and does not require

[12]One strategy of this project is to represent designs in such a way that a standard C/C++ compiler can easily and analyze some operators common to DSP designs and produce vectorized implementations of them. For more complex algorithms such as FFT or GEMM when specialized matrix properties are not present, the use of libraries such as these could potentially be used by the Laminar emitter.

communication between multiple sockets and servers which MPI provides. Because of this, MPI was viewed as more heavyweight than was necessary and is not used[13]. This project also takes an alternate approach to that of OpenMP's fork/join threading model [69] by creating long-running threads that execute streaming dataflow and pass data via FIFOs.

In general, this is a very interesting time to investigate software radio. There is an opportunity to reassess some of the execution models for DSP from the 1980s and 1990s, such as synchronous dataflow and self-timed synchronization, on modern many-core general-purpose CPUs. It is an area of active interest to the communications industry, with multiple efforts underway to allow interoperability between different components that make up modern cellular networks, and the hope that some of these components can be implemented in software. Academic research has also been ramping up in this area with works such as [70], which focuses on Forward Error Correction (FEC) coding but also provides mechanisms for accelerating general software radio, recently brought to the attention of the author. While conducted independently and focusing on different areas, the general consistency of the observations in this project and that of [70] illustrate the promise of continued research into software radio on general-purpose CPUs. Dataflow processing, an important component of this project, has also been experiencing a resurgence in interest with machine learning frameworks such as Caffe and TensorFlow making use of dataflow graphs for the source description of neural networks [71] or when accelerating computations [72], respectively. A domain-specific source-to-source compiler, combined with insight into common motifs in radio design and the advances in modern general-purpose C compilers, has the potential to generate high-performance software radio implementations.

1.4 Dissertation Overview

As a project spanning several electrical engineering and computer science sub-specialties, the early chapters of this dissertation aim to provide a common foundation on which software radio can be evaluated. Chapter 2 introduces the basics of digital radio and baseband signal processing. It also covers information about the Cyclops baseband used throughout this dissertation as an example of a radio DSP workload and as a demonstrator for different optimization techniques. Chapter 3 discusses the architecture of modern CPUs that run the software radio applications generated by this project. Chapters 3 and 4 detail the different modes of parallelism available in the CPU and radio baseband design. Effectively expressing the parallelism in the radio design and mapping them to the parallel resources available on the CPU is essential to achieve high-performance results and constitutes a major focus of this work. Chapters 5 and 6 discuss the Laminar compiler and the optimization passes it performs on the radio design. Chapter 7 focuses on the techniques and optimizations for passing data between cores on the system. This also includes a discussion of how platform characterization results are used to inform the partition-core mapping process. Chapter 8

[13]The execution model of the code produced by Laminar should be compatible with MPI and it should be possible to use MPI for thread control and FIFO transfers if it is later determined to be advantageous.

covers software radio algorithm co-design considerations that, if adopted, can address problems faced when mapping a given radio design to software. Chapter 9 covers experimental results of Laminar generated realizations of the Cyclops baseband running on an AMD Ryzen Threadripper 3970X and how these results could potentially be improved by running multiple instances of the radio, simultaneously. Concluding remarks, including a discussion of remaining challenges, lessons learned, and future work are discussed in chapter 10.

Chapter 2

Radio Baseband Processing

The target application of this research is the implementation of radio baseband processing. While this work can be generalized to other signal processing tasks, exploiting our knowledge of baseband processing allows us to introduce optimizations and specific to this domain. As this research draws from disparate fields including computer architecture, DSP, and communications, it is not expected that the reader will be familiar with all aspects of the project. This section provides a primer on radio baseband processing and outlines the structure of Cyclops [73], the radio design used as the demonstrator throughout this document. For more information on general radio processing, see texts on digital communications systems such as [74]–[76].

2.1 The Basics

The goal of digital communications systems is to send a set of data from one point to another without the use of wires. In general, this is accomplished in the following steps:

1. Modulation: The digital data is converted into an analog signal. Typically, this signal is centered around DC in the frequency domain and is called the *baseband signal.*
2. Mixing: The analog signal goes through a mixer which moves the center frequency of the modulated signal to a higher frequency, called the *carrier* frequency. To put this in a context most people are familiar with, the carrier frequency is what is dialed in on a car radio to select the station to listen to.
3. Amplification and Transmission: The Mixed Signal is amplified and sent out of the transmitter (Tx) antenna into the air.
4. The Channel: The transmitted signal moves through the environment, emanating from the transmitter and eventually reaching the receiver.
5. Reception and Amplification: The signal is then detected by the receiver (Rx) antenna and is then amplified.

6. Mixing and Filtering: The received signal is passed through another mixer and filter which moves the signal from the carrier frequency back to DC (the baseband signal) and suppresses other received signals.
7. Demodulation: The received analog signal is then mapped back to a digital signal.

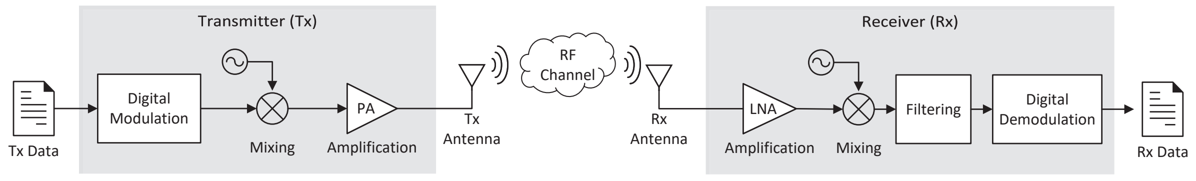


Figure 2.1: Idealized Radio Design

2.1.1 Impairments

In an ideal environment, not much signal processing would be required to achieve the goals outlined above. However, real world conditions significantly complicate matters:

- The signal power density falls off with the square of the distance from the transmitter[1].
  - This demands high gain, linear, low distortion power amplifiers (PAs) on the Tx side and low noise, high gain, linear, amplifiers (LNAs) on the Rx side.
  - Because of signal attenuation in the air, the signal to noise ratio (SNR) at the receiver suffers, making it harder to correctly decode the received data.
- Since the Tx and Rx are separated without wires, their oscillators and clocks are typically not synchronized[2].
  - This results in the Tx and Rx having slightly different views of the carrier frequency, resulting in Carrier Frequency Offset (CFO).
  - This also means that the Rx may not know exactly when a piece of information is being sent.

[1]This effect is visible in the received power expression for a signal propagating in free space, sometimes called the Friis transmission equation, $P_R = \frac{P_T G_T A_R}{4\pi d^2}$. P_R is the received power, P_T is the transmitted power, G_T is the gain of the transmit antenna, A_R is the effective area of the receive antenna, and d is the distance between the transmitter and receiver [77].

[2]Synchronization against GPS as a reference is sometimes done for fixed, outdoor, installations.

- Objects in the environment can send echoes of the transmitted signal to the receiver. These delayed and attenuated versions of the signal cause the environment to appear as a filter[3].
- Motion of the Tx or Rx introduces a Doppler frequency shift which can be viewed as a time varying CFO component.

While these impairments can be addressed in the analog domain, for example, by adjusting the Phase Locked Loops (PLLs) feeding the mixers to address CFO, it is typical to digitize the received signal and perform corrections in the digital domain.

2.1.2 Data Converters and Analog QAM Modulation

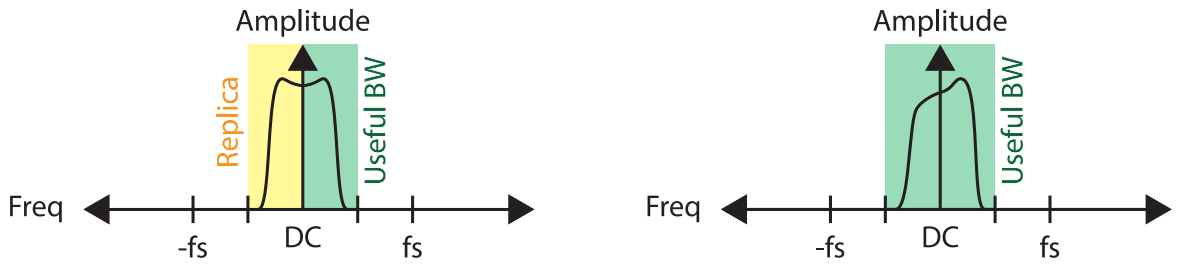
To facilitate the signal processing in the digital domain, data converters are included in the Tx and Rx chains. A Digital-To-Analog Converter (DAC) is included in the Tx chain and an Analog-to-Digital Converter (ADC) is included in the Rx chain[4]. To satisfy the Nyquist sampling theorem[5], the data converters need to operate at a sampling rate at least double the maximum frequency they plan to handle. Real signals are conjugate symmetric across DC in the frequency domain, resulting in half of the sampling rate being available as useful bandwidth. Many modern radios use a trick to utilize lower speed data converters to capture a wider usable bandwidth. Quadrature Amplitude Modulation (QAM) allows a signal to be split into two channels: in-phase (I) and quadrature (Q). On the modulator side, the I channel is mixed up with a cos wave and the Q channel is mixed up with a sin wave (which is $90°$ out of phase with the cos). The results are then summed together. The QAM demodulator does the reverse operation, splitting the received signal and mixing down one instance with a cos and the other with the sin wave. The composite signal can be viewed as a complex phasor at the frequency of the sin/cos with its amplitude and phase derived from the magnitude and angle of the point in the I/Q plane[6]. Because of this relationship, the I and Q signals are typically viewed as the real and imaginary components of a complex signal. Complex signals are not necessarily conjugate symmetric across DC in the frequency domain, resulting in double the useful bandwidth at the cost of an additional ADC/DAC channel. A diagram with a typical data converter and RF frontend topology is shown in Figure 2.3.

[3]Finite Impulse Response (FIR) filters are constructed by summing delayed and attenuated/amplified versions of the signal. For a fixed moment in time, the environment behaves like a frequency selective filter.

[4]The exact composition of the Tx and Rx chain can vary. For example, direct conversion transmitters and receivers omit the analog mixing stage, utilizing high speed ADCs/DACs to handle signals directly at the desired carrier frequency. The mixing operation is performed in the digital domain. Superheterodyne radios mix to an intermediate frequency (IF) before mixing to either baseband or the carrier frequency

[5]See [78] for an explanation of the Nyquist sampling theorem which is also referred to as the Shannon sampling theorem or simply as the *Sampling Theorem*.

[6]For a derivation of the phasor relationship for analog QAM modulation, see the appendices of [10].



(a) Real Signal (b) Complex Signal

Figure 2.2: Real vs. Complex Signal Spectrum

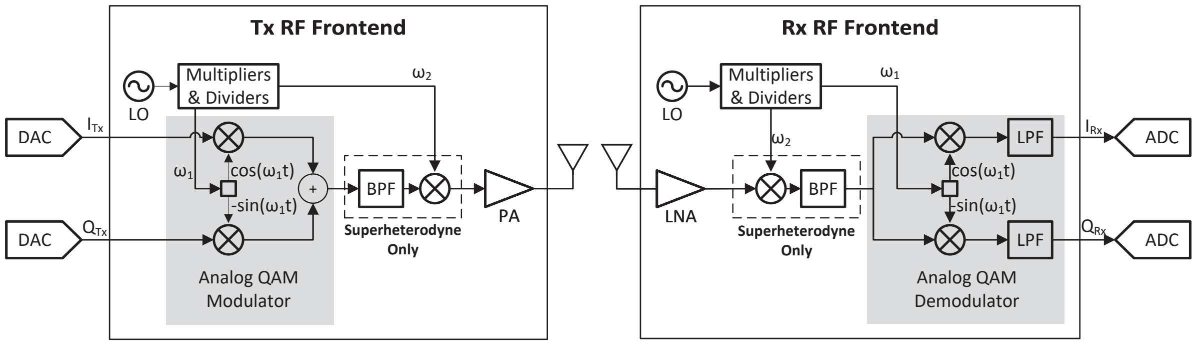


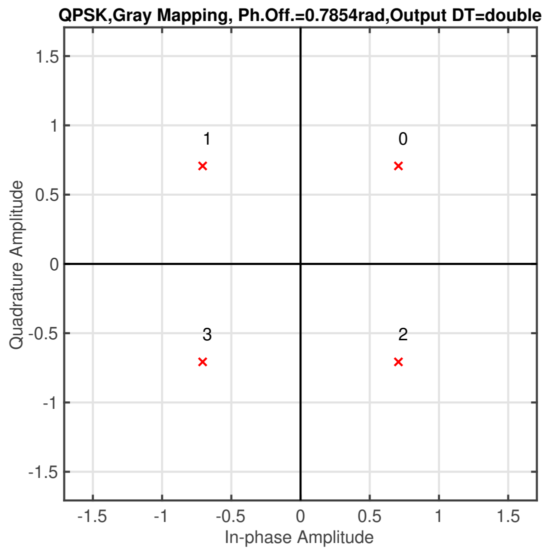
Figure 2.3: Data Converters and Attached RF Frontends (Modified from [10])

2.1.3 Baseband Representation of Digital Signal

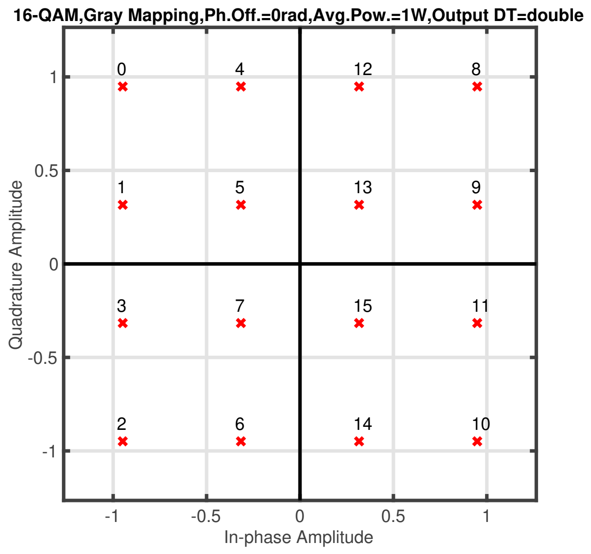
While there are multiple ways to map digital data to analog signals, a very common technique assigns bit patterns to coordinates on the I/Q plane referred to as constellation points. The process of taking bits of data and mapping it to a constellation point is digital modulation and the set of constellation points is referred to as the modulation scheme. The waveform that is created is typically referred to as a symbol. Examples of two common modulation schemes are shown in Figure 2.4.

The transmitted signal changes constellation points at the *symbol rate*. Because sharp changes in the time domain result in wide spectral emissions, the transition between points is typically smoothed through some form of low-pass filter. On the Tx side, it is common for some of this filtering to be performed digitally. In addition to providing some degree of band limiting, the Tx filtering is often used to shape the symbol to exhibit favorable properties such as limited inter-symbol interference. To achieve this, the signal needs to be upsampled to a faster rate before filtering[7]. On the Rx side, it is common for the ADC to run at a faster

[7]Otherwise, being sampled exactly at the Nyquist rate, it would be unable to effect the out-of-band



(a) QPSK



(b) 16QAM

Figure 2.4: Constellation Diagrams for Some Common Digital Modulation Schemes - Generated by Matlab/Simulink [79][80]

sample rate than the symbol rate. This serves multiple purposes, including:

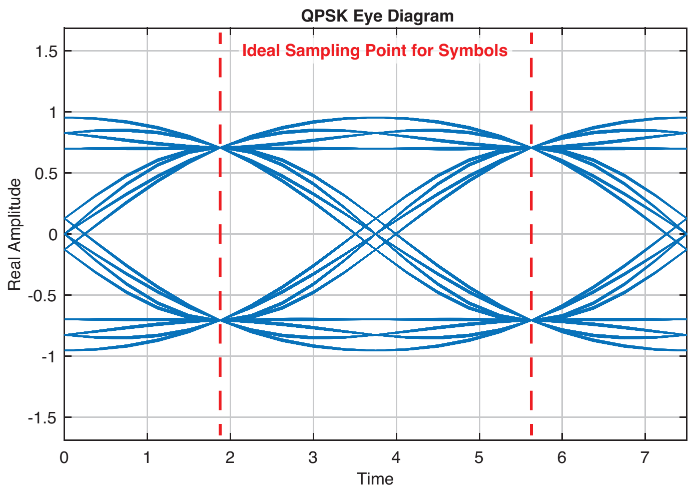
- Allowing some filtering to be implemented in the digital domain, including the matched shaping filter.
- Allowing the identification of the correct symbol sampling point and interpolating the result completely in the digital domain.
- Providing frequency headroom for dealing with CFO in the digital domain.

An example of multiple symbol waveforms for the QPSK modulation scheme after lowpass filtering is presented in Figure 2.5. The signal takes a different path depending on the sequence of symbols and the length of the lowpass filter.

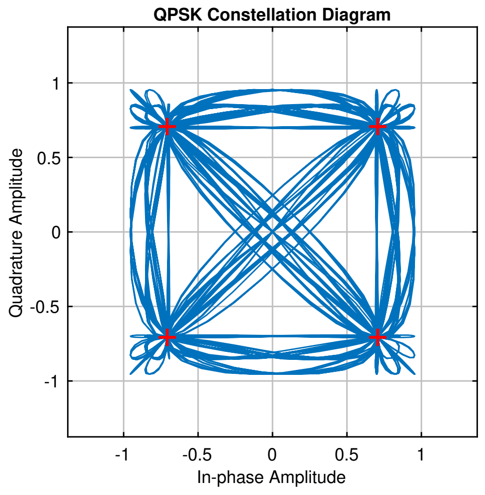
The ratio of the data converter sampling rates to the symbol rates is referred to as the oversampling factor[8].

emissions or effect the time domain shape of the symbol.

[8]Sampling at the symbol rate is theoretically possible but hard to implement. The highest frequency component of the transmitted symbol, assuming proper smoothing, would be at half the symbol rate. This would occur with alternating and opposite constellation points being sent. This would produce a sinusoid with half the frequency of the symbol rate. This can be seen in the time domain eye diagram in Figure 2.5a where transitions between two opposite symbols (along the real axis) results in half a sin wave



(a) Eye Diagram (Real Component)



(b) Constellation Diagram

Figure 2.5: Example of Transitions Between QPSK Constellation Points After Lowpass Filtering

2.1.4 Impairment Effects on Baseband Signal

Given that much of the correction for the impairments presented in subsection 2.1.1 can be accomplished in the digital domain, it is important to know how these impairments effect the baseband signal.

Noise

One of the most fundamental impairments to radio systems is noise. Noise is present in every radio system and is practically impossible to eliminate. There are often multiple sources of noise including thermal noise which can be injected and amplified at different points in the radio chain. One of the most analyzed channel models is Additive White Gaussian Noise (AWGN) which models noise as a Gaussian distributed random variable with a flat frequency spectrum which is added to the signal. The ratio of the signal power to the noise power is referred to as the Signal-to-Noise Ratio (SNR) and is typically expressed in dB. An example of QPSK symbols in an AWGN channel with an SNR of 15dB is presented in Figure 2.6. The added noise results in the collection of received symbols forming point clouds around the desired constellation points. The lower the SNR, the wider these point clouds are, increasing the risk of a symbol crossing into the region of another constellation point and being incorrectly decoded. The probability of a decoding error is also dependent on the modulation scheme used with tightly packed constellation points having a higher probability of error for a given SNR.

The use of Gaussian random variables allows AWGN to be analyzed analytically, some-

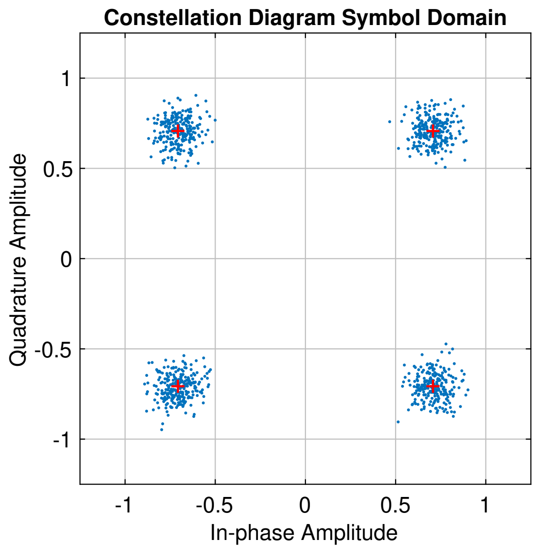


Figure 2.6: QPSK in Additive White Gaussian Noise (AWGN), 15 dB SNR

thing usually infeasible for more complex channel models. One of the key insights provided by the AWGN model is the theoretical capacity limit of a wireless channel, known as the Shannon Capacity [77]. It is also possible to quantify the expected bit error rate for most well-known modulation schemes operating in AWGN channels [77]. Matlab provides a function, berawgn, which returns these theoretical bit error rates for a given signal to noise level expressed as EbN0 [81], [82]. Designers typically benchmark their radio's performance against the theoretical bit error rate to determine if further design iterations are warranted. Bit Error Rate (BER) vs. SNR[9] for different modulation schemes are shown in Figure 2.7. Because higher order modulation schemes (with more bits per symbol) have higher densities of constellation points, their BER performance is worse for a given SNR. Radios can only practically tolerate a certain BER while providing sufficient performance to the upper layers of the networks stack. As a consequence, most radio systems switch between modulation schemes depending on the SNR present in the environment.

[9]Note that there can be some confusion in terms surrounding SNR. In this document, SNR refers to Signal Power/Noise Power for the captured signal at the ADC. The noise power is integrated across the entire captured spectrum. If oversampling is used, the transmitted baseband signal will only occupy a subset of the captured spectrum. The Energy Per Symbol to Noise Power Spectral Density (EsN0) normalizes out this oversampling factor. Energy Per Bit to Nose Power Spectral Density (EbN0) further normalizes out the number of bits per symbol and is often used as a method to compare performance across modulation schemes and radios. Both are presented for convenience as SNR can be intuitively easier to understand and measure while EbN0 is often preferred by communications specialists.

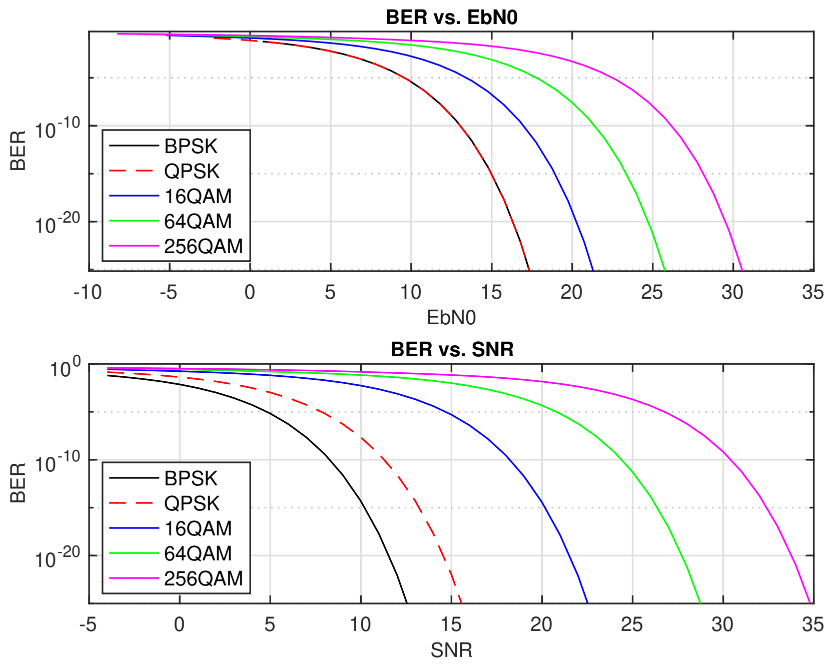


Figure 2.7: Performance Limits for Different Modulation Schemes (With 3x Oversampling)

Symbol Timing Error

As was mentioned in subsection 2.1.1, most wireless systems do not have synchronized transmitter and receiver oscillators/clocks. A properly lowpass filtered signal has an ideal point to sample a given symbol, as shown in Figure 2.5a. Outside of this point, the signal is transitioning between symbols. If sampled outside of the optimal sampling point, some of this transition is captured. Sampling with a relatively small offset from the optimal sampling point is shown in Figure 2.8. The symbol timing offset resultes a wider point cloud around the constellation points, similar to the effect of noise. However, the point cloud is more structured and is the result of sampling the different transition paths between constellation points as shown in Figure 2.5b. If there is a frequency offset between the transmitter and receiver symbol clocks, the point cloud will widen and contract over time according to point in the transition between constellation points being sampled.

Carrier Offset and Doppler Shift

A similar problem to symbol timing error, as presented in section 2.1.4, is carrier phase and frequency offset. Carrier offset, like symbol timing offset, occurs when the transmitter and receiver oscillators feeding the mixer stages are not synchronized. The less severe case occurs when the oscillators are locked in frequency but have a phase difference. A carrier

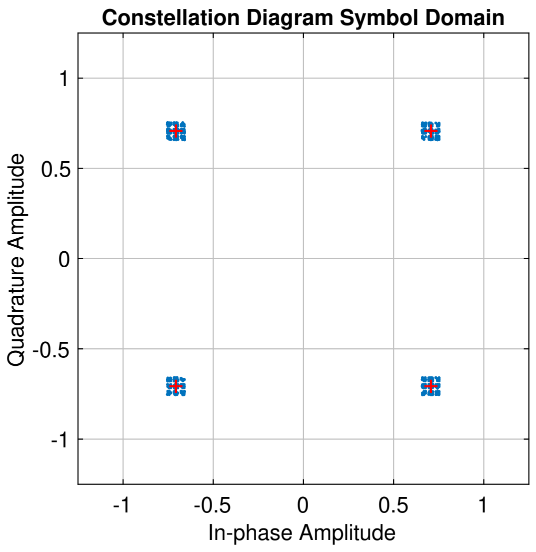


Figure 2.8: QPSK in the Presence of Symbol Timing Phase Error

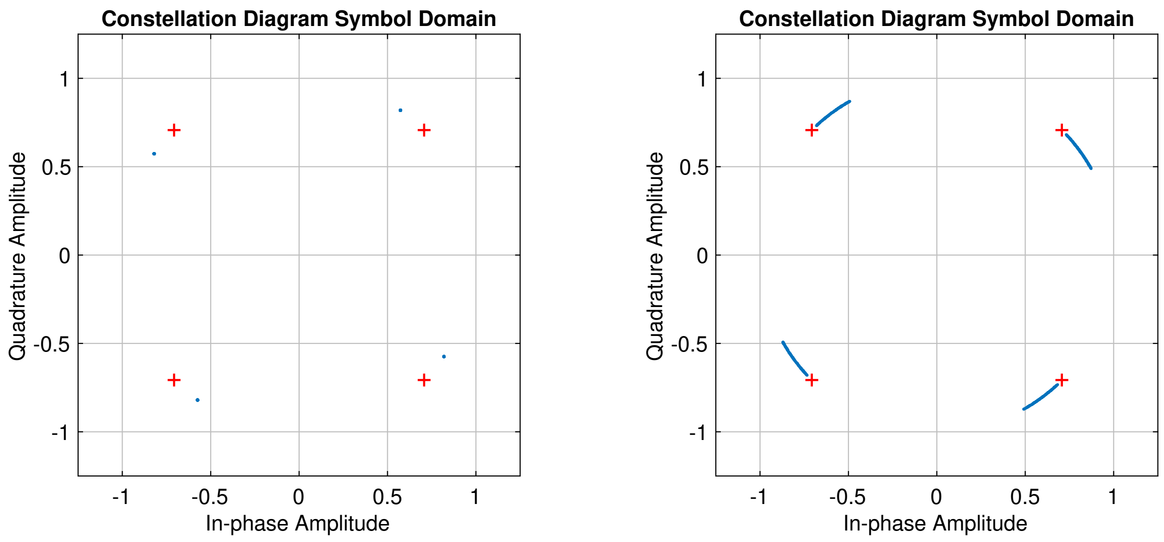
phase offset results in the signal being rotated about the origin in the receiver I/Q plane after analog QAM demodulation[10]. The result of a small carrier phase offset is shown in Figure 2.9a with the received symbols being rotated away from their corresponding constellation points. If the transmitter and receiver oscillators have a frequency offset, Carrier Frequency Offset (CFO) is introduced. This appears like a time-varying phase error between the Tx and Rx, resulting in the received signal continuously rotating about the origin of the I/Q plane. An example of a small CFO is shown in Figure 2.9b.

One other common cause of CFO occurs when radios are in motion relative to each other. This motion causes an offset of the signal in the frequency domain due to the Doppler shift. This CFO component can be time varying as the relative velocity between the Tx and Rx changes.

2.1.5 Multi-path Fading and Frequency Selective Channels

Up to this point, the only consideration of the physical environment has been the distance between the Tx and Rx, which affects the received power level. In reality, most radios experience additional impairments from the environment they are placed in. Radio waves, like light rays, can reflect off and refract through different materials. As a result, the energy traveling from the Tx to the Rx can arrive via multiple paths. The direct path between the Tx and Rx is typically referred to as the line-of-sight path. Without obstructions, it is typically the strongest path. Other propagation paths cover longer distances, delaying the

[10]For a detailed explanation and derivation of this effect, see the appendices in [10].



(a) Carrier Phase Offset (b) Carrier Frequency Offset (CFO)

Figure 2.9: QPSK in the Presence of Carrier Impairments

signal reaching the receiver. They also tend to be weaker due to energy lost either through reflection or refraction. These different paths all sum together in the air at the Rx antenna.

The result at the receiver is the sum of different delayed and attenuated versions of the signal. This is the same mechanism which occurs in a Finite Impulse Response (FIR) filter, which sums scaled and delayed versions of a signal. Just as FIR filters can change the frequency response of a signal, the different reflections in the environment can cause the frequency response of the signal at the Rx to be non-flat. The result is the distortion of the received signal. There are multiple terms for this phenomenon, with *multipath fading* and *frequency selective channel* being two common ones. Various multipath models have been developed in an attempt to represent what radios may experience in the real world. For more information on multipath fading, see [83].

2.2 Baseband Signal Processing with Cyclops

The purpose of radio baseband signal processing is to produce a signal to be sent out of the DAC on the Tx side and to interpret the signal received by the ADC on the Rx side. Any impairments that remain uncorrected in the analog domain must be resolved by the baseband signal processing. Cyclops is an example of a radio baseband that handles most impairments in the digital domain. Originally implemented on an FPGA for a 60 GHz test/development platform and documented in the author's master's thesis [10], the design was modified and extended for Berkeley's DARPA Spectrum Collaboration Challenge (SC2) effort. After the conclusion of Berkeley's run in SC2, the design was further modified and improved as a demonstrator for mapping baseband designs onto general-purpose CPUs [73].

Cyclops assumes that it is connected to an analog QAM modulator and demodulator, as described in subsection 2.1.2. The common convention of representing the signals exchanged with the dual channel data converters as being complex (having both a real and imaginary component) is taken. Cyclops is best described as a *single carrier*[11] radio which modulates a stream of data into symbols which are then low pass filtered based on the symbol rate and sent to the DAC.

Designed without a particular MAC in mind, Cyclops provides most compensation for the physical channel in the receiver[12]. As a result, the bulk of the signal processing workload is done by the receiver.

2.2.1 Transmitter

The Cyclops transmitter has two main functions:

- Modulate data into a target modulation scheme.
- Perform Tx pulse shaping (the Tx side of the matched filtering operation).

Framing, including preamble insertion, occurs outside of the Tx DSP with symbols and their associated modulation schemes being sent to the transmitter. A high level of the Cyclops Tx baseband processing is shown in Figure 2.10.

2.2.2 Receiver

As mentioned above, Cyclops was designed with most of the signal processing workload contained in the receiver. A general outline of the Cyclops receiver is shown in Figure 2.11 with data generally flowing from the ADC through the following processing steps:

1. Receive Matched Filtering - correlating against the expected pulse shape of the signal and reducing inter-symbol interference
2. Automatic Gain Control (AGC) - Normalizing the power of the received signal to an expected level

[11]A popular alternative to single carrier radios is Orthogonal Frequency Division Multiplexing (OFDM) based systems. While these two types of basebands take different approaches, especially in modulation/demodulation and equalization, they share some common challenges. For instance, both single carrier radios and OFDM radios are sensitive to CFO. While there are some additional opportunities for exploiting parallelism across sub-carriers post FFT in OFDM, the signal processing techniques explored in the single carrier radio are still relevant to OFDM based systems. For more information on OFDM, see section A.1.

[12]Some radio systems perform pre-distortion at the transmitter to compensate for the channel characteristics. Pre-distortion requires estimates about the channel characteristics at the receiver to be sent back to the transmitter in a timely fashion to be effective.

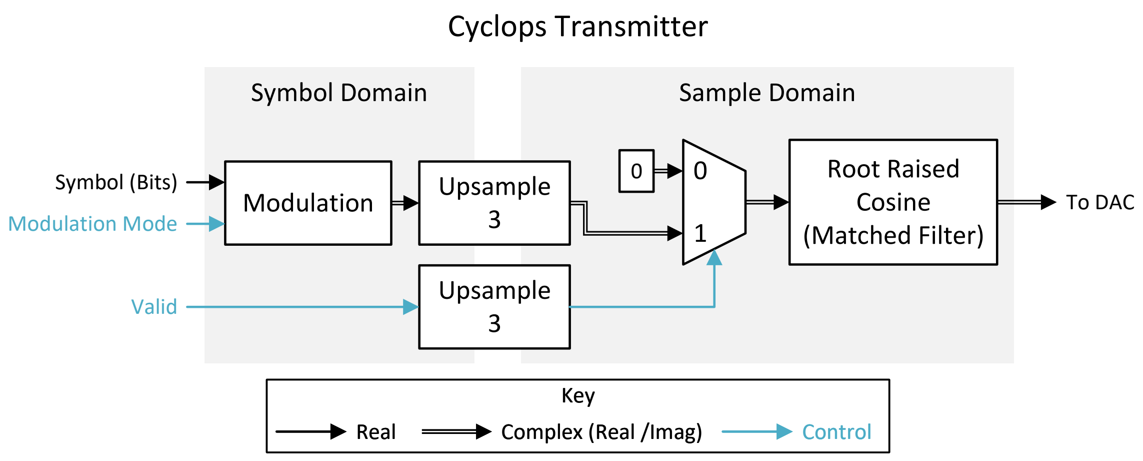


Figure 2.10: Cyclops Tx: Overview

3. Timing Recovery - Estimating and correcting for differences in the Tx and Rx symbol clocks. Transitions from samples to symbols

4. Coarse CFO Correction - Estimates and Corrects for CFO

5. Equalization and Demodulation - Estimates and corrects for reflections in the environment. Corrects for residual CFO and carrier phase error. Demodulates header and payload data

6. Data Packing - Packing data from different modulation schemes into bytes to send to the receiving program

The result at the end of these steps, assuming successful frame detection, is header and payload data. The bulk of these processing steps are devoted to addressing the impairments presented in subsection 2.1.1. As these steps represent the computational workload being mapped to the CPU, a brief description of the larger units of the design are described below.

AGC

The AGC is responsible for normalizing the power of the incoming signal to a desired level[13]. This simplifies some of the downstream processing blocks by allowing them to assume

[13]Many AGCs control the analog gain of a front end amplifier in the RFIC to maximise the dynamic range of the signal captured by the ADC. While the algorithm used in Cyclops could be used to modify an external amplifier, it presently only scales the digital value received by the ADC. A compelling approach to maximize the dynamic range of the signal would be to use a hybrid scheme with an ADC running in a closed-loop fashion in the RFIC.

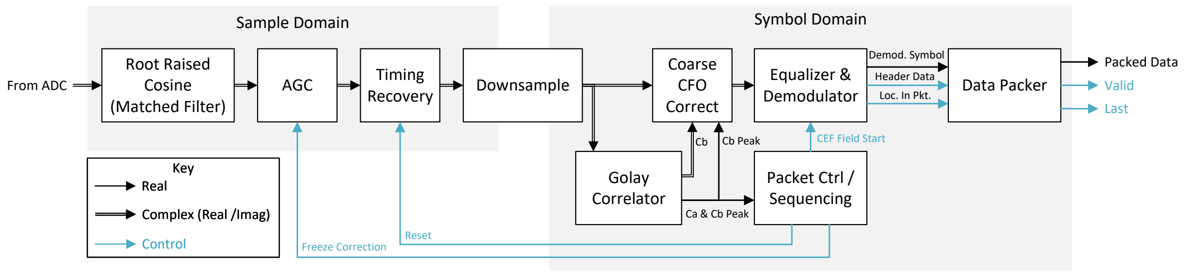


Figure 2.11: Cyclops Rx: Overview

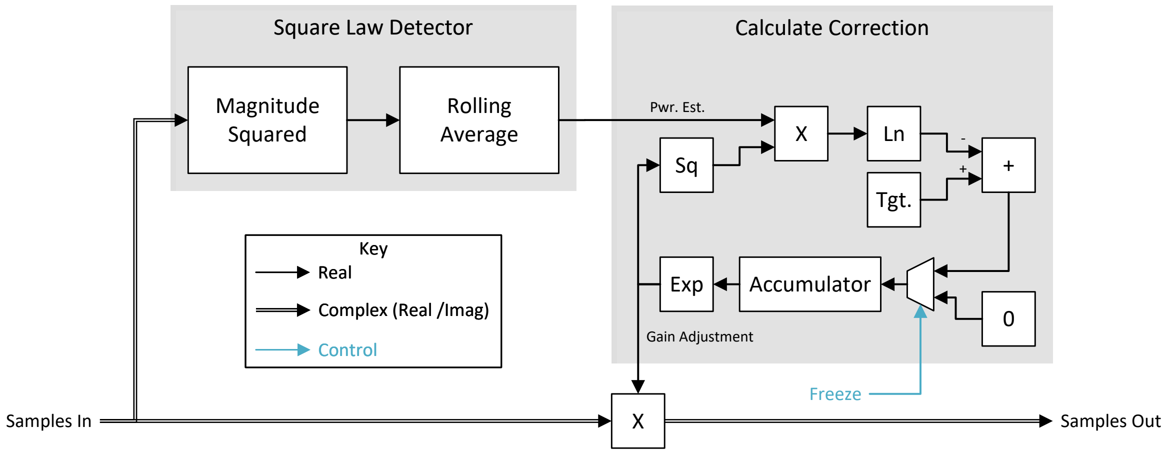


Figure 2.12: Cyclops Rx: AGC

a fixed power level once the AGC settles. Blocks that rely on this property include peak detection in Timing Recovery and equalization through the step size parameter. There are multiple AGC topologies available and the one implemented approximates a log-exponential design. An advantage of the log-exponential algorithm is that the time to settle should be constant regardless of power of the incoming signal [84]. This provides some degree of certainty on how much time other blocks have to settle once a signal is detected.

The log-exponential design is similar to the one used in an earlier version of the Cyclops radio [10]. One change is that the estimation/correction control no longer includes the receive matched filter in the loop, which has been relocated to before the AGC. This move was practical once the number representation was changed from fixed-point arithmetic in the early versions of Cyclops to floating-point in this version. The control loop also no longer includes the power detection and averaging logic. The result of these modifications is that the core AGC control loop contains less logic and less delay. In addition to improving the control system by avoiding unnecessary delay in the feedback chain, the reduction in workload within the control loop reduces the need to partition across the loop.

To reduce the workload further, lookup tables (LUTs) are used to approximate the Ln and Exp functions. These tables are implemented with a limited resolution and range which, in principle, allows fast lookup.

Timing Recovery

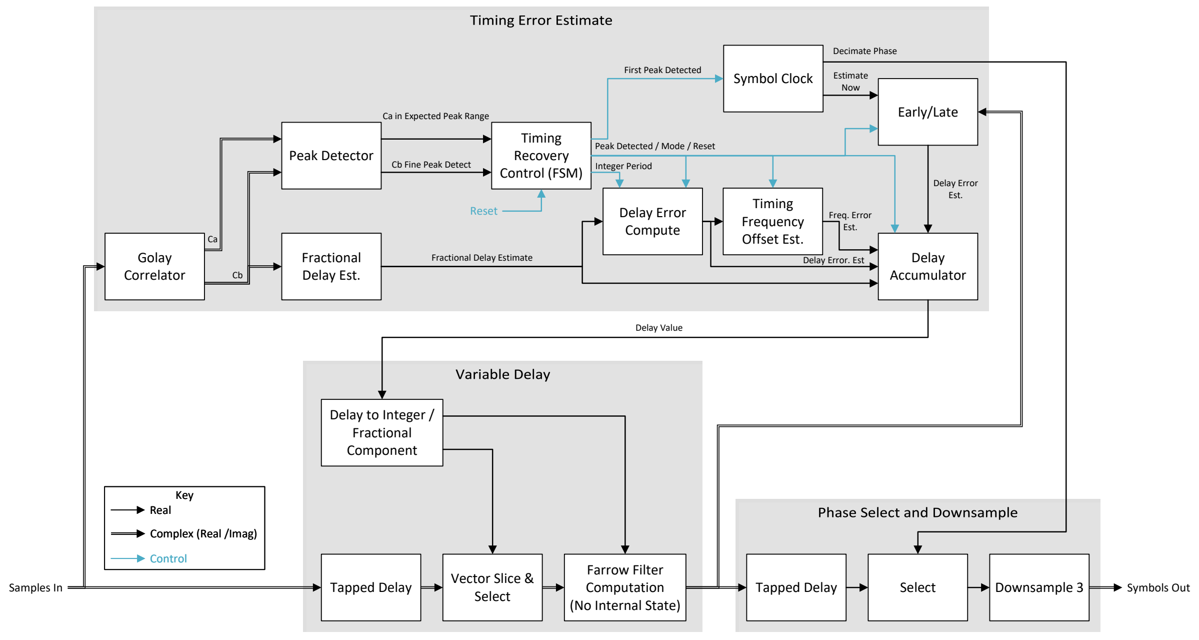


Figure 2.13: Cyclops Rx: Timing Recovery

Timing Recovery is the most computationally intensive block in the Cyclops baseband. This is due both to the large number of operations used to perform timing recovery and the fact that this block runs at the sample rate rather than the symbol rate.

The Timing Recovery block has changed significantly from the non-data-aided approach used in early versions of the Cyclops baseband [10]. The timing recovery block employs a two-phase coarse/fine correction strategy to correct for both symbol clock frequency and phase differences. The coarse correction is feed-forward and leverages properties of the preamble. Like early versions of the Cyclops baseband [10], the preamble is based on the control PHY preamble from 802.11ad [85] and consists of repeated Golay sequences[14] [85]. The preamble

[14]Golay sequences come in complementary pairs, denoted as the A sequence and the B sequence. In addition to producing relatively small false peaks and a small DC component, Golay sequences have an additional property where the false peaks of the correlations of the A and B sequences cancel when summed

is BPSK modulated and, like 802.11ad [85], is separated into 2 segments, called the Short Training Field (STF) and the Channel Estimation Field (CEF). The Golay correlations of the different segments of the preamble is shown in Figure 2.14. During the short training field, the B Golay sequence is repeated several times. Once the first peak is detected, the Timing Recovery block begins estimating the timing error. The number of samples until the next peak along with information about the shape of the correlation peak is used to estimate the timing phase difference between correlation peaks. The timing differences between adjacent peaks are averaged to produce an estimate of the symbol timing frequency error. In the time between correlation peaks, the frequency estimate is used to interpolate the desired timing delay to realign the received signal. At each correlation peak, the requested timing delay is reset to the newly estimated timing phase.

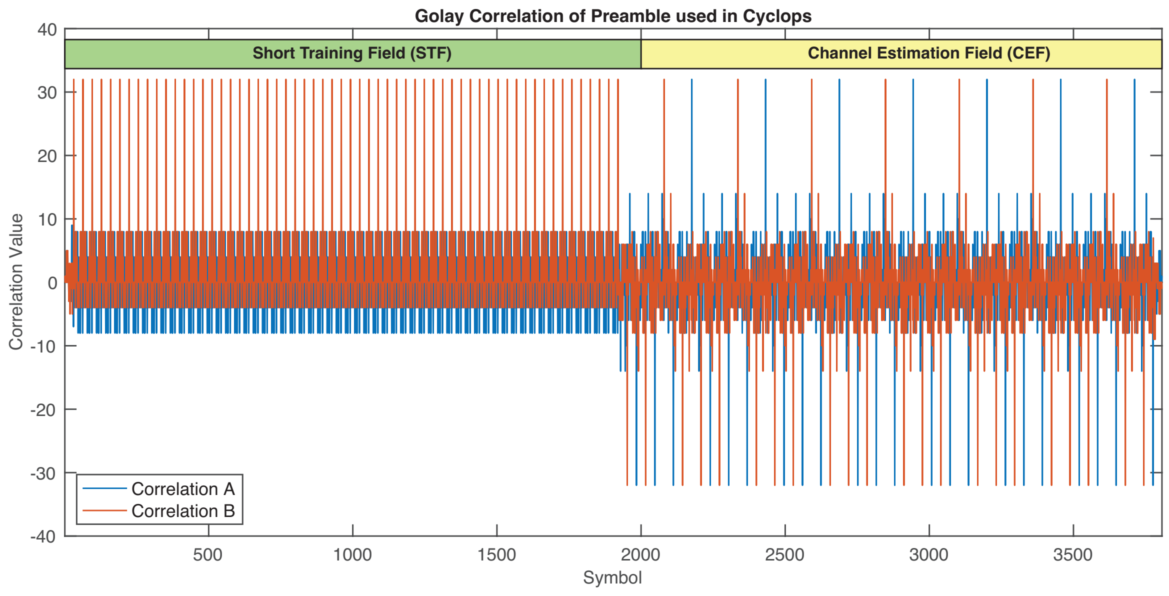


Figure 2.14: Cyclops Preamble - Based on 802.11ad's Preamble [85]

The timing delay estimate, which is computed in the Delay Accumulator block shown in Figure 2.13, is used to drive a variable delay subsystem. This subsystem is capable of delaying the incoming signal by integer numbers of samples as well as interpolating fractional delays between samples via a Farrow Filter structure[15]. The Farrow Filter used is similar to the one used in [10]. The core difference is that the Farrow Filter in the current Cyclops baseband does not have any internal state. Vectors sized for the Farrow structure are sliced

[86]. See section 4.6 of [10] for more information about the properties of Golay complementary sequences. Compared to the mmWave version of Cyclops, the number of repeated peaks has been adjusted and a 32 symbol Golay pair is used instead of a 128 symbol Golay pair.

[15]For information on the structure and function of the Farrow Filter, see [87]

from a separate tapped delay block. The position of the slice selects the integer delay, and the Farrow structure provides the fractional delay interpolation. This is due to a major change since early versions of Cyclops where enabled subsystems (clock enabled subsystems) were used along with a sampling clock to provide the integer delay portion of the variable delay and decimation. The downside to this approach was that it prevented some optimizations which can occur when a fixed decimation is known. Because the Tx symbol clock could be faster than the Rx clock, it was possible for earlier versions of the Cyclops radio to occasionally pass an extra symbol. The new version of the Cyclops baseband accommodates this possibility by provisioning a tapped delay large enough to accommodate the maximum expected symbol timing offset across the entire packet. Because the symbol phase of the received packet is not known a priori and can change from packet to packet, a second tapped delay and selector is used before decimation to select the appropriate decimation phase. With these modifications, the implemented decimation from sample to symbol domain can be fixed. For more information on the advantages of this design, see section 8.4.

Once the end of the STF is detected, the timing recovery system shifts into a feedback mode. The frequency estimate is frozen[16] and is used to interpolate the correct delay value for subsequent samples/symbols. If the phase estimate and frequency estimate were perfect and the underlying values were constant, this would be sufficient. However, in practice, the estimates are not perfect and timing phase and frequency may potentially drift. To handle these cases, an early-late detector is used to derive timing estimates[17]. The early-late detector compares the correlation of the pulse shape of the current sample against the sample before and the sample after. On average, the correlation before and after should be equal in magnitude with the pulse shape correlation expected to be symmetric. An imbalance, on average, indicates that the sampled value is either early or late. This correction is averaged, run into a proportional integral (PI) block, and is then used to drive additional correction to the requested delay value for the variable delay. By running the early-late detector on the result of the variable delay, it creates a PI feedback control system. So long as the timing frequency error estimate is close, the changes required from the early-late control system are small, allowing it to have delay in the feedback path.

Coarse CFO Estimation/Correction

The Coarse CFO estimation/correction is the first part of a coarse/fine correction mechanism for CFO. Like Timing Recovery, the coarse CFO block uses the repeated Golay sequences in the STF portion of the preamble (shown in Figure 2.14) to estimate the CFO. Unlike timing recovery, which uses the correlation shape of the peaks, the Coarse CFO block uses the relative phase of subsequent correlation peaks to estimate the CFO. The estimate starts when the first correlation peak in the STF is detected. The Coarse CFO block FSM

[16]The correlation shapes at the boundary of the STF and CEF are slightly different due to the new data pattern, resulting in a bad final estimate. These final estimates are discarded before the timing frequency error estimate is frozen.

[17]For more information on early-late detectors and early-late symbol timing recovery, see [88].

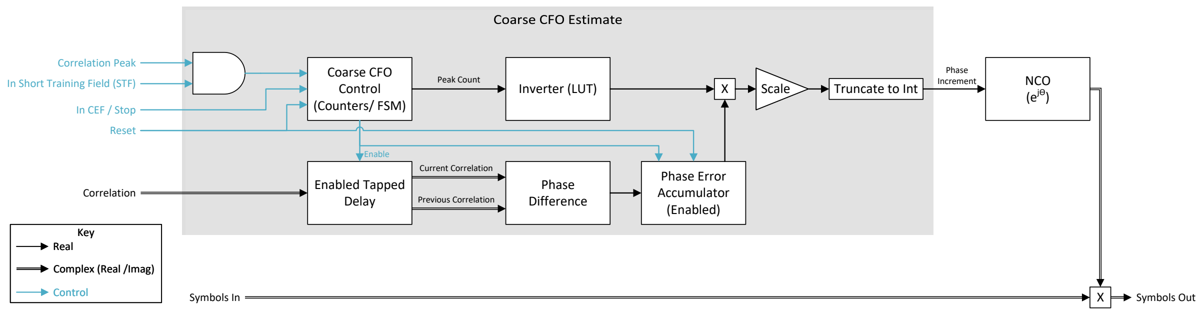


Figure 2.15: Cyclops Rx: Coarse CFO

then enters a timekeeping mode, where it samples the correlation when it expects a peak. This allows it to avoid issues if peak detection does not trigger in the presence of noise. The CFO estimate is then transformed into an integer phase increment signal fed to a digital Numerically Controlled Oscillator (NCO). The NCO outputs a complex exponential ($e^{j\theta}$) oscillation with a frequency described by the phase increment. It uses a phase accumulator and quarter-wave lookup table to generate the oscillation. See [89] for information on the structure of NCOs. This oscillation is applied to the incoming signal to stop the rotation of the constellation as described in section 2.1.4. In practice, the estimation has some error that results in the constellation continuing to rotate, but very slowly. The residual CFO error along with the carrier phase are corrected in the downstream equalization block.

Equalization and Demodulation

Equalization serves multiple purposes in the Cyclops baseband. As its name suggests, it aims to correct for frequency selective fading in the RF environment as described in subsection 2.1.5. Within the Cyclops radio, it also serves the additional function of correcting any excess CFO error that was uncorrected by the Coarse CFO block as well as correcting any carrier phase offset.

The core algorithm used by the Cyclops implementation of the equalizer (EQ) is LMS which aims to minimize the Least Mean Squared error of the received signal by modifying the taps of an FIR filter [90], [91]. LMS is an iterative stochastic[18] gradient decent algorithm which converges *in the mean* to the desired filter coefficients[19] [90]. There are multiple variations on LMS, and Cyclops uses two of them. The first is to introduce a variable step size which decreases over time[20]. This aims to reduce the amount of coefficient jitter about

[18]Because LMS an approximation of the true gradient decent algorithm for a sample mean of one [90], it can be described as *stochastic* gradient decent algorithm by the definition given in [92].

[19]Under certain conditions including the assumption that the environment is a Wide-Sense Stationary (WSS) process.

[20]The method used in Cyclops uses a static step size reduction instead of the more complex method

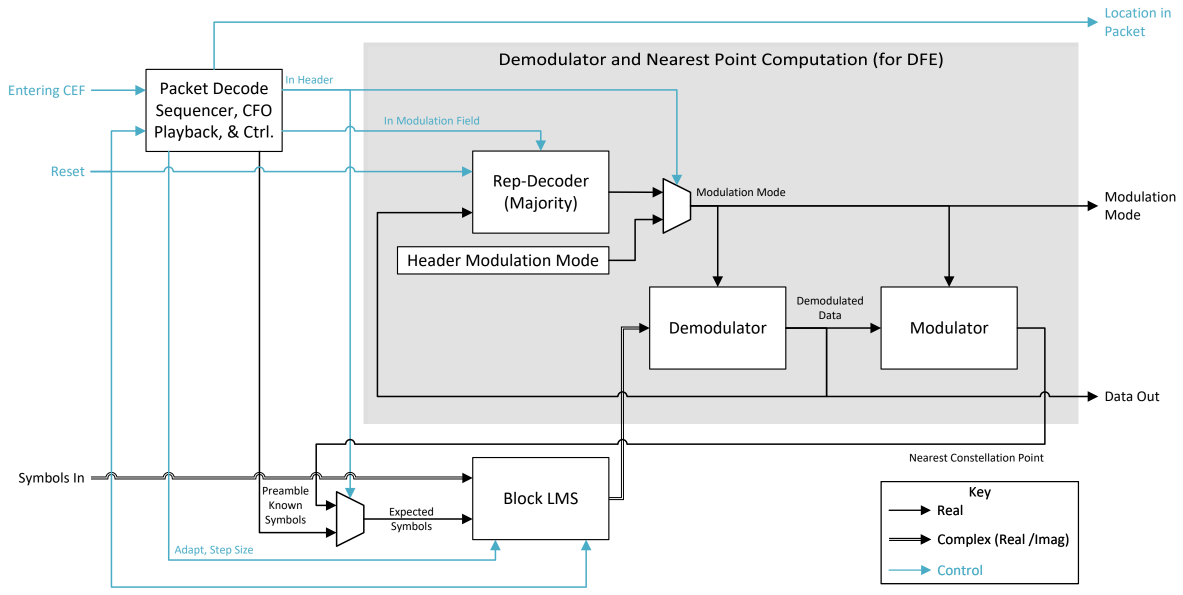


Figure 2.16: Cyclops Rx: Equalization and Demodulation

the desired point while still reaching the neighborhood of the solution in a reasonable amount of time. The other variation is Block-LMS [93], [94] which does not adapt the filter coefficients after each sample. Instead, the coefficients are held constant for multiple samples with the coefficient update accumulated and applied after the block. This modification helps weaken the dependencies in the feedback loop, allowing more independent processing to occur. The benefits of this approach will be discussed in more depth in chapter 8.

Like the Timing Recovery block, the EQ block operates in two distinct modes depending on the location within the packet current being received. At the transition between the STF and CEF in the preamble, the ideal CEF is played back as the reference signal to the LMS algorithm. For the rest of the preamble, the LMS trains against the known CEF field. After the preamble, the EQ shifts into decision directed mode which assumes the closest constellation point to the received point is correct and trains based off that assumption. This allows the EQ to adapt to slowly changing channels as well as to continue correcting for small residual CFO error. Because demodulation is required for decision feedback EQ, demodulation also occurs within the EQ block rather than being separated into a different logical unit. As part of its demodulation responsibilities, the EQ is also responsible for decoding the modulation field in the header, which indicates the modulation scheme used for the payload portion of the packet. The modulation field, which is the first part of the BPSK encoded header, is encoded with a rep3 repetition code. This code repeats each bit of

discussed in [90].

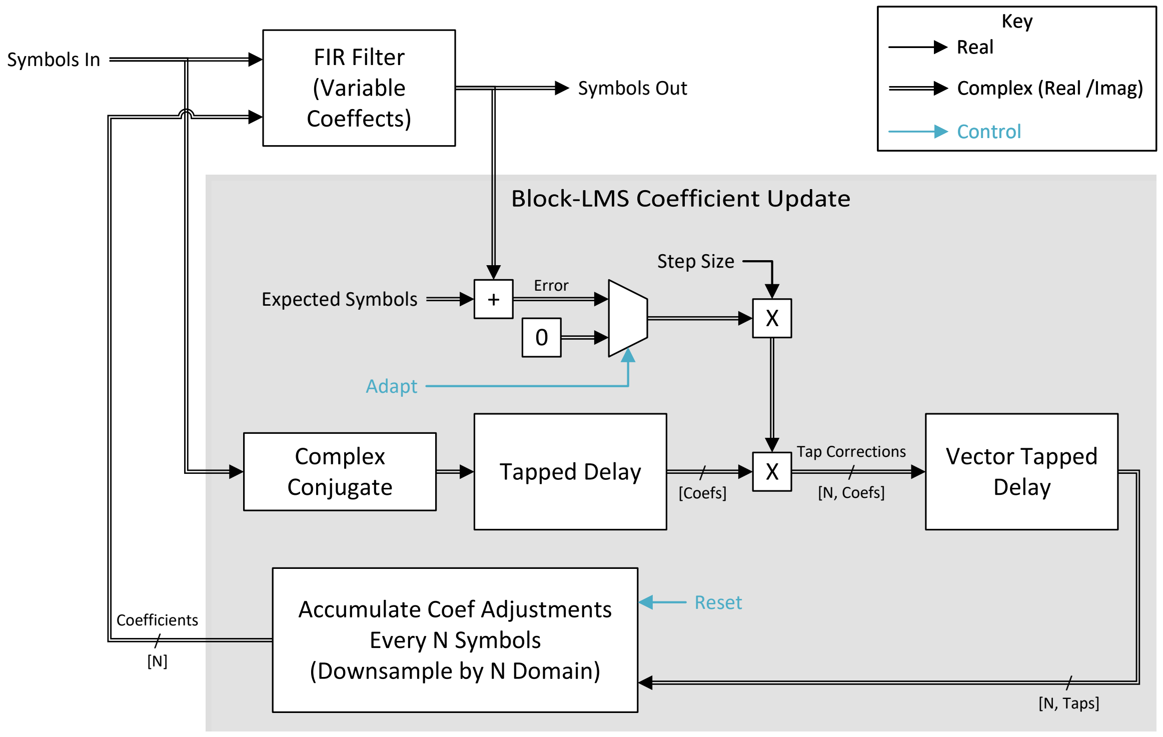


Figure 2.17: Cyclops Rx: Block LMS

the modulation field three times. The Rep-Decoder block performs the majority operation to decode the modulation field. If two or more of the repetitions are 1, the decoded bit is 1. Likewise, if two or more of the repetitions are 0, the decoded bit is 0. This encoding allows the modulation scheme to be correctly decoded with single errors of each bit. If the modulation field has more errors, the incorrect modulation scheme will be decoded, and the packet will effectively be lost. Failure to decode the modulation field correctly is considered a catastrophic failure condition for an individual packet, similar to cases where estimation/correction blocks fail to converge.

Packet Control and Sequencing

Because several different blocks within the Cyclops Rx shift modes depending on what part of the packet is being received, there needs to be some mechanism responsible for tracking the different stages of packet reception as well as sending control signals to the relevant blocks in the Rx chain. The Packet Control / Sequencing Block performs this logic in the symbol domain portion of the baseband by analyzing information from the preamble correlators. It in turn sends signals to the EQ, Timing Recovery, and AGC blocks to signal to them when to change modes. Over multiple revisions of the Cyclops baseband, some of the functions of the Packet Control and Sequencing block have been distributed throughout the radio. For example, some sequencing logic is now contained in the EQ and some of the logic is replicated in the controller for the Timing Recovery block.

2.2.3 Performance

The Matlab/Simulink simulated BPSK and 256QAM performance of the Cyclops baseband in an AWGN channel is shown in Figure 2.18 and Figure 2.19 respectively. The performance is generally good with performance typically well within 1 dB of the ideal BER. One exception is under low SNR conditions the radio suffers elevated bit errors and packet decode errors, which are shown in Figure 2.20. This is likely due to the various estimation and correction components failing to settle to the correct values in the low SNR case. Simulations with Timing Error and CFO were also conducted to verify the performance of the Timing Recovery, Coarse CFO, and EQ blocks. Results from these simulation runs generally match the results without Timing Error or CFO which indicates that the radio can cope with moderate imperfections present in real-world operation. For more information on the methodology behind the BER sweep as well as the performance for all payload modulation schemes, see section A.2.

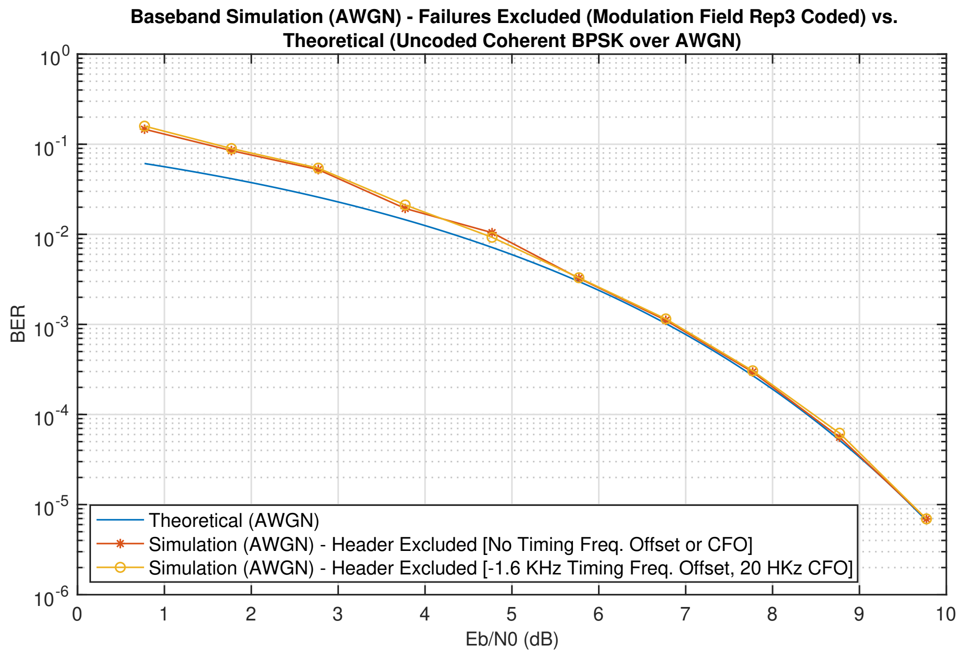


Figure 2.18: Cyclops Bit Error Rate Performance - BPSK

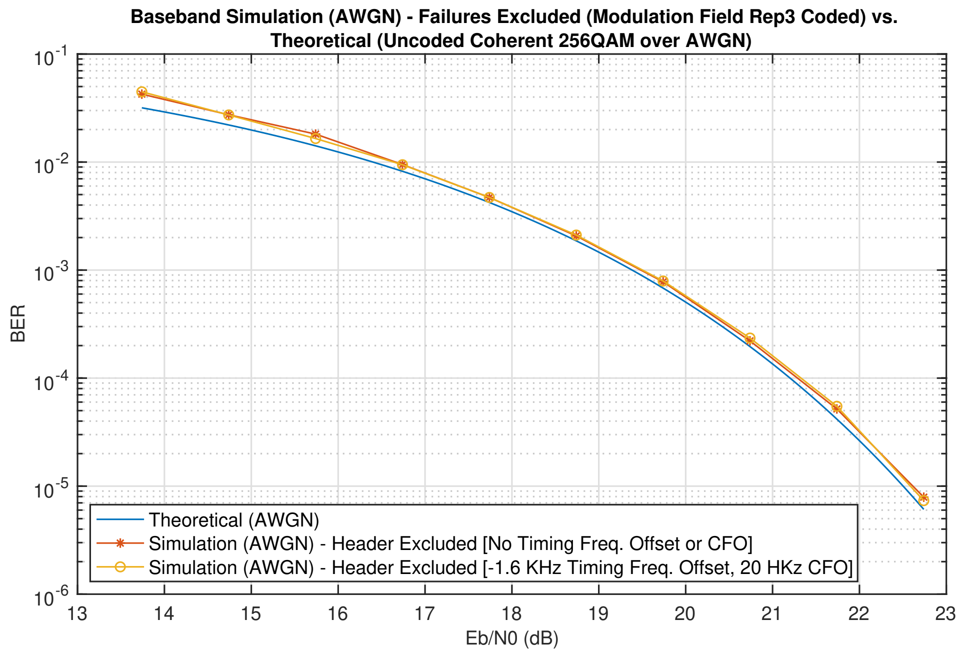
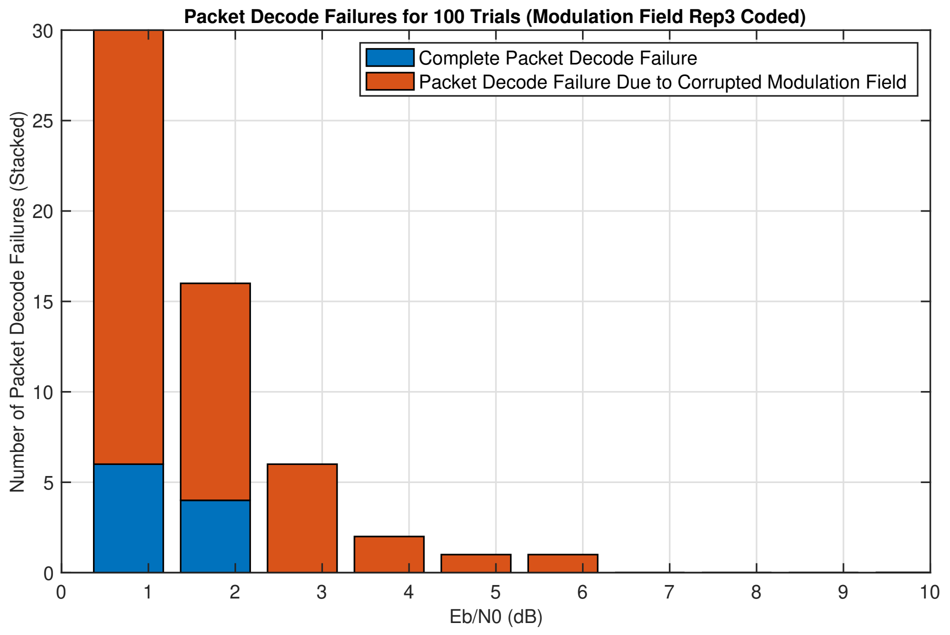
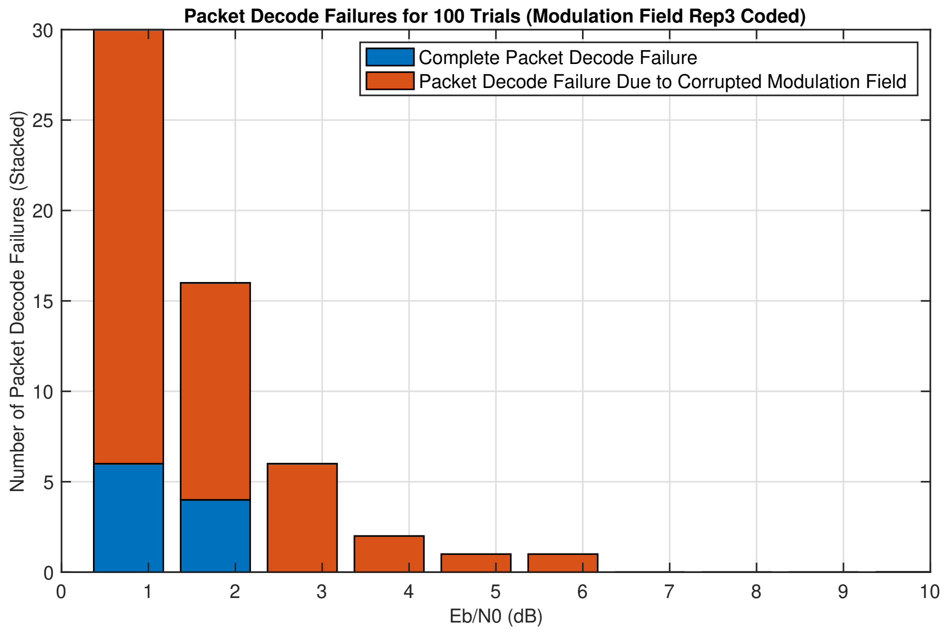


Figure 2.19: Cyclops Bit Error Rate Performance - 256 QAM



(a) BPSK, No Timing Frequency Error or CFO



(b) BPSK, Timing Frequency Error -1.6 KHz, CFO 20 HKz

Figure 2.20: Cyclops Packet Detect Failures and Modulation Field Decode Errors

Chapter 3

Modes of Parallelism: CPUs

There are 2 basic ways to increase the performance of a CPU: increase the speed of performing an operation or do more operations at once. Each has its own set of advantages and disadvantages.

Prior to the 2010s, it was common for CPU vendors to increase the clock speed of their parts year-over-year. This was thanks to semiconductor technology shrinking the size of transistors (known as transistor scaling [95]), which let them switch faster. But as transistors shrank, the wires connecting them shrank as well, increasing their electrical resistance[1]. The industry followed two paths:

- Dennard Scaling: Linearly improving delay and keeping power density constant by lowering the supply voltage as transistor dimensions shrink [95].
- Frequency Scaling: Quadratically improving delay and cubically increasing power density by keeping the supply voltage constant while transistor dimensions shrink [95].

Frequency scaling was popular due to the quadratic improvement in speed. However, there was a heavy price to pay for this improvement in the form of rapidly growing power density. Eventually, CPU vendors encountered what is often referred to as a *power wall* where power and heat limitations blocked further use of frequency scaling.

Another effect of semiconductor process scaling is that the number of transistors per unit area increases as transistor dimensions decreases. The rate of density improvement was famously observed by Gordon Moore as the number of transistors increasing every 1-2 years [95], [96]. The general trend of increased transistor density described by Moore's Law has continued as time has gone on. However, the speed at which density is increasing has slowed with the increased challenge to scale semiconductor devices further. While the end of Moore's Law has been predicted for a long time, process technology experts have continued to find new ways to increase density such as the development of the FinFET [97]. While Moore's Law will likely come to an end at some point, other techniques are being actively utilized to

[1]This increased wire resistance from interconnect scaling is not typically a problem at short distances but can become problematic for long lengths [95].

increase the amount of logic in modern CPUs. Several modern CPUs utilize multiple silicon dies with high-speed interconnects using new packaging and interposer techniques. Utilizing multiple dies also has the benefit of increasing the effective component yield vs. a large single die, reducing component costs [98].

With the effective end to frequency scaling but the possibility of inserting more logic into the same area, many of the CPU performance improvements over the past several years have involved performing work in parallel. Unlike frequency scaling which improves all workloads, parallel resources require that tasks be broken into independent work units to provide performance improvements. Effectively extricating parallelism from the DSP design and mapping it to the different modes of parallelism available on the CPU is at the core of this project and is critical to attain the required performance.

While there are many Instruction Set Architectures (ISAs) available today including ARM (and its variants), RISC-V, MIPS, and IBM Power, x86_64 is especially pervasive. x86, and its extensions, have held the dominant positions for personal computing and servers for some time, with Intel and AMD being the top producers. At the inception of this project x86_64 based CPUs enjoyed high performance and were aggressive with core counts at the high end. At the time, Intel had released 28 core server parts [99] and AMD had announced 32 core consumer and server parts. The core count has continued to increase with Intel releasing a 58 core server part [99] and a 18 core consumer part [100] and AMD releasing 64 core consumer and server parts [6], [101].

Due to its position in the server and HPC space as well as delivering many high-performance CPU cores, x86_64 was chosen as the target demonstration ISA for this project. Many of the techniques and insights of this project will map to other ISAs but there will be some x86_64 specific optimizations that will be mentioned.

3.1 Modern x86_64 CPUs

Modern x86_64 CPUs support multiple forms of parallelism, each of which requires different aspects from the target workload to be useful:

- Multiple Instruction / Multiple Data (MIMD) across cores.
- Single Instruction / Multiple Data (SIMD) vector execution units in cores.
- Superscalar, Out-of-Order, execution within cores.
- Pipelined, branch predicted, execution within cores.
- Simultaneous Multi-threading (SMT), execution within cores.

3.1.1 MIMD

The most flexible of these modes of parallelism is MIMD across cores. To put it simply, each core can run a separate program operating on different sets of data. The operating system (OS) on most computers takes advantage of this parallelism by scheduling different processes or threads onto different cores. For example, a desktop OS may schedule music streaming playback to one core and photo editing to another core. This model of parallelism works well if there are multiple independent or lightly dependent tasks that need to be run. To take advantage of multiple cores in a single application, the software developer needs to explicitly expose parallel workloads by either creating multiple threads of execution or by using a framework such as OpenMP.

Communicating work between cores typically occurs through the memory and cache subsystem, which will be discussed in section 3.2.

3.1.2 SIMD

Most Intel and AMD CPUs implement SIMD vector extensions to the x86_64 ISA. These extensions include MMX, SSE (multiple versions), AVX (multiple versions), and FMA. AVX and AVX2 support vectors that are 256 bits wide. Intel has implemented AVX-512 which supports 512 bit-wide vectors. The SIMD model is that a single operation is executed across multiple pieces of data. SIMD execution generally saves on overhead by amortizing the cost of an instruction across multiple operations. A common problem when converting a program to use SIMD/vector instructions occurs when operations occur conditionally. Many SIMD extensions support a set of masked operations which allows the user to specify which elements in a vector the operation should be performed on. Conditional operations can be handled through masked vector instructions but there is an added cost. If a branch is encountered with different elements in the vector taking diverging decisions, both paths need to be executed with masking[2]. For more information on SIMD see [16]. For more information on x86_64 ISA extensions, see [103], [104].

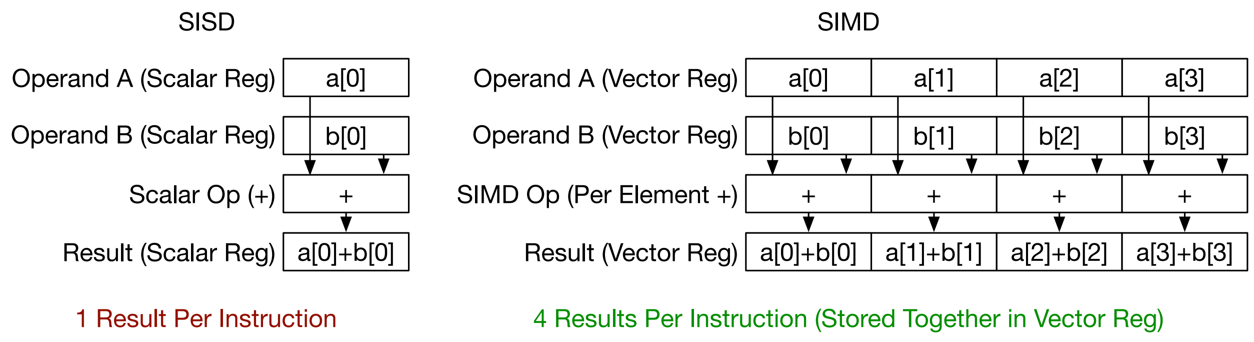


Figure 3.1: Example of Scalar Operation (SISD) vs. SIMD with a Vector Length of 4

[2]There is an analogous problem with GPUs. Nvidia GPUs, for example, execute GPU threads in warps. When a branch is encountered and threads within the warp take diverging control decisions, only a subset of the threads in the warp are allowed to execute at a time. To see how this manifests in several generations of Nvidia GPUs, see [102].

3.1.3 Out-of-Order and Superscalar

The naive view of a CPU is that it executes the instructions in a program one at a time in the order which they were written. While this is true for some CPUs, particularly in the embedded space where cost and power efficiency are of paramount importance, most x86_64 CPUs used in laptops, desktops, and servers can execute instructions out-of-order. There are many advantages to this approach such as allowing independent work to be executed while another instruction is waiting for an operation to finish. Analysis of instructions to be executed is performed so that instructions are only scheduled to be executed after any dependent operations finish. Re-arranging the results of different instructions can be accomplished in multiple way including using a re-order buffer or by maintaining a pool of registers and dynamically renaming them as instructions are executed. The operation queue, register renaming blocks, register files, and scheduling blocks of AMD Family 17h CPU cores are shown in Figure 3.2. For more information on the different techniques used and constraints on out-of-order CPU operation, see [16].

In addition to executing programs out-of-order, most x86_64 CPUs can execute more than one instruction at a time across multiple different execution units. Processors that are capable of executing more than one instruction at a time are referred to as superscalar. Superscalar operation typically complements out-of-order execution as it allows multiple independent instructions, which may not be contiguous in program order, to execute at once. When multiple independent instructions are executed simultaneously, the CPU is said to be exploiting *Instruction Level Parallelism (ILP)*. For a concrete example of a recent superscalar CPU, the AMD Family 17h Processors Models 30h and Greater (which includes the Ryzen Threadripper 3970X 32-Core Zen2 CPU[3]) has four integer Arithmetic Logic Units (ALUs), three Address Generation Units (AGUs), and four floating point / vector units per core. Execution units can generally execute a subset of the CPU's supported operations [17]. The different execution units in AMD Family 17h Models 30h and Higher cores is shown in Figure 3.2 along with the different operations they are capable of executing. For more details on the implementation of superscalar CPUs, see [16].

To take advantage of these different execution units, an ideal program would be composed of many independent or lightly dependent operations which can be executed simultaneously. When operator dependencies are present, it is best to make sure that enough independent operations are present in the look-ahead window (the set of instructions considered for scheduling) so that new instructions can be issued to the execution units in place of dependent operations which need to wait for results. Because functional units can be specialized to specific instruction types, the composition of operation types in the program can also influence achieved performance. As an example, the AMD Zen2 cores depicted in Figure 3.2 have two floating-point units capable of floating-point multiplies and two different units capable of floating-point additions. Assuming no outside bottlenecks or inter-instruction dependencies, a program with an interleaved 50/50 split of floating-point multiplies and adds should perform approximately 2x better than a program with just floating-point multiplies or adds.

[3]Ryzen Threadripper 3970X is CPU Family 17h, Model 31h

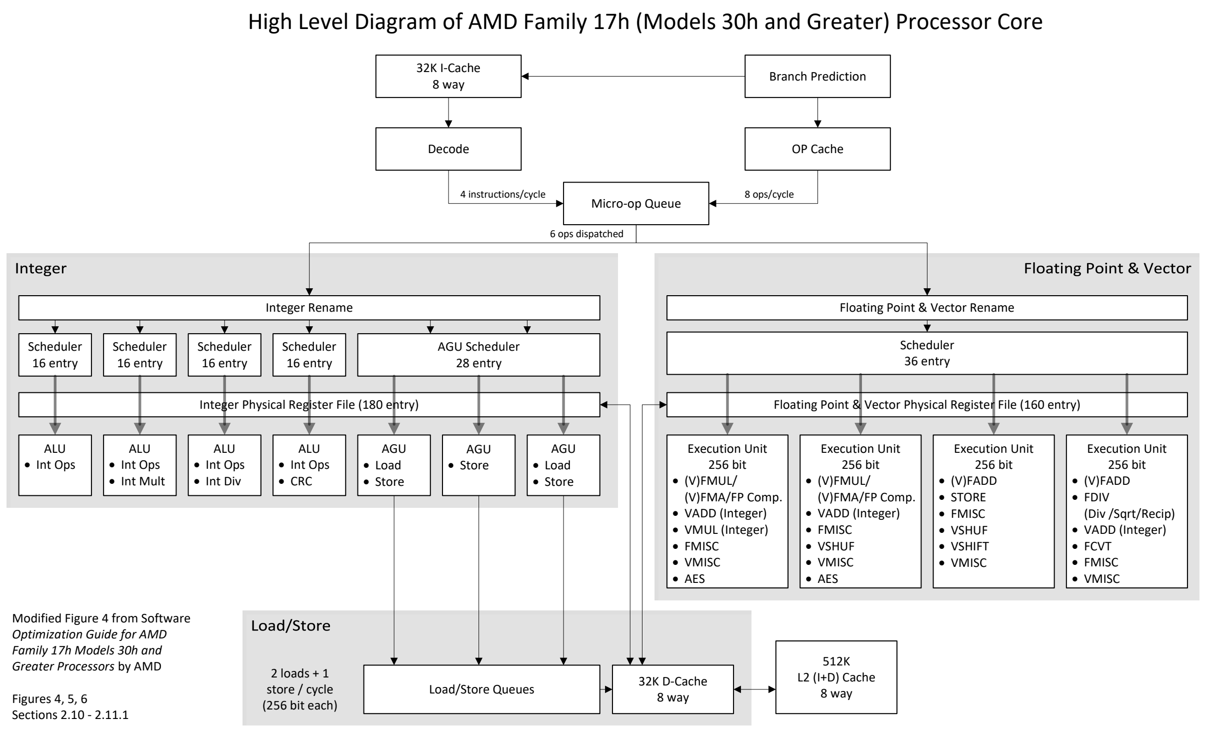

High Level Diagram of AMD Family 17h (Models 30h and Greater) Processor Core
32K I-Cache 8 way
Branch Prediction
Decode
OP Cache
4 instructions/cycle
8 ops/cycle
Micro-op Queue
6 ops dispatched
Integer
Floating Point & Vector
Integer Rename
Scheduler 16 entry
AGU Scheduler 28 entry
Integer Physical Register File (180 entry)
ALU • Int Ops
ALU • Int Ops • Int Mult
ALU • Int Ops • Int Div
ALU • Int Ops • CRC
AGU • Load • Store
AGU • Store
AGU • Load • Store
Floating Point & Vector Rename
Scheduler 36 entry
Floating Point & Vector Physical Register File (160 entry)
Execution Unit 256 bit • (V)FMUL/(V)FMA/FP Comp. • VADD (Integer) • VMUL (Integer) • FMISC • VMISC • AES
Execution Unit 256 bit • (V)FMUL/(V)FMA/FP Comp. • VADD (Integer) • FMISC • VSHUF • VMISC • AES
Execution Unit 256 bit • (V)FADD • STORE • FMISC • VSHUF • VSHIFT • VMISC
Execution Unit 256 bit • (V)FADD • FDIV (Div /Sqrt/Recip) • VADD (Integer) • FCVT • FMISC • VMISC
Modified Figure 4 from Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors by AMD
Figures 4, 5, 6
Sections 2.10 - 2.11.1
Load/Store
2 loads + 1 store / cycle (256 bit each)
Load/Store Queues
32K D-Cache 8 way
512K L2 (I+D) Cache 8 way

Figure 3.2: AMD Zen2 CPU Core - Modified Diagram from [17]

3.1.4 Pipelining and Branch Prediction

Modern CPUs, including in-order CPUs, leverage pipelining to increase throughput. Instead of an instruction being fetched, executed, and completed within a single clock cycle, the different phases of execution are split via pipeline registers. By reducing the critical path through the CPU, pipelining typically enables faster clock rates at the expense of instructions now requiring multiple cycles to finish. In the absence of conflicts, a new instruction can enter and another can exit the pipeline each cycle. By allowing multiple instructions to be in the pipeline at once (each at a different phase of execution), the throughput of the CPU generally increases while the latency for an individual instruction typically suffers. The execution units of superscalar out-of-order CPUs are often pipelined, allowing new instructions to be issued to the execution unit before the last instruction had finished. For example, the AMD Zen2 leverages pipelining with the throughputs of various instructions detailed in [17]. For an example of pipelining in an in-order processor, see [105].

As useful as pipelining and out-of-order execution are for improving performance, it does come at a cost when branch instructions are encountered. When a branch is encountered, such as an `if ... else`, the next instruction to execute is dependent on the result of the branch condition. This dependency interferes with both pipelining and out-of-order execution which provide improvements by beginning the execution of instructions before previous ones have completed. A sub-optimal strategy would be for the pipeline to stall or out-of-order execution to avoid scheduling new instructions until after the branch direction has been determined. This effectively gives up the advantage provided by pipelining and out-of-order execution in the regions surrounding branches and can be expensive, especially if extensive pipelining is used in the CPU design. A popular alternative approach is to *predict* the direction of a given branch and to speculatively execute the instructions along the predicted path. If predicted correctly, this keeps the pipelines full and execution units busy. If predicted incorrectly, the CPU needs to revert the results of any of the speculated execution and execute the correct branch. The delay in performing this is referred to as the *misprediction penalty* and can be very costly. The goal with most branch predictors is that, after enough executions of the branch, a pattern can be identified which results in overall performance improvements. For more information on branch prediction and how it manifests in both in-order and out-of-order CPUs, see [16], [105].

Pipelining and branch prediction can both place constraints on high performance code. In the presence of aggressively pipelined execution units, the number of independent instructions required to keep execution units busy increases. More independent instructions are required to compensate for the additional number of cycles operations spend being executed, preventing dependent operations from running for more cycles. The types of control decisions present in the program can also have a strong impact on the performance of the branch predictor. Loops which follow a predictable, fixed, pattern can often be predicted correctly with modern techniques. However, unpredictable branching or branching with complex patterns can interfere with the prediction logic, resulting in excessive branch misprediction penalties.

3.1.5 Simultaneous Multi-Threading (SMT)

As mentioned above, leveraging the execution resources available in a CPU core requires leveraging instruction level parallelism (ILP). On a general-purpose system, it is possible that a single program may not be able to provide sufficient ILP, may not have the correct balance of different instruction types, or may be bottle-necked on memory or I/O. Simultaneous Multi-Threading (SMP, sometimes called hyper-threading in Intel contexts) aims to improve efficiency by filling the execution gaps in one program with work for another program. This multiplexing occurs at the hardware level and is presented to the OS/user as a given CPU core having multiple (often two) virtual cores. Assuming the programs are independent this should allow the CPU core to perform more total work in a given unit of time. The OS is free to schedule work on these virtual CPUs and the core will multiplex between these threads. The core maintains separate architected state for each virtual CPU and can typically switch between executing instructions from these two virtual CPUs at relatively low cost. For details on the AMD Zen2 implementation of SMT, see [9].

SMT is most helpful when a single system is executing multiple programs, each of which does not have the required workload to keep the core busy. It can have downsides, however, as both virtual CPUs share some resources such as the cache, potentially causing contention. In the context of this project, attempts will be made to maximize the amount of ILP available to each core for the given signal processing application. SMP will generally be disabled to provide tighter control over the execution and more consistent/predictable results.

3.2 Modern x86_64 Cache, Memory, and Core-Core Communication

At the time of writing, DRAM performance has significantly lagged CMOS logic performance [16]. It can take many clock cycles to fetch data from DRAM or write results back into DRAM. To reduce the impact of this performance mismatch on overall system performance, most modern high-performance processors employ extensive caching. Caching is arranged in levels with the lowest level cache (L1) providing the fastest access and the higher-level caches providing more capacity but less performance. Caches operate on *lines* of data and the size of cache lines can vary between platforms. On AMD Zen2, the cache line size is 64 bytes [17]. When data is requested, the cache line to which the data belongs is fetched from the lowest level data cache (L1D). If the data is not resident there, the next cache level is queried with the result copied into the lower-level cache and returned to the core. If the data is not present in any cache level, a request is made to DRAM. Caches can be either inclusive (having a copy of all the data in caches below it) or exclusive. Zen2's L2 cache is inclusive while its L3 cache is exclusive except for data shared between cores [17].

A subsection of the cache hierarchy used in the Ryzen Threadripper 3970X CPU is conceptually diagrammed in Figure 3.3. This CPU uses a multi-die-in-package approach with 4 separate CCD compute dies on the same package substrate connected by an IOD I/O die. Each core complex (CCX) of four cores shares a single L3 cache with each core having private L2, L1 Data (L1D), and L1 Instruction (L1I) caches.

The importance of the memory subsystem of a modern CPU is often underestimated when high-performance is needed. The memory subsystem is not only responsible for hosting programs themselves and the state they rely on, but also often the only feasible high-performance mechanism to communicate across cores on the same platform. Careful consideration of the memory subsystem and interconnect has been a hallmark of High-Performance Computing (HPC) for a long time and is a key focus of HPC courses like CS267 at Berkeley [108]. Details of the memory subsystem's characteristics in the context of multiple cores is discussed below.

3.2.1 Muti-Core Cache Coherence, MESI, and Memory Consistency

When a CPU has multiple cores, there are choices on how to handle memory and caching. One possibility would be to partition memory across the cores and for each to have its

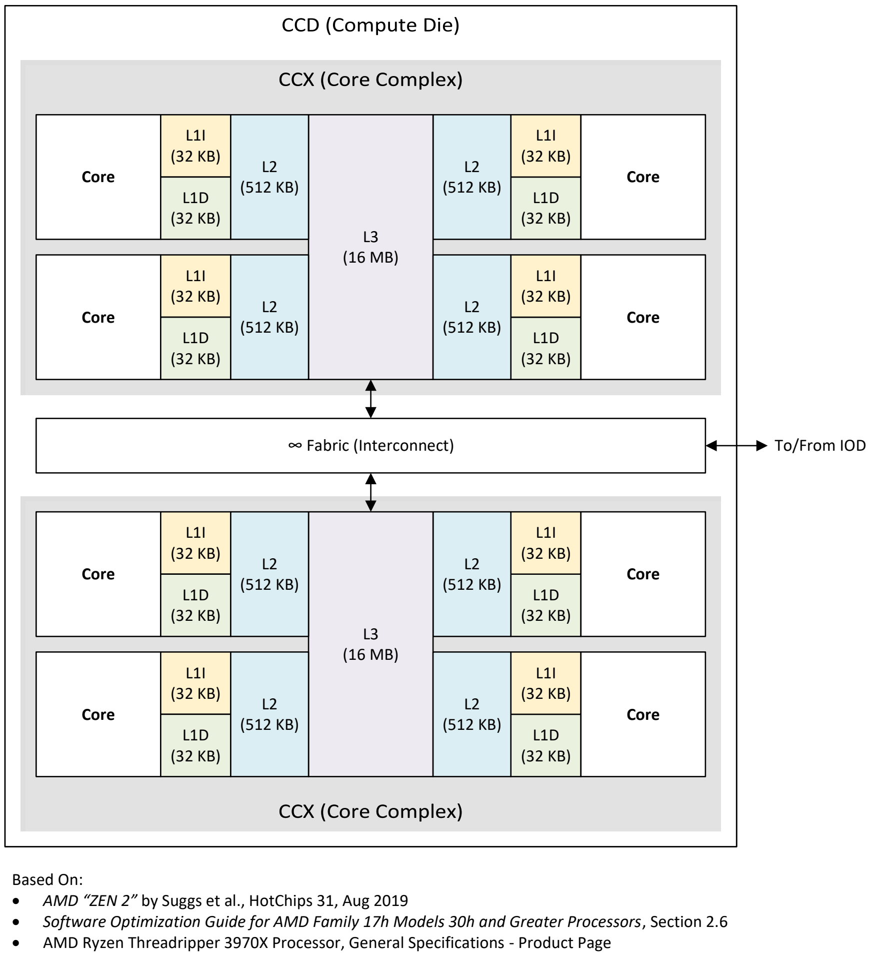

CCD (Compute Die)
CCX (Core Complex)
Core
L1I (32 KB)
L1D (32 KB)
L2 (512 KB)
L3 (16 MB)
∞ Fabric (Interconnect)
To/From IOD
Based On:
AMD "ZEN 2" by Suggs et al., HotChips 31, Aug 2019
Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors, Section 2.6
AMD Ryzen Threadripper 3970X Processor, General Specifications - Product Page

Figure 3.3: Ryzen Threadripper 3970X - Cache, CCX and CCD [17], [106], [107]

own cache. However, this would complicate the programming model for developers and would require explicit mechanisms to communicate information among cores. An alternative approach is for all the cores to share access to memory and for the different caches in the system to maintain coherence. Coherence, in this context, specifies requirements on what values can be returned for a read to a given cache line by different caches within the system [16]. Conditions for coherence, as described by [16], include:

- If a given core writes to a memory location, and no other core writes to the location, subsequent reads by that core should return the new value.

- When a cache line is written by one core, subsequent reads by other cores' caches should return the new value subject to enough time passing since the write.

- In the case that multiple cores write to the same cache line, the writes appear in the same order to all cores on the system (i.e., the writes are serialized).

Maintaining the coherent view of memory becomes the responsibility of the cache coherency protocol. Cache coherency is a complex topic with many different alternative implementations. In-depth information on cache coherency is available in computer architecture texts such as [16]. While CPUs often use more complex cache coherency protocols (such as AMD Zen1 Zeppelin, which uses a 7 state MDOEFSI cache coherency protocol [109]), it will generally be sufficient in this document to view the cache coherency protocol as *MESI like*. MESI is an acronym for the different states for cache lines in this protocol:

- Modified (M): The cache line has been modified in this local cache. The entry for this cache line in all other caches should be invalid.
- Exclusive (E): This cache line is exclusively held by this core. Modifications can be made by this core. The entry for this cache line in all other cores should be invalid.
- Shared (S): This cache line has copies in multiple cores. No modification can be made without first obtaining exclusive access.
- Invalid (I): This cache line is not resident in this core's cache.

A coherent view of memory is maintained by forcing cores to obtain exclusive access to the cache line before modifying the data. See [16] for more information on MESI and other cache coherency protocols.

By reading and writing the same memory locations, different cores in a multi-core CPU can communicate via the cache coherency mechanism. The transportation of data from core-to-core occurs via the cache coherency protocol. For example, assume that core A wants to communicate some data to core B. Communication can occur in the following fashion:

1. A portion of the shared memory space is allocated with both cores A and B knowing the address.
2. Core A writes data to the shared memory space. This causes it to request exclusive access to the cache lines if it does not already have it. This request invalidates any copy of requested cache lines in other cores' caches. After the write, core A's cache holds the shared cache lines in the *modified* state.
3. Core B attempts to read the shared data. Its local cache should have the shared cache lines in the *invalid* state either because it was not resident in the local cache or was invalidated when core A acquired exclusive access to the cache lines. Core B's cache issues a request for the shared memory addresses causing the modified cache lines from

core A to be written back to memory[4] and shared with core B. Cores A and B will then have the shared cache lines in the *shared* state in each of their private caches.

3.2.2 Memory Consistency Models

The cache coherency protocol is not the only thing to consider when communicating between cores. While cache coherency should prevent writes by different cores creating an incoherent view of memory, it does not guarantee a global ordering of how different memory actions are viewed across the entire CPU. The order in which different memory operations appear to the system is referred to as the *memory consistency model*. As with many aspects of CPUs, different consistency models exist, and different CPU ISAs make different decisions on which one to adopt. x86_64 typically provides a relatively strong memory consistency model [103], [104], [110]. It is not as strong as full sequential consistency, which specifies that memory operations among different threads appear in order with some interleaving between the different threads, but it is one of the stronger memory consistency models used in modern CPUs [110]. The x86_64 memory consistency model is described as *Total Store Ordering (TSO)* [103], [110], and preserves order among writes but does not guarantee when the write will be perceived by other cores. Details of the x86_64 memory consistency model are available in the Intel and AMD architecture manuals [103], [104]. TSO is also discussed as a relaxation to sequential consistency in [16].

Cache coherency and the memory consistency models carry exceptionally high importance for modern multi-core systems. Outside of interrupts, cache coherency is often the only feasible way to communicate between cores. Due to the complexity of the memory and coherence models and the divergent decisions taken by different CPU ISAs, it can be difficult to create portable solutions that work across different ISAs. In this project, x86_64's specific memory consistency model was used to create lockless single producer/consumer FIFOs between cores. This may not be possible on all ISAs, particularly ones using weaker memory consistency models. This project uses the C11 `atomic` constructs to indicate to the compiler the necessary relationship between different memory operations. For x86_64, the compiler is able to avoid the use of locks and explicit fences, but this may not be true for other ISAs. The implementation of the FIFOs will be discussed in section 7.1.

3.2.3 Inter-die Interconnect Effects and Non-Uniform Memory Access (NUMA)

As the number of cores in modern CPUs have increased, the cache and interconnect have become more complex to provide the desired I/O and memory subsystem performance.

[4]An extension of the MESI cache coherency protocol utilizes another state, Owned (O), to allow the delayed writeback of the modified lines to memory. AMD's x86_64 CPUs reportedly use MOESI cache coherency protocols [104]. The statement in [109] that the AMD Zepplin CPU uses a 7 state MDOEFSI protocol suggests that the hardware implementations may utilize additional states. In general, it should be sufficient to view the AMD cache coherency protocol as MOESI from the programmer's perspective.

Multi-socket systems[5], which have existed in the server market for some time, further complicating interconnect requirements by allowing multiple CPUs to exist in the same, cache coherent, domain. A common implementation of these multi-socket systems was to connect DRAM to each CPU in the system. Access from a CPU to the memory directly connected to it was typically fast while access to memory connected to another CPU in the system imposed a significant performance penalty. Architectures that provide different performance when accessing memory depending on the location of the requesting CPU and the requested memory are referred to as providing *Non-Uniform Memory Access (NUMA)*. The boundaries of fast memory access for a given set of cores defines what is typically referred to as a *NUMA Domain*. Once a relatively niche issue for HPC developers, NUMA considerations have become more important with the introduction of new many-core CPUs. For more information on NUMA, see [16].

AMD has been a major proponent of multi-die-in-package technology to provide many core CPUs at reasonable costs [98]. This approach spreads the components of the CPU across multiple silicon dies which are connected via advanced packaging or silicon interposers. In general, any time a signal goes “off-die” or “off-package” it incurs a latency and power efficiency penalty. The first generation Epyc (and Threadripper 2990WX) CPUs utilized multiple dies which contained both cores and local memory controllers[6], creating noticeable NUMA effects [98]. The second generation Epyc and (Threadripper 3970X) take a different approach by using two different die types [98]:

- Compute Dies (CCDs) which contain CPU cores and caches.
- A single I/O Die (IOD) which contains the memory controllers, I/O controllers, system level controllers, and an interconnect.

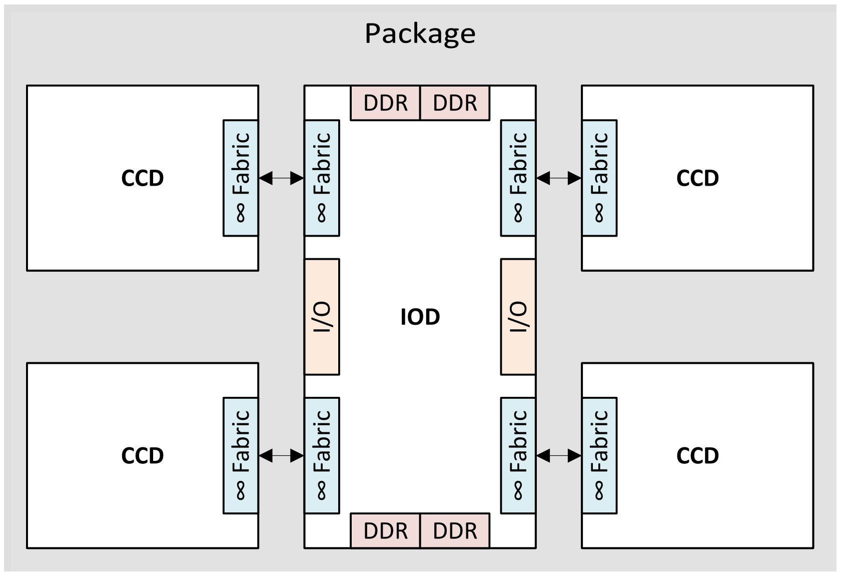
A conceptual diagram of the die layout of the Ryzen Threadripper 3970X is shown in Figure 3.4[7]. Requests not serviced by a local L3 are sent over the interconnect to the IOD[8][98]. Because each CCD needs to access the IOD to access memory, the NUMA effects are reduced although not eliminated. The server IOD design, depicted in Figure 3.5, does not maintain constant latency between CCD connections and DDR controllers [98]. While [98] describes

[5]Multi-socket systems have more than one CPU residing on a single motherboard.

[6]Only two of the dies on the Ryzen Threadripper 2990WX have DDR memory controllers and are directly connected to memory. The other cores must communicate with neighboring dies via the interconnect to access memory [111].

[7]This diagram assumes the Ryzen Threadripper 3970X IOD is similar to the 2nd Generation “Rome” Epyc IOD with the number of DRAM channels reduced from 8 to 4 and lack of multi-socket capabilities. [98] notes that the chiplet design from the 2nd Generation Epyc was later used for 64 core HEDT products, ie. Threadripper. Figure 23 also shows the 3rd Generation Threadripper using a similar die layout to the 2nd Generation “Rome” Epyc server products. This is partially corroborated by technology media coverage of the launch of the Ryzen Threadripper 3970X[113] noting its use of the 2nd Generation Epyc server parts along with accompanying slides showing similar die sizing and layout to 2nd Generation Epyc.

[8]Testing performed as part of this project did not show a significant performance advantage going between L3s on the same CCD vs L3s on different CCDs.



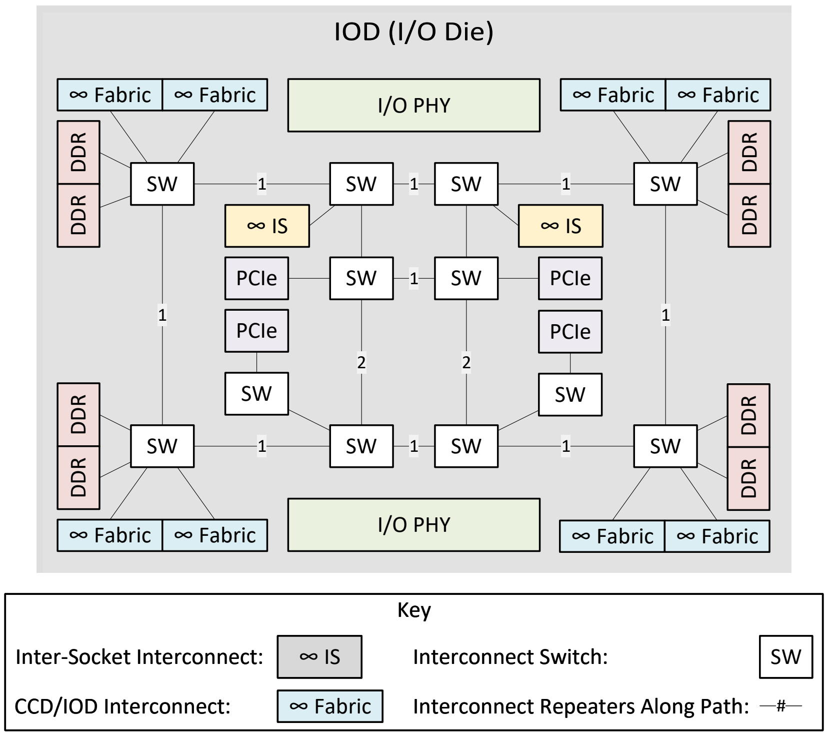
Simplified Package Design Diagram from:

- *Pioneering Chiplet Technology and Design for the AMD EPYC and Ryzen Processor Families: Industrial Product* by Naffziger et al., ISCA 2021
- *AMD "ZEN 2"* by Suggs et al., HotChips 31, Aug 2019
- *AMD Chiplet Architecture for High-Performance Server and Desktop Products* by Naffziger et al., ISSCC 2020

*** See Footnote for Assumptions**

Figure 3.4: Ryzen Threadripper 3970X - Die Layout [98], [106], [112]

the server IOD, it is suspected that the consumer Threadripper 3970X's IOD is similar with a reduction of memory channels from eight to four and the removal or deactivation of socket-to-socket interconnect points.



Simplified Figure 13 from *Pioneering Chiplet Technology and Design for the AMD EPYC and Ryzen Processor Families: Industrial Product* by Naffziger et al., ISCA 2021

Figure 3.5: 2nd Generation Epyc IOD - Simplified Diagram from [98]

Chapter 4

Modes of Parallelism: Radio Signal Processing

Chapter 3 primarily focused on the design of modern CPUs and the different modes of parallelism available. This is only half of the equation as these parallel resources are of little use unless the workload can leverage them. As was noted in section 3.1, the different modes of parallelism available on the CPU perform best when the program being run has specific characteristics. In the context of this project, the program is the radio signal processing design. Fortunately, radio designs typically present multiple forms of parallelism. While there are ways to extract additional parallelism out of the design, the following are examples of parallelism which come directly from the radio signal processing domain. More details about introducing additional parallelism into the Cyclops baseband will be discussed in chapter 8.

4.1 Parallel Operators

It is not uncommon for DSP designs to conduct multiple computations in parallel. This can take several forms and includes separate computational paths in feed-forward segments of the design. For example, a single value may be fanned out to two different computational paths which have no dependency besides the common source value. It is also possible that two different inputs have independent computations performed on them before their results are ultimately used in a common operation. In the example shown in Figure 4.1, Op1 and Op2 can execute in parallel after Op0 has finished executing since they are not dependent on each other. Op3 cannot execute, however, until both Op1 and Op2 have completed.

In hardware, pipelining is one technique to introduce parallelism by breaking a long chain of combinational logic into separate chains which can execute in parallel separated by a pipeline register. In hardware, pipelining along the critical path allows the clock rate to be increased which, combined with work being conducted in parallel in each pipeline stage, results in improved throughput. Pipelining can have a similar effect in software by breaking

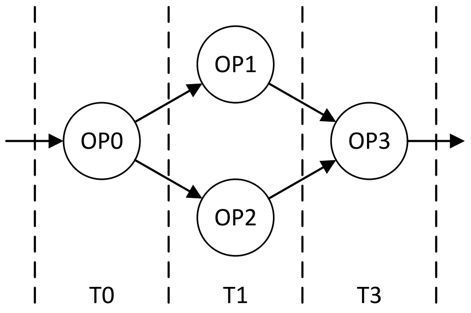


Figure 4.1: Example of Parallel Combinational Paths

long operator dependency chains. In this context, it is referred to as *software pipelining*[1] and introduces ILP across the two segments broken by the pipelining. Techniques such as retiming can be used to shift delays in the design to try to produce pipeline stages with roughly equal amounts of work. This general technique has an impact on both hardware and software implementations. However, there are other constraints in software, particularly surrounding core-core communication, which often have a greater need for the delays.

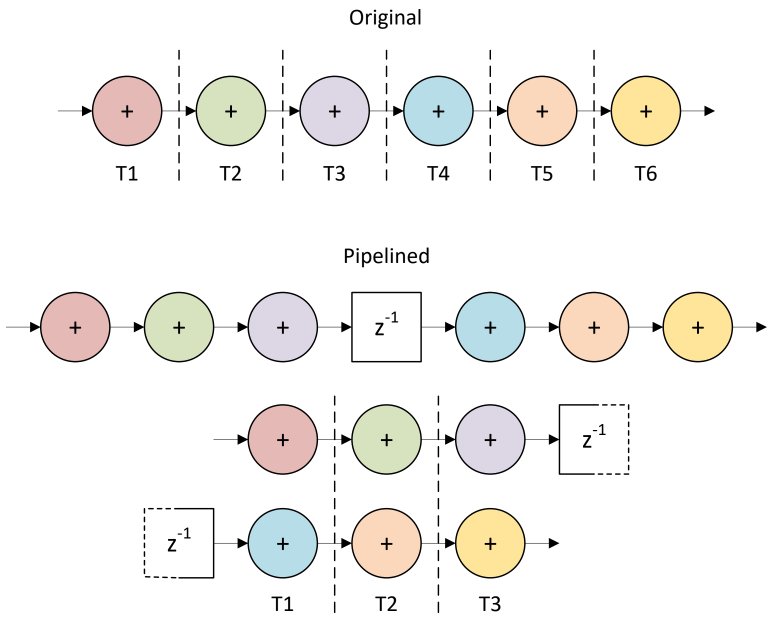


Figure 4.2: Example Software Pipelining ILP Improvement (T = Operator Execution Time)

[1]For more information on software pipelining, see Appendix H of [16].

4.2 Vectorizable Structures

4.2.1 FIR Filters

Several widely used DSP and communications constructs lend themselves to vectorization which exploits the SIMD parallelism present on the CPU. FIR filters are perhaps the prototypical example as they are composed of a *tapped delay* and a *dot product*. The dot product, being a vector operation itself, is vectorizable in modern CPUs and can even take advantage of specialized Fused Multiply Accumulate (FMA) units. There are complications with the tapped delay, but these will be discussed in subsection 6.2.3.

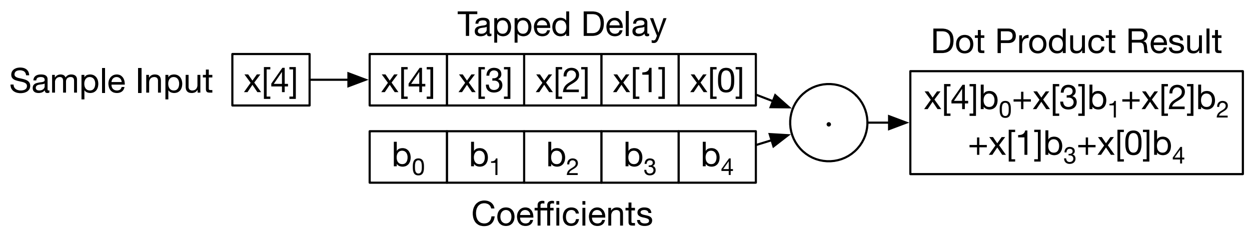


Figure 4.3: FIR Filter Described with Dot Product

4.2.2 Forward Error Correction (FEC) Decoders

Despite our best efforts, radios typically experience some errors when decoding a packet. Forward Error Correction (FEC) is a technique to reduce the number of errors by including redundant information in the transmission. There are many different FEC techniques including block codes and convolutional codes, each with its own set of advantages and disadvantages. To be most effective, FEC codes often create complex dependencies between a series of transmitted bits, which tends to make efficient implementations challenging.

While FEC is one of the more challenging aspects of a typical radio system to implement efficiently in software, these blocks too present opportunities for parallelization. In the course of this project, the potential for parallelizing the decoding of convolutional codes was investigated. Specifically, the Viterbi decoder (originally described by Viterbi in [114] and later made more accessible in [115]) was investigated. This popular decoder provides the maximum likelihood decoding of the received signal[2], a property which was shown in the often cited tutorial paper by G.D. Forney [117]. A brief introduction to convolutional encoding and the Viterbi algorithm is given in section A.3.

[2]The Viterbi algorithm is only the maximum likelihood decoding scheme when the traceback includes the entire message. For practical implementations, traceback is typically limited to several times the constraint length. A rule of thumb mentioned in [116] is that a traceback length longer than five times the constraint length usually results in negligible performance loss.

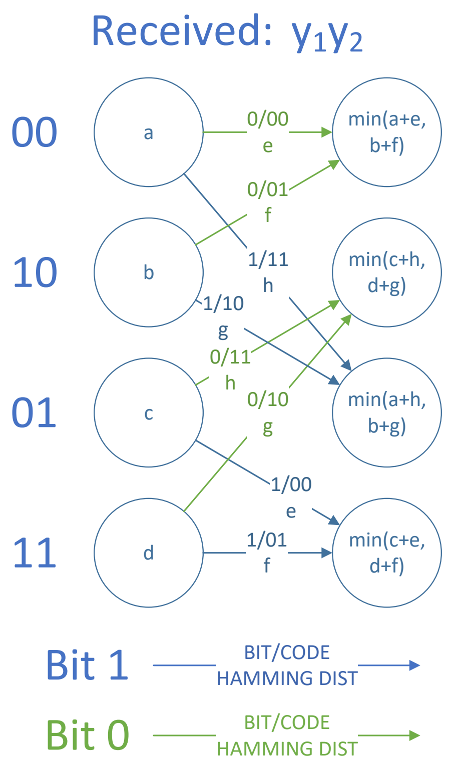
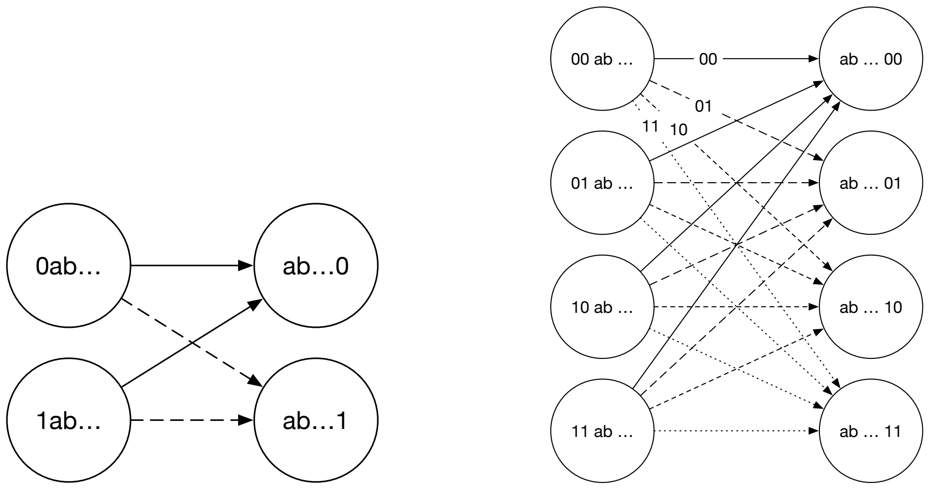


Figure 4.4: Example Viterbi Decoder Trellis (1 Iteration) for k=1, Generators 0b111, 0b110

Due to its usefulness and popularity in earlier communications systems, the Viterbi decoder has been extensively researched. This research includes methods to accelerate decoding through various techniques including re-framing the algorithm to be more friendly for implementation. As originally posed in its trellis form, optimizing the Viterbi algorithm appears possible but unfriendly. While the data dependencies are static, there can be large strides between elements for which the dependency exists. Hamming distances for a given round either need to be re-computed for paths or pre-computed but used with an irregular dereference during computation.

Several different works including [117]–[119] note that the Viterbi trellis bears a lot of similarities to the Fast Fourier Transform (FFT) structure, another extensively researched algorithm. While the Viterbi algorithm cannot utilize partial results like the FFT can, its trellis can be decomposed into regular structures. These structures are *butterflies* and *perfect shuffle networks* as described in [119]. This particular decomposition provides multiple benefits from a vectorization standpoint. First, the computation can be more easily vectorized across the different butterflies in the network as they share the same structure and operand placement. Second, the perfect shuffle network is an interleaver. This is important as vector



(a) k=1 (b) k=2

Figure 4.5: Viterbi Butterfly Structures

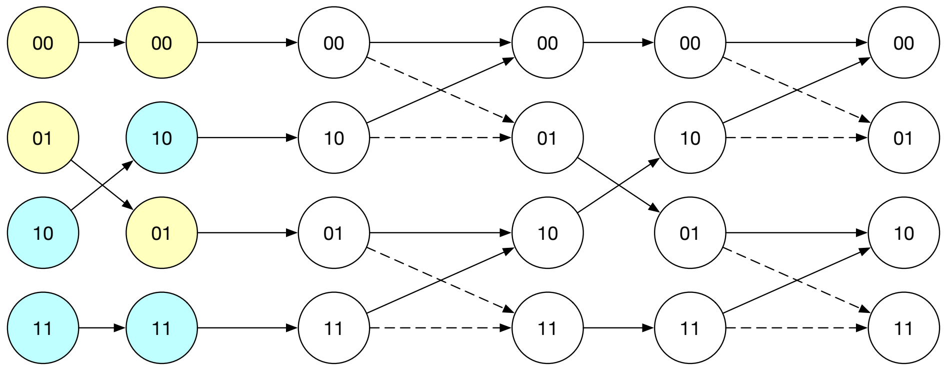
ISA extensions tend to include instructions to support interleaving/de-interleaving[3]. The net result is that, just by re-structuring the Viterbi trellis, the Viterbi decoder can be made much more vector friendly. An example of the butterfly networks for $k = 1$ and $k = 2$ are shown in Figure 4.5[4]. Examples of decomposed Viterbi decoders with $k = 1$ and different constraint lengths[5] (K) are shown in Figure 4.6. Specifically, Figure 4.6a is the refactoring of the Viterbi trellis shown in its traditional form in Figure 4.4. By changing how the shuffle network is incorporated, it is also possible to produce interleaved butterflies. This re-organization can potentially assist in vectorizing across the butterflies in the forward pass of the Viterbi algorithm by organizing state such that vectors contain a given input to n different butterflies. Operations on that single input can then be vectorized across the different butterflies natively without re-organization. An example of Figure 4.6a's interleaved structure is shown in Figure 4.7. One downside is that is requires additional operations to compute node addresses in the traceback portion of the Viterbi algorithm.

A further optimization trick comes from exploiting a property in some common generator polynomials used by convectional codes. As noted in [120], for rate 1/n codes, certain generator polynomials result in the edges going into a node in the trellis having complementary coded bits. In a butterfly, all four edges experience this complementary symmetric property, as shown in Figure 4.8. The net result when using this subset of generators is that only

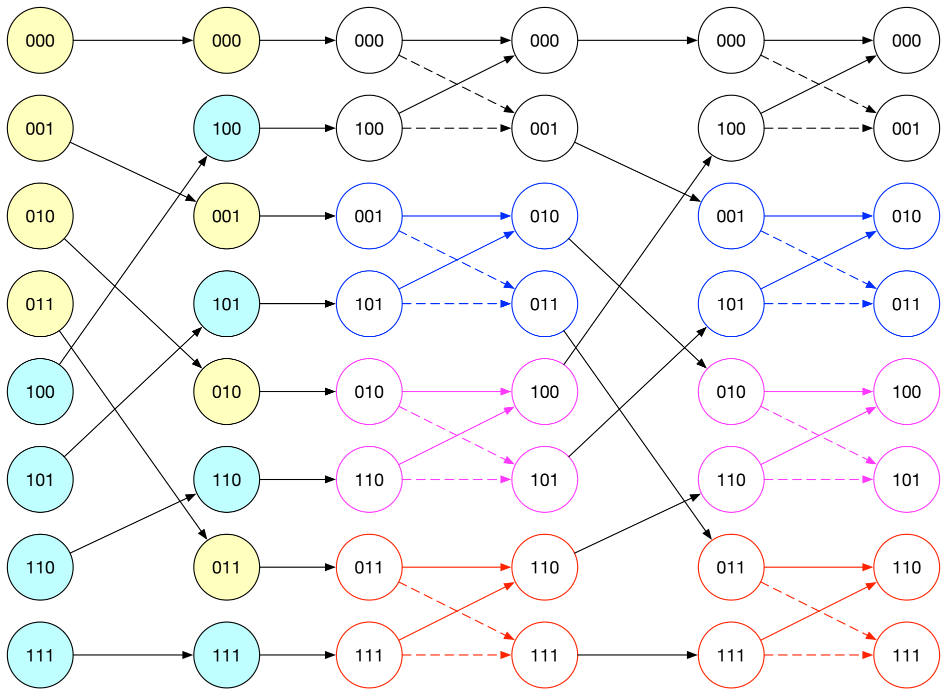
[3]For example, The AVX and AVX2 x86 vector extensions provide a variety of instructions which perform interleaving including `VPUNPCKLBW` and `VPUNPCKHBW` which interleave bytes from two vector registers [103]. There are additional interleaving vector instructions for different word widths.

[4]k is a parameter for convolutional codes and is the number of bits shifted into the convolutional encoder at a time.

[5]The constraint length (K) is a parameter of convolutional codes and denotes the length of the shift register.



(a) K=3



(b) K=4

Figure 4.6: Viterbi Decomposed into Butterfly and Shuffle Networks (k=1)

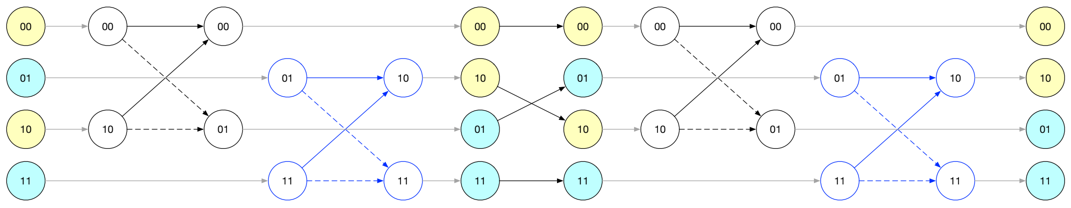


Figure 4.7: Viterbi Decoder Trellis (1 Iteration) for k=1, K=3, Butterflies Interleaved

a single Hamming distance needs to be calculated per butterfly in the forward pass of the Viterbi algorithm.

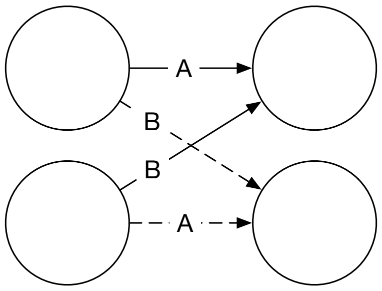


Figure 4.8: Viterbi Decoder Trellis (k=1) when Polynomial Symmetry Present

Additional implementation tricks include re-normalizing the accumulated path metrics along the forward pass [119] to reduce the required size of the accumulator. The frequency of this re-normalization depends on the characteristics of the given convolutional code.

To see what the C compiler could accomplish without vector intrinsics or inline assembly, a version of a rate 1/2 Viterbi decoder was written in C [121]. This version was compared to several implementations of the Viterbi algorithm including multiple derivatives [122], [123] of a decoder originally authored by Phil Karn (KA9Q) [124]. Much of the work in the KA9Q derivatives appears to be geared towards supporting modern x86_64 CPUs. This may have caused some performance degradation as the achieved performance is lower than what was observed in the benchmarking results presented in [125]. Some of this performance degradation could also be due to differences between the CPU platform used in [125] and the 2.0 GHz CPU platform used in the tests presented in Table 4.1

This project's implementation, before restructuring or optimization, produced a result that was close to one of the KA9Q forks but was slower than others including the Spiral portable version. Notably, none of the KA9Q derivatives nor the Spiral portable version were able to achieve double digit Mbps processing rates on this platform. After re-organizing the algorithm to use butterflies and the perfect shuffle networks and introducing polynomial symmetry and re-normalization optimizations, this project's C implementation was able to achieve a processing rate of $\approx$39 Mbps. This is a notable improvement over the other portable C implementations tested and showcases what the C compiler is capable of when presented with a carefully considered implementation. The C compiler was able to properly vectorize butterflies and interleave. There is still room for improvement, however, as it is still slower than the specialized Spiral implementation which was able to achieve a 66.2539 Mbps processing rate on the same platform. One notable difference is that the specialized Spiral implementation makes extensive use of x86_64 SSE vector intrinsics and therefore would need to be updated to support other vector extensions. Particularly relevant to this discussion is that additional vector extensions for x86_64 have been introduced and widely adopted since this generator was written. While modern processors can still execute SSE (128-bit wide vector) instructions, they now generally support AVX/AVX2 which uses 256-bit wide vectors. Some high-end Intel processors even support AVX-512 which extends vector lengths to 512-bits.

There is potential for both the performance of this work as well as the Spiral implementation to improve with some modifications. Traceback is one component of this project's implementation which could use further investigation. In particular, reversing the bit endianness of state IDs could potentially avoid a bit reversal operation in the traceback step. The Spiral implementation could potentially be improved by utilizing AVX/AVX2 instructions which support double the vector width of SSE. For recent research on software implementations of FEC decoding that was recently brought to the author's attention, see [70].

4.2.3 Common Operations Across Instances

The previous sections have focused on vectorization opportunities within single blocks in a baseband. An alternative approach is to vectorize across instances of a given operation if multiple instances exist. This can occur at both fine and coarse levels of granularity. For example, one technique to accelerate FEC would be to interleave different convolutional codes and to feed the lower rate de-interleaved results to convolutional coders operating in parallel. Common operations across the different FEC decoders could potentially be vectorized. Similarly, in OFDM type radios, common operations that occur post FFT, such as equalization, can potentially be vectored across sub-carriers.

The concept can extend to an even coarser domain when multiple radio baseband instances are running simultaneously on the same platform. This could occur for several reasons, including in cellular deployments with multiple tower sectors being processed on a common system or radios on separate frequency bands being processed in the same system. Assuming that the instances of the basebands are identical apart from the input stream, some common operations across the different instances of the baseband could be combined and vectorized. This would most likely work best with the simpler datapath portions of the baseband rather than the control portions or conditionally executed portions as control decisions present an obstacle to effective vectorization. While vectorizing across multiple basebands is a valid approach, this project has primarily focused on vectorizing within single instances of the baseband. Techniques, such as sub-blocking, are used to facilitate additional vectorization within a single instance of a design.

4.3 Opportunities for Increased Independence with Protocol Co-design

As alluded to above, decisions made in the specification of the radio PHY and MAC can affect the amount of parallelism that is easily extracted. Blocks such as FEC, which have been historically challenging to implement in software, could be accelerated by creating

Library	Benchmark Speed (Mbps)			Benchmark Name	Benchmark Config
	gcc 7.5	gcc 10.2.0	Vendor Compiler		
ka9q-fec (KA9Q Fork) [122]	5.74127	5.86174	6.29758	vtest27 (from automake)	10000 2048-bit frames
libfec (KA9Q Fork) [123]	2.43474	2.43815	2.31306	vtest27 (from cmake)	10000 2048-bit frames
libfec (KA9Q Fork) [123] - Modified Cmake to turn on optimization (Ofast)	5.7601	5.75628	6.30708	vtest27 (from cmake)	10000 2048-bit frames
This Work (before butterfly / shuffle network restructure)		≈2.5	≈2.8	speedDecode	2048-bit frames
Spiral Portable C [125] (makefile with Ofast, validation change for clang)	7.46892	7.26308	8.30595	viterbi	10000 2054-bit frames
Spiral **SSE2 16 Way - Intrinsics** [125] (makefile with Ofast, validation change for clang)	61.1791	63.086	**66.2539**	viterbi	10000 2054-bit frames
This Work (non-interleaved butterfly, polynomial symmetry, and re-normalization, **no intrinsics**)		≈31	≈**39**	speedDecode	2048-bit frames

Table 4.1: Rate 1/2, K=7 (Voyager) Viterbi Implementation Performance On 2 GHz CPU Supporting 256-Bit Wide Vectors (AVX2)

multiple independent or lightly dependent FEC blocks which operate in parallel. This would require a change to the PHY spec to support interleaved FEC.

Another challenge with any software radio system is expanding the processing rate of a single instance to support very wide bandwidths. For example, 802.11ad channels are wider than 2GHz [85], [86]. To accomplish real-time processing on a modern 2 GHz server, more than one complex sample would need to finish processing every CPU cycle on average. Vectorization, superscalar execution, and multiple cores appear to make this theoretically possible. However, difficult to vectorize constructs such as finite state machines (FSMs) as well as instruction latencies within the CPU combined with communication challenges within the memory subsystem make achieving these bandwidths with a single instance impractical, if not impossible. Even with an OFDM based system where parallelization across subcarriers is feasible, some operations exist pre-FFT and would need to run at the data converter rates. The FFT and IFFT would also need to be exceptionally high performance, consuming and producing samples at the required fast rate. One method to mitigate this would be to use a channelized approach where the wide bandwidth was split into multiple smaller channels. Each of these channels could be handled independently by a different instance of the baseband with data interleaved across the different channels.

Splitting a larger frequency band into smaller isolated sub-bands is not a foreign concept to communications systems. Notably, the Filtered Multitone (FMT) multi-carrier modulation technique was proposed for Very High-Speed Digital Subscriber Lines (VDSL) [126]. This technique uses a channelizer to produce spectrally separate channels over which separate information can be sent. A conceptual view of the spectrum is shown in Figure 4.9. One consideration for any multi-channel system is the degree of separation between adjacent channels. The wider the guard bands between channels, the less spectrally efficient the overall system. However, the narrower the guard bands, the more crosstalk exists between channels[6], distorting the signals in each band. It is important to note that, even though FMT style channelization introduces spectral inefficiency due to the use of guards between subcarriers, alternative methods which use "cyclic extensions" also introduce inefficiency [126]. Although OFDM leverages orthogonality between subcarriers, its use of the cyclic prefix introduces an inefficiency as a subset of samples is repeated.

This approach was explored in an earlier stage of this project and was demonstrated at the Berkeley Wireless Research Center (BWRC) Spring 2020 Research Retreat. In this demonstration, four different instances of the baseband were used. The Tx side used a single compute core for each channel while the Rx side used multiple cores per channel. Channel combining at the transmit side and splitting of the channels at the receive side were performed by polyphase channelizers. The channelizers performed the filtering operations as well as the mixing operations to relocate channels between their selected carrier frequencies

[6]While crosstalk can theoretically be mitigated in systems with overlapping channels so long as the channelizer design fulfills the *perfect reconstruction* criteria, the orthogonality between channels is effectively lost in real world communication channels [126]. As noted in [126], the solution in the Discrete Multitone (DMT) multi-carrier modulation schemes with significant overlap between subchannels is to use a cyclic extension. This is a very similar mechanism to the cyclic prefix used in OFDM based systems.

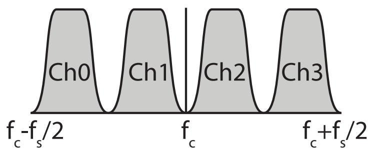


Figure 4.9: Example of Multi-Channel Operation in the Frequency Domain

and DC (baseband). In this demo, the channelizers were implemented in software with a single compute core allocated for each. Payload data was interleaved across the different channels on the Tx side and reassembled at the Rx side to provide a faster aggregate data rate than what was possible in a single channel. A diagram of the demonstration along with rate information is shown in Figure 4.10. Due to guard bands between channels, the rates between the RF frontends and channelizers was greater than 1.

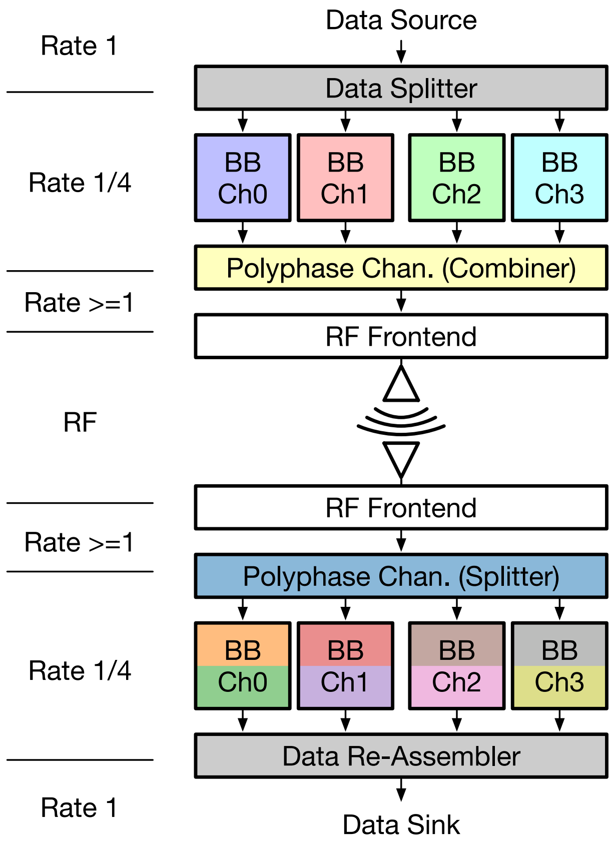


Figure 4.10: 4 Channel Demo (Color = Allocated CPU Core)

While the Spring 2020 Research Retreat demo was running at a much slower rate (≈ 4 Msps) than what can currently be achieved, the general principle can be adapted to basebands operating on wider channels. One consideration when scaling to wider aggregate bandwidths is that the channelizer needs to be able to operate at the rate of the full bandwidth signal. This potentially makes the channelizer the bottleneck in the system. There are several optimization techniques which can be applied to the channelizer including polyphase techniques like those discussed in [127]. If the aggregate bandwidth is wide enough that software implementations become impractical, relocating the channelizer into the RF frontend is a compelling alternative. While relocating any signal processing into an FPGA or ASIC connected to the data converters gives up some flexibility, channelizers are an ideal candidate as they are mostly agnostic to the underlying PHY standard used on each channel. This means that a single RF frontend with integrated channelizer could be used by many different radio standards, increasing its market appeal. By providing channelization in the RF frontend and allowing channelization in the PHY standard, software radio processing could be made practical at aggregate bandwidths previously requiring specialized hardware implementations.

4.4 Exploiting Frame Independence

So far, this chapter has discussed parallelization opportunities within a single instance of a radio and parallelization across multiple instances of a radio operating simultaneously on independent data-streams. Another opportunity exists if consecutively transmitted frames in a communication system are independent relative to each other. In this case, one could conceivably buffer up received frames and process each one independently with a different instance of the radio baseband. Assuming frame boundary detection occurs efficiently, this presents perhaps the simplest method to parallelize signal processing on a multi-core system. The developer would only need to translate the radio signal processing design into a single-threaded application running on a single core. Data for different frames would then be buffered and routed to different cores in the system, each running a single instance of the radio baseband. If there are n cores available on the system, n frames could be processed simultaneously.

This is in contrast to another approach where a single baseband design is partitioned to run across multiple cores of a CPU. Each core participating in the baseband processing would be responsible for a certain subset of operations in the design. The cores could operate simultaneously in a pipeline-like fashion. The difference between these two approaches is shown on a hypothetical radio system depicted in Figure 4.11 with the independent frame processing depicted in Figure 4.12 and the partitioned pipeline-like operation depicted in Figure 4.13. Note that both implementations use the same number of cores.

4.4.1 Comparing Processing Approaches

A reasonable question when posed with the two different processing options presented above is if there is a significant benefit from choosing one over the other. To tackle this

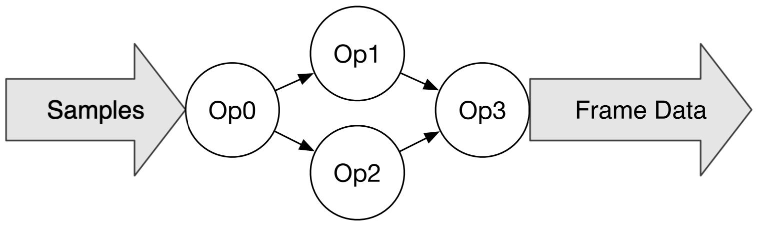


Figure 4.11: Example Radio Design

question, let us analyze the throughput, latency, and time to decode a given frame with these two schemes under the following assumptions:

- 20 Cores Operating at 2 GHz with 1 Op/Cycle.
- 1000 Operations Required / Sample.
- 60000 Samples / Frame.

Independent Frame Processing

With each frame processed independently (one frame per core), the time it takes to process a single frame is:

$$\frac{(1000 \text{ operations/sample})(60000 \text{ samples/frame})}{2 \text{ GOps/Second}} = 30 \text{ ms} \tag{4.1}$$

Because there are 20 cores operating independently, the throughput of the system is:

$$\frac{20 \text{ frames}}{30 \text{ ms}} = 666.7 \text{ FPS} \tag{4.2}$$

If performed as a batch, 20 frames finish decoding every 30 ms. If the processing is staggered, a new frame finishes decoding every 1.5 ms. However, the time for each frame to be decoded (from the time it first appears) is still 30 ms.

Pipelined Frame Processing

Let us take the case where a single radio baseband is distributed across the 20 cores. We will assume, for now, that the splitting is ideal with each core performing 50 operations/sample and there is no overhead in communication between cores. Let us also assume that each core operates on a block of 100 samples at a time, breaking the frame into 600 blocks. Each core processes a block in:

$$\frac{(50 \text{ operations/sample})(100 \text{ samples/block})}{2 \text{ Gops/Second}} = 2.5 \text{ µs/block} \tag{4.3}$$

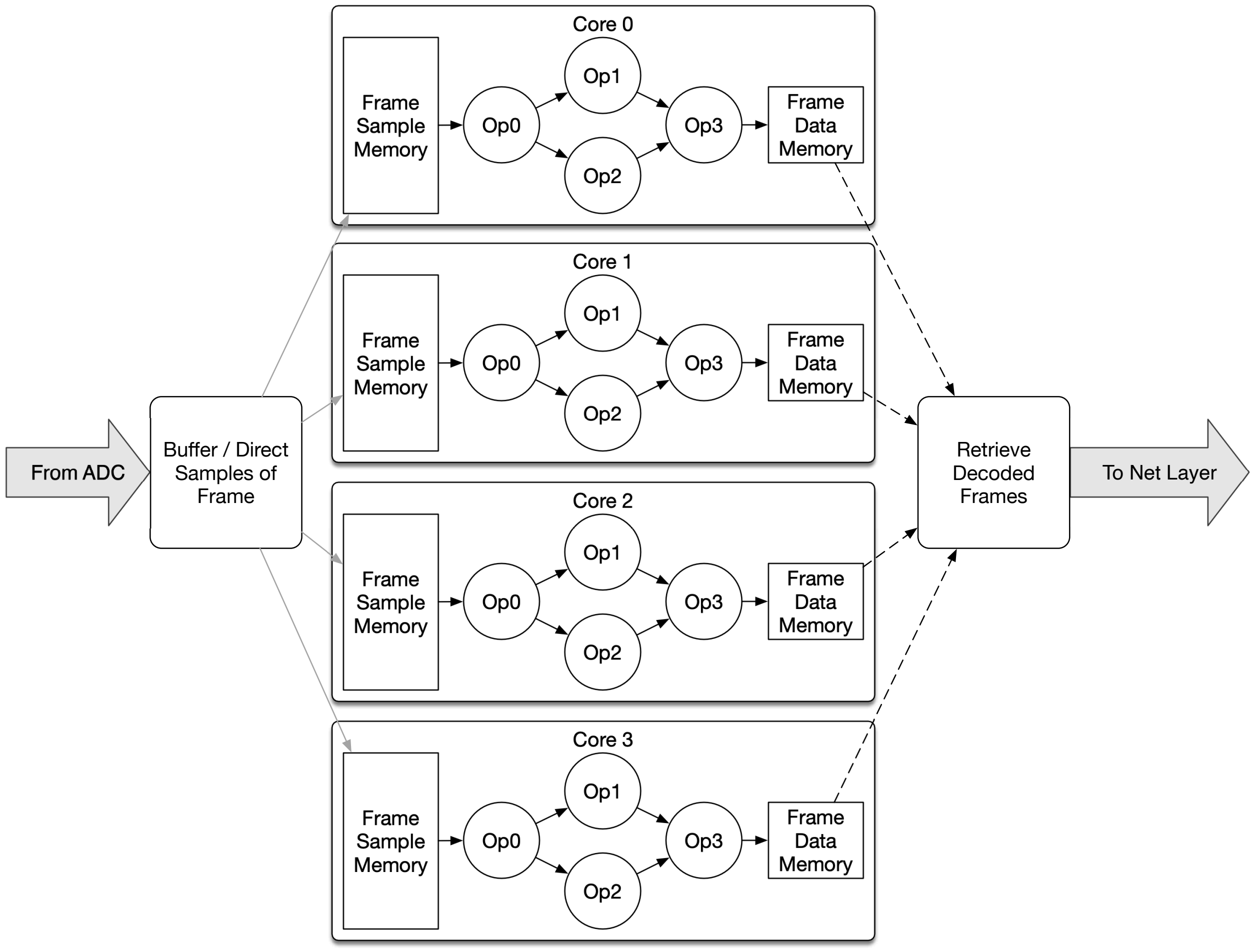


Figure 4.12: Example Radio Design with Independent Frame Processing

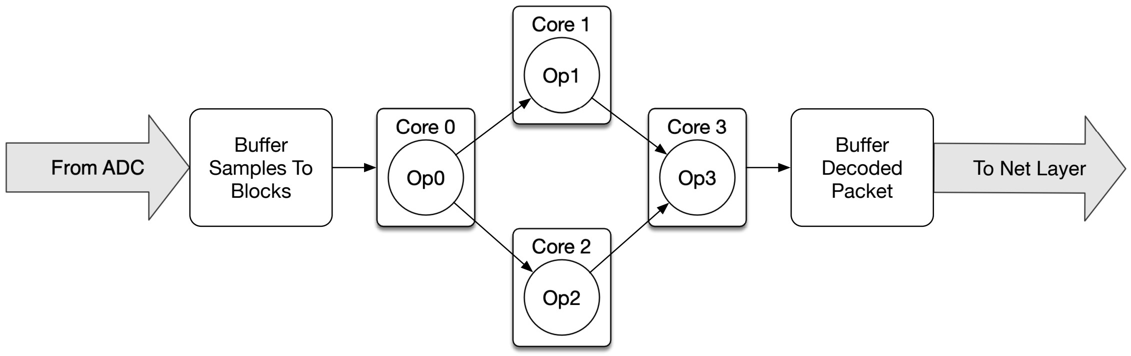


Figure 4.13: Partitioned Example Radio Design (Pipeline-Like Operation)

Let us assume that the frame processing occurs in a pipeline involving all cores. The time required to finish processing one frame is:

$$(600 \text{ blocks})(2.5\ \mu\text{s/block}) + 19 * (2.5\ \mu\text{s/block}) = 1.548 \text{ ms} \tag{4.4}$$

The 19 term comes from priming the pipeline. After that, a new block finishes processing every 2.5 µs. Because there are 600 blocks in the frame, the frame finishes decoding after 600, 2.5 µs, periods. Note that this is the worst-case scenario for frame latency. If there is a fork in the design where two cores operate in parallel on the same block of data, this number decreases.

With each core simultaneously working on an independent block, and the potential for the end of one frame and the start of the next one to be present in the pipeline at the same time, the throughput of this system is:

$$\frac{1}{(2.5\ \mu\text{s/block}) * (600 \text{ blocks/frame})} = 666.7 \text{ FPS} \tag{4.5}$$

Deciding on a Processing Methodology

A key takeaway is that both methods provide the same throughput but the version with independent frame processing on each core requires significantly more time to decode a given frame even though the pipelined approach technically has higher latency (time from data being available to process to the first partial result appearing). In most networking contexts, the time to completely decode the content of the frame is most important because the decision on whether to forward up to the next layer in the networking stack is based, in many circumstances, on whether the frame (or upper layer) checksums are correct. Checksum validation for most frame formats cannot be completed until the entire packet has been decoded. Because of that, the complete packet decode time can effectively become latency to an upper layer of the protocol stack. For some protocols, like TCP[7], increased latency can cause performance degradation. The challenge with running a single baseband across the whole system is obtaining as close to optimal partitioning as possible while limiting overhead. Load imbalance introduces inefficiency which slows down the entire signal processing chain, as does any overhead involved with communicating between cores. The ideal approach is likely a combination of partitioning basebands to run across multiple cores as well as using multiple instances of the baseband. Those multiple instances could either process different spectral channels of data using a filter bank approach or process separate frames.

[7]TCP's congestion control mechanism relies on acknowledgments to properly set the congestion window which controls the number of packets that can be in flight at once [128]. The acknowledgment cannot be created until the entire packet has been decoded, as it must be checked for correctness. As noted in [129], TCP starts conservatively and, especially for short flows, may never reach the maximum rate. The flow completion times have a noticeable contribution from the round-trip time which is impacted by the time to generate an acknowledgment.

Chapter 5

Introduction to Laminar: Optimizing DSP Compiler

It is clear from the past several chapters that there is a variety of parallelism available within radio designs and CPUs. However, exploiting these different modes of parallelism can be challenging for the designer, often requiring the design to be modified or re-organized to expose the parallelism in a way that the CPU can exploit. The Laminar Optimizing DSP Compiler [130] was written as a part of this project to provide a framework to experiment with different optimization techniques and to improve designer productivity by automating as much of the process as possible.

Laminar was written as a source-to-source compiler which takes the dataflow description of a radio design and produces C code which can then be passed to an existing C compiler such as clang/LLVM or a vendor provided optimizing compiler such as aocc (AMD Optimizing C/C++ Compiler) or icc (Intel C/C++ Compiler). The Laminar compiler was written in modules to support potential future expansion. The main components are:

- `simulink_to_graphml`: A set of scripts written Matlab m-code which walk a Simulink model graph and export it to GraphML [131] (an XML based graph interchange format). The export scripts export evaluated parameters for specific blocks as well as datatype information for arcs. If Stateflow FSMs are presents, they are synthesized to C using Simulink Coder and are inserted into the graph as black boxes. For multi-core implementations, the Simulink design should be annotated by the designer to identify the partitions different nodes reside in.
- `simulinkGraphMLImporter`: A C++ application which imports the GraphML file exported by `simulink_to_graphml` and converts it to Laminar's intermediate representation which is then written to a Laminar dialect GraphML file. This program also strips out unused information from the Simulink export file. This executable effectively constitutes the front-end of the Laminar compiler. Adding additional language support to Laminar would involve creating a similar program which would ingest the input language and produce a dataflow graph in the Laminar dialect of GraphML.

- `multiThreadedGenerator`: A C++ application which takes in a Laminar GraphML file and generates a multi-threaded C project including associated headers, I/O adapters, and Makefiles. This is the main Laminar application which contains the optimization passes, analysis passes, and C emitter.

Laminar uses the GraphML [131] graph interchange format as its method of graph serialization. GraphML was chosen because of its support for node and arc properties, support of nested graphs, support by other graph processing libraries such as NetworkX [132], support by visualization tools such as yEd [133], and its use of XML as its base. Because it is XML based, GraphML can be imported using a standard XML parser such as Xerces-C [134]. A general depiction of the Laminar flow from Simulink to C with the different scripts/applications and intermediate files is shown in Figure 5.1.

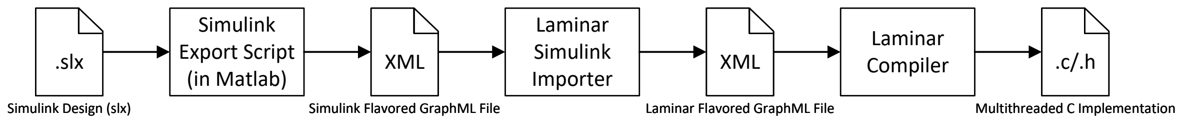

.slx
Simulink Export Script (in Matlab)
XML
Laminar Simulink Importer
XML
Laminar Compiler
.c/.h
Simulink Design (slx)
Simulink Flavored GraphML File
Laminar Flavored GraphML File
Multithreaded C Implementation

Figure 5.1: Laminar Components

From this point forward, references to Laminar will generally be referring to the primary `multiThreadedGenerator` application. The details and rationale behind using dataflow as the input and intermediate representation of the design will be discussed in section 5.1. The operating principles and assumptions made by laminar will be discussed in section 5.2. The general process performed by Laminar to produce C code from a given design, including some of the basic optimization passes, will be described in section 5.3

5.1 Design Representation: Dataflow Graphs

Laminar's input comes in the form of dataflow graphs. In a dataflow graph description of a design, operators are written as nodes and data flowing between operators is represented by directional arcs. The existence of an arc from a node A to a node B indicates a dependency of node B on the result of node A. The inputs into and the outputs from the system are typically represented by source and sink nodes, although they are sometimes implicit. Laminar represents all the inputs coming into the system as coming from one super input node and all outputs from the system going to a single super output node. There are also single super nodes representing unconnected operators, terminated (unused) results, and results to be visualized[1]. The dataflow graphs used in Laminar are *streaming* where a continuous flow of

[1]Visualization sinks are currently unsupported by the Laminar flow but could be implemented in the future.

operators exists at the input source node with a continuous stream of results at the output sink node. A graphical depiction of a Laminar dataflow graph is shown in Figure 5.2.

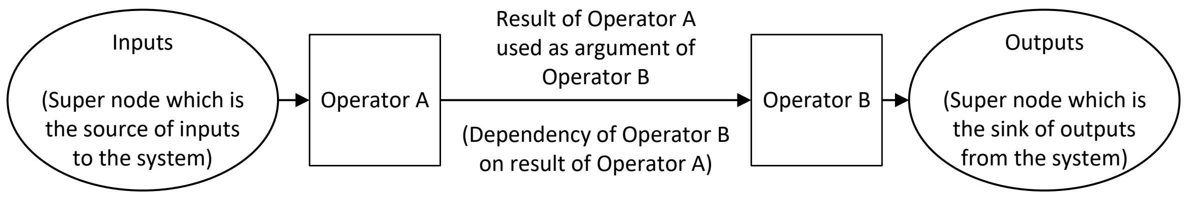

Inputs
(Super node which is the source of inputs to the system)
Operator A
Result of Operator A used as argument of Operator B
(Dependency of Operator B on result of Operator A)
Operator B
Outputs
(Super node which is the sink of outputs from the system)

Figure 5.2: Laminar Dataflow Graph Description

Streaming dataflow graphs are a natural description for DSP with most graphical descriptions of DSP algorithms being expressed in a dataflow form. In these graphs, the unit of data being transferred through the graph, from operator to operator, is generally samples. New samples are typically introduced at a fixed rate with the period between samples being referred to as a cycle time of the system. If the sample is coming directly from an ADC, the data converter rate sets the cycle time. In Laminar, the number of samples ingested by each block and results outputted are statically known, causing its descriptions to be classified as *synchronous dataflow* as described in [11]. In Laminar, standard nodes ingest single samples at their inputs and return single samples at their outputs in a cycle. Special rate transition nodes are exceptions to this rule and can accept/produce different numbers of samples.

One important property of DSP dataflow graphs is the existence of delay nodes, labeled as z^{-n} with n being the delay expressed in samples. These blocks represent state in the dataflow graph, as they save the past n samples to be returned later. By returning an input from a previous cycle, these delays can break the dependency chain of the signals passing through them. This dependency breaking property allows operators before and after the delays, in the absence of other dependencies, to operate in parallel. This also allows loops to be present in the design, so long as each loop contains at least one delay. Without a delay, there becomes a circular dependency within a single clock cycle which may not be resolvable.

The DSP description of a dataflow graphs can be easily mapped into an abstraction used in general digital hardware design called Register Transfer Level (RTL). RTL describes a digital hardware design as sets of combinational logic between registers[2] [135]. Delays in DSP designs map to registers or memories in RTL with stateless operators mapping to combinational logic. The cycle of new samples being introduced and propagated through delay nodes in the DSP algorithm translates to clock cycles driving the registers in the design. As in DSP design, loops may exist in RTL if they contain at least one register or state element to break the dependency chain. Without a state element, the loop is referred to

[2]For more information on Hardware Description Languages (HDLs), RTL, and digital design, see texts like [95].

as a *combinational loop* and is typically not accepted as a valid digital hardware description [136]. While such a combinational loop could be physically built, its behavior is ill-defined under digital design abstractions and would require more detailed analysis such as analog analysis.

There are several different options for representing dataflow graphs, including both graphical and textual. One example is MathWorks Simulink [41] which provides a graphical interface to describe systems and models including DSP designs. It is capable of simulating the design and provides tools to analyze the design and to produce standalone hardware or software descriptions of the design under certain constraints. MathWorks also provides a set of add-on toolboxes for DSP [42] and communications [43] which aid in design and verification. Cyclops was originally described in Simulink because of its dataflow representation, the generally accepted golden reference blocks and functions available in the MathWorks DSP and communications toolboxes, and the availability of Simulink HDL Coder [137] to translate the design into an HDL description for hardware. While a Simulink to C/C++ tool (Simulink Coder) exists [45], it did not, to the author's knowledge, support multi-threaded targets at the time this project began[3].

While graphical tools like Simulink or LabView may first come to mind when considering dataflow descriptions of DSP algorithms, it is possible to express them textually. Thanks to the close relationship between RTL and DSP dataflow, designs can typically be naturally represented in text using Hardware Description Languages (HDLs). HDLs, such as Verilog and VHDL, provide textual ways to express hardware designs either structurally by describing the connection between modules or behaviorally by describing operations that occur on a set of inputs. Legal HDL descriptions which do not use unsynthesizable simulation constructs[4], can be synthesized into structural RTL descriptions. An example of such a mapping from behavioral Verilog (shown in Listing 5.1) to an RTL description via the open source Yosys synthesis tool [138] is shown in Figure 5.3. Note that the two sum operators in this example can execute in parallel because a register (shown as $dff for *D Flip-Flop*) separates them and there are no other dependencies between these operators. Also note that the integrator contains a register in the feedback path, making the loop legal. To illustrate the close relationship of DSP dataflow graphs to RTL descriptions, the same design expressed in MathWorks Simulink is shown in Figure 5.4. In the Simulink description, the clock signal is implicit at the delay nodes. All registers operate in the same clock domain unless separated by clock domain crossing nodes[5].

[3]Simulink Coder has begun supporting threaded targets and SIMD support. For more information, see section 1.3.

[4]Some HDL languages, such as Verilog have their origin as documentation and simulation languages [95] and support operations that make sense in simulation but cannot be converted to hardware. As such, not every Verilog design can be converted into an actual structural hardware description.

[5]Simulink supports several different simulation modes including ones with continuous step sizes. For DSP designs such as Cyclops, the solver is set to use discrete step sizes. While some blocks allow inputs and outputs of conflicting rates, Laminar does not support this and requires clock domains be clearly delineated with explicit rate transition blocks with clearly defined semantics.

```verilog
module integrator_with_offset_out_pipeline(input [15:0] in,
                                           input [15:0] offset,
                                           input clk,
                                           input rst,
                                           output reg [15:0] out);
  reg [15:0] sum_next;
  reg [15:0] sum;
  reg [15:0] sum_offset;

  always @(*) begin
    if(rst) begin
      sum_next = 16'd0;
    end else begin
      sum_next = sum + in;
    end

    sum_offset = sum + offset;
  end

  always @(posedge clk) begin
    sum <= sum_next;
    out <= sum_offset;
  end
endmodule
```

Listing 5.1: Example Verilog Integrator with Offset and Output Pipeline

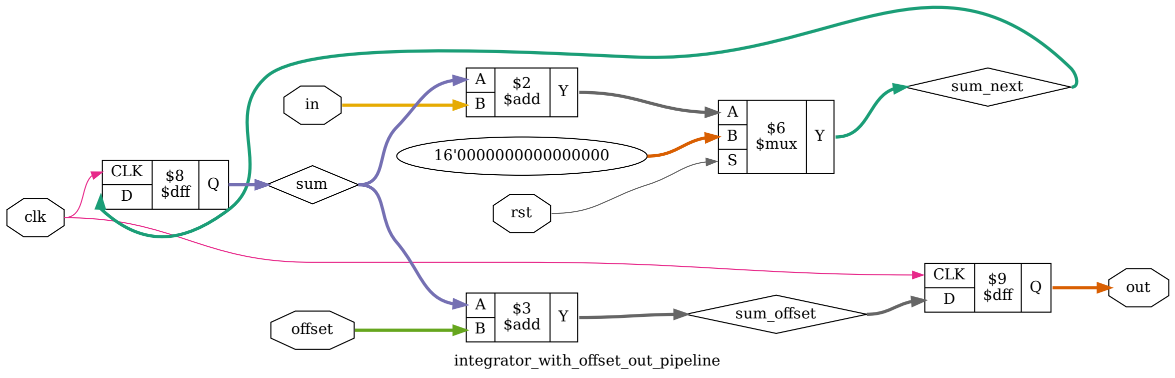


Figure 5.3: Yosys Synthesis of Example Verilog Design

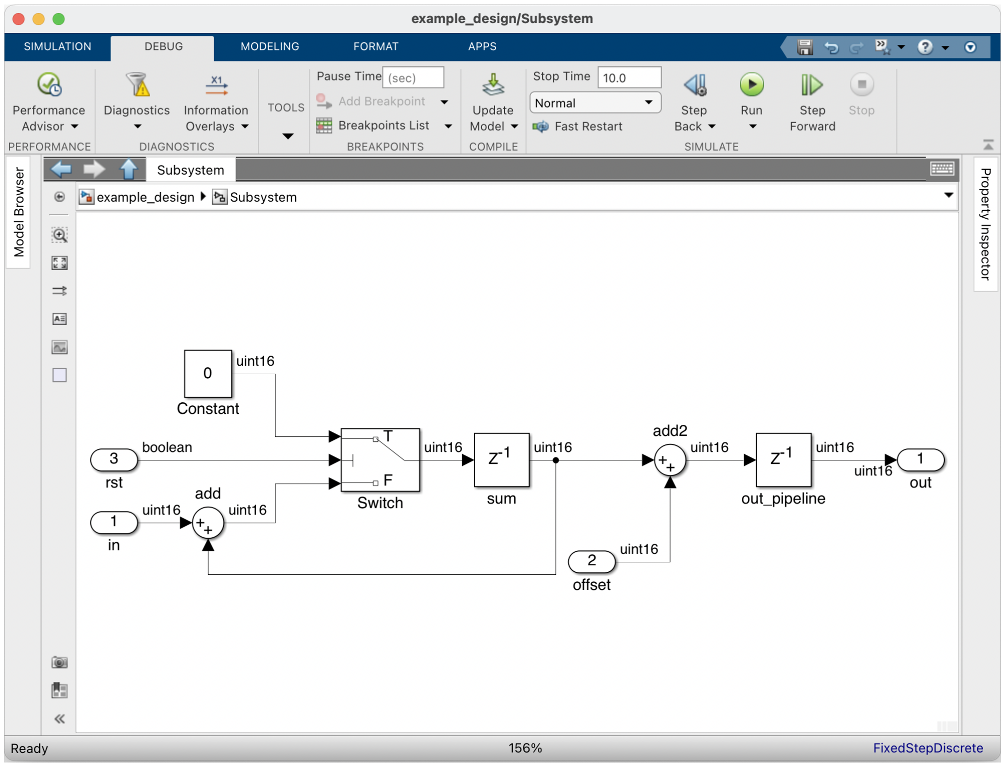


Figure 5.4: Simulink Representation of Example Verilog Design

As demonstrated above, dataflow graphs have the very desirable property that dependencies between operators are explicit and unambiguous in the graph. Prerequisites can be determined by simply tracing back along the arcs going into a particular node. Operator-level parallelism is also explicitly expressed with operators that do not have a dependency chain between them being able to execute in parallel. It was because of the power of the representation as well as the tendency of DSP algorithms to be described diagrammatically as dataflow that *streaming, synchronous, dataflow graphs* were selected as the input and intermediate representation for Laminar. Simulink was selected as the frontend representation due to the simulation and analysis tools provided for design as well as the fact that Cyclops was originally described and simulated in Simulink. Other dataflow graph representations, such as synthesized HDL could potentially be adopted as input formats (under specific clocking constraints).

5.2 Operating Model

As discussed above, Laminar operates on dataflow graph descriptions of a DSP design. One of the major features of Laminar is to produce a multi-core implementation of the design using the partitioned approach mentioned in section 4.4. While automated partitioning would be best from a designer productivity standpoint, it is a complex problem balancing multiple concerns including load balancing across threads as well as accounting for potentially heterogeneous communication overheads from modern hierarchical cache designs present in many CPUs. For now, Laminar requires that the partitioning be specifically annotated by the designer in the input graph. This is accomplished by placing a specially named *Constant* node within a subsystem in Simulink. The inclusion of this annotation node indicates to Laminar that all operators/nodes contained within the subsystem are to be contained within the given partition. In the case that a node is nested within multiple subsystems with conflicting subsystem annotations, the most specific annotation (the lowest in the hierarchy) takes precedent. The name of the annotated node can be either `LAMINAR_PARTITION` or `VITIS_PARTITION` with *vitis* being the original name of the Laminar compiler and unrelated to the Xilinx tool of the same name.

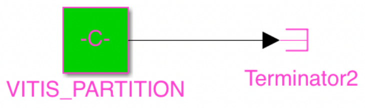


Figure 5.5: Laminar Partition Annotation

Currently, Laminar maps partitions one-to-one onto individual POSIX threads (pthreads). These threads are then typically mapped to a specific logical core on the system by setting the pthread *affinity mask* for each thread, limiting the core eligibility of each thread to a single core. Each of these threads shares the same virtual address space and can communicate via shared memory. On modern Linux platforms, pthreads are implemented as light-weight processes (LWP) and are scheduled by the operating system [139], [140]. To prevent other processes from being scheduled on the same CPU cores, the `isolcpus` kernel option is used to isolate all but one core from standard OS scheduling. For a user process to run on an isolated core, the process/thread affinity must be explicitly set. For more information on the `isolcpus` option, see [141], [142].

Any time there is an arc passing from one partition to another, information must be transferred between threads. This is accomplished by the automatic insertion of single producer, single consumer, FIFOs at each arc cut by a partition boundary. Code is automatically added to the source thread to write information to be sent into the FIFO. Similarly, code is automatically added to the destination thread to read data from the FIFO. Because there is a cost to access the FIFO due to the underlying cache coherency system, information is typically sent in blocks of multiple samples to amortize the fixed cost associated with FIFO access. The blocking size is configurable via the Laminar Command Line Interface (CLI).

Each partition is written into its own `.c` and associated `.h` files. Each file contains a *compute function* which contains the operators in the DSP design, a *thread function* which conforms to the pthread standard (accepting a void pointer argument and returning a void pointer) and manages interactions with FIFOs as well as calling the compute function, and a reset function to clear the state within the partition. Each thread function is composed of an outer loop which generally[6] performs the following actions:

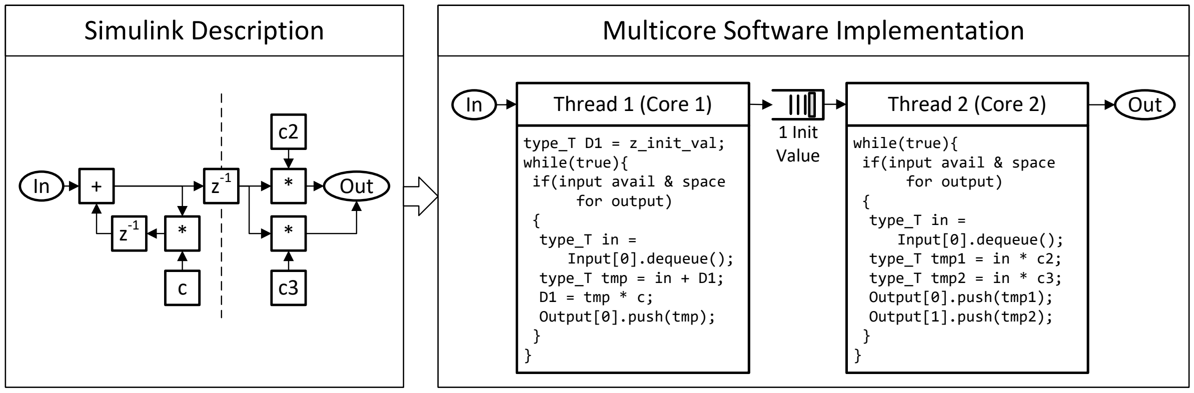
1. (Optional) Check telemetry timer and write telemetry to file. Typical telemetry reporting period is 1 second.
2. Wait for input FIFOs to be ready (have date to read). Poll input FIFOs until all are ready.
3. Copy a block from each input FIFO into local buffers. Update the state of the input FIFOs to indicate the read (dequeue) has occurred.
4. Execute the partition compute function
    a) Iterate over samples in the block, executing the operations in the DSP design for each sample. Results are written into buffers passed to the function through the function call in the outer thread loop.
5. Wait for output FIFOs to be ready (have room to write data). Poll output FIFOs until all are ready.
6. Write a block of results into output FIFOs. Update the state of the output FIFOs to indicate the write (enqueue) has occurred.

Note that no global schedule is created for the execution of the compute functions by the different threads. Each thread makes its own decision about when to execute based on the availability of input data and space for results. This makes the Laminar execution model *self-timed* in the Lee taxonomy described in [13].

The compute function is composed of the various operators in the partition emitted in such an order that no operator runs before the values it is dependent on are available. Operators are emitted in a style similar to *static single assignment* (SSA), which is an intermediate representation used by several C compilers[7]. Temporary variables are created for most operators which are assigned once. One exception is variables that are set as the result of conditional logic, such as a multiplexer. In this case, the variable can be assigned in multiple locations. Arrays containing the samples to process are passed to the compute function by the thread function along with array pointers for results to be written into. State for the partition is stored in a persistent structure which is passed by pointer to the compute function.

[6]Experimentation with different execution modes including double buffered FIFOs and operating on FIFO data in-place (without an intermediate copy) were tried and are still available in the Laminar CLI options. This represents the operations conducted by each thread in the standard, default, mode of operation.

[7]The LLVM assembly language is based on SSA [143]. GCC also makes internal use of SSA when optimizing code [144]. See [145] for information on static single assignment and its representation.



```
type_T D1 = z_init_val;
while(true){
 if(input avail & space
      for output)
 {
  type_T in =
      Input[0].dequeue();
  type_T tmp = in + D1;
  D1 = tmp * c;
  Output[0].push(tmp);
 }
}
```

```
while(true){
 if(input avail & space
      for output)
 {
  type_T in =
      Input[0].dequeue();
  type_T tmp1 = in * c2;
  type_T tmp2 = in * c3;
  Output[0].push(tmp1);
  Output[1].push(tmp2);
 }
}
```

Figure 5.6: Laminar Multi-Thread Generation Flow (Pseudo-code Result)

In addition to the partitions, Laminar also generates I/O adapter threads which provide different ways to interface with the DSP design. The adapters provide the shim between a given inter-process communication construct and the inter-partition FIFOs present within the Laminar generated DSP design. From the perspective of the Laminar partition threads, I/O appears like any other inter-partition FIFO. Current supported adapters include:

- POSIX Pipes: Support exchanging data via named POSIX pipes.
- Shared Memory: Supports exchanging data via OS allocated shared memory which is mmap-ed into the memory space of the communicating applications. The semantics of this interface were defined to be similar to the POSIX pipes interface and are defined in BerkeleySharedMemoryFIFO [146]. *This is currently the highest performance I/O adapter.*
- BSD Network Socket: Supports exchanging data via network sockets. This allows stimulus to be provided and results analyzed by a different system. This is particularly useful for checking a generated design against the original Simulink via the creation of a custom block in Simulink which opens a socket connection to the generated design. An example of this is shown in [147].
- Constant: Passes constant values to the design. Used to test basic functionality of the compiled system.

Like the FIFOs in the Laminar generated partitioned design, the I/O threads transact with outside applications in blocks. The block size is not necessarily the same as the block size used within the application with re-buffering occurring in the I/O thread.

5.3 Dataflow Graph to C Translation

Laminar, like many modern compilers, is constructed around the idea of *passes* over the design which modify it in some way. Some of these passes are essential for the design to be properly converted to C while *optimization passes* modify the design with a focus on produce better quality of results (QoR). It should be noted that, while the optimization passes are important, they do not constitute all optimization techniques taken by Laminar. Other mechanisms include vector/matrix support, specialized implementations of some DSP constructs, and different FIFO implementations. The options used in essential compiler passes can also have a strong impact on achieved performance and therefore cannot be neglected.

The general steps that Laminar takes to convert a design to C are[8]:

1. Radio Design with Annotation: The designer annotates the design with partition and sub-blocking information.

2. Import Design: The dataflow graph is imported into an in-memory representation within Laminar on which design passes can be performed.

3. Design Pruning *(Optimization Pass)*: The design is pruned of unused operators and arcs.

4. Clock Domain Handling: Clock Domains are discovered and configured.

5. Enabled Subsystem Context Expansion *(Optimization Pass)*: Expands conditional execution blocks to include additional combinational logic, when possible.

6. Discover and Mark Contexts *(Partial Optimization Pass)*: Discovers and marks contextually executed segments of the design including enabled subsystems and multiplexers. Multiplexer context discovery can be viewed as an optimization pass as it seeks to include as much combinational logic as possible in the different mutually exclusive execution blocks.

7. Blocking and Sub-Blocking *(Optimization Pass)*: Creates a global blocking domain to facilitate partitions operating on blocks of samples transacted by FIFOs instead of single samples. Creates sub-blocking domains and specialized implementations of some operators so that they can operate on sub-blocks of samples, rather than a single sample at a time inside the loop created by the global blocking domain.

8. Context Encapsulation: Wrapping operators in the discovered execution contexts, such as enabled subsystems, multiplexers, clock domains, and blocking domains. This wrapping helps the emitter when the design is ultimately written to C files.

[8]For an in-depth look at all the steps taken by Laminar see `src/docs/generator.md` [130]

9. Create Context Update Nodes: Inserts nodes which represent updates for values affected by conditional execution, such as the output of a multiplexer.

10. Discover Partition Crossings and Insert Partition/Thread Crossing FIFOs

11. FIFO Delay Ingestion: Pull delays at the boundaries of thread crossing FIFOs into the FIFOs as initial conditions. Note that these delays need to exist outside of contextual execution domains (with a few exceptions).

12. FIFO Merging *(Optimization Pass)*: FIFOs between the same pair of partitions and to/from the same contexts are merged to help amortize FIFO costs.

13. Report FIFO Communication

14. Check for Inter-Partition Deadlock Conditions

15. Create State Update Nodes: State updates can occur any time after the downstream operators have been executed. Instead of forcing all state to update at the end of a compute cycle, similar to how clock edges update all state in a hardware design, state update nodes perform the state update operation and can be scheduled like other operators.

16. Intra-Partition Operator Scheduling: Operators within partitions are scheduled such that each operator will only be emitted after operators it is dependent on.

17. Report Partition Workload

18. Emit C: This includes emitting the C files for each partition including the thread, compute, and reset functions. The operators are emitted in their scheduled order within the compute function.

19. Emit I/O Adapters

Many of the core functions will be discussed in more depth in the following subsections. Optimization passes will be primarily covered in chapter 6.

5.3.1 State Update Nodes and Scheduling

One interesting feature of implementing a streaming DSP design in software is that there is more flexibility in how state updates can be handled compared to typical synchronous digital logic. In a synchronous digital system, state is predominately updated on clock edges. For registers in the same clock domain, this update occurs at approximately the same time, subject to clock jitter and clock skew. For DSP designs implemented as synchronous digital hardware, this means all delays within the same clock domain update at the same time. The combinational logic implementing the operators DSP design operates between the clock edges with their result required to be stable before the next clock edge. The maximum clock rate is set by the path in the design that takes the longest to produce its result, called the *critical path*[9].

Software implementations of DSP algorithms differ from the synchronous hardware implementation in how results are computed and states are updated. Unlike the hardware implementation where the results of combinational logic are computed in parallel, software computes the results of combinational logic by executing discrete instructions, with most producing their own intermediate result which may be stored in a variable (backed by either a CPU register or memory). State in the DSP design is likewise stored in variables. Unlike the hardware design where state updates on a clock edge, state in the software implementation is updated when the value of the state variable is overwritten by a new value. A sample is completely processed after all operations in the design have been performed on it and state has been updated for the next sample. This process can, and likely does, take multiple clock cycles of the CPU executing instructions. While all state updates could be saved until the end of sample processing, analogously to how all state is updated at the clock edge in hardware, this is not strictly necessary in software. Rather, a state variable can be updated any time after its result has been used by all dependent operations[10] with their results stored in their own temporary variables. Because Laminar emits code in a single assignment like style, this occurs after most operations.

To take advantage of this property, Laminar explicitly represents state updates with special *StateUpdate* nodes. A compiler pass in Laminar adds these state update nodes to the design. Each stateful node type in Laminar is responsible for implementing a state update node placement algorithm. Most delay-like state elements use the same basic approach, creating a state update node which is dependent on the input to the stateful node being computed and is also dependent on the immediate downstream nodes from the stateful node being computed. An example of this can be seen in Figure 5.7 where the state update node is dependent on the delay input[11] as well as the two operators dependent on the delay. These state update nodes can be scheduled like any other operator and are emitted like other operators by the emitter. There are several potential benefits for scheduling state updates to occur earlier. Scheduling a state update shortly after the value has been used could potentially be more cache friendly than waiting until the end of computation for a sample when the state may have been evicted from lower-level caches. In superscalar processors, there is also a potential load/store unit bottleneck if all state is updated at the end of sample processing. By distributing state updates throughout the computation, there is the chance

[9]For more information on clock networks, timing analysis, and digital design in general, see texts like [95].

[10]This style of updating state actually does have a hardware analog called *asynchronous design* where state updates occur via a form of handshaking without the use of a global clock. While this concept has been around for quite some time and commercial examples do exist, it is less commonplace than synchronous design - requiring a relatively major change in design methodology and tool-flow to exploit. For more information on asynchronous hardware design, see [148].

[11]Laminar views scheduling the Delay node as scheduling the computation of the delay input. Because of this convention, the StateUpdate node can be dependent on the Delay node rather than the upstream node.

to take advantage of underutilized load/store units during computationally heavy segments of sample processing.

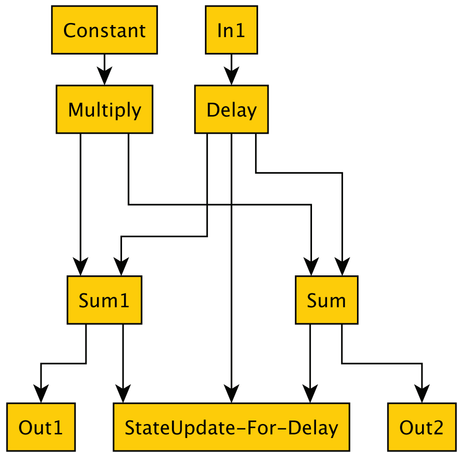


Figure 5.7: Partial Laminar Intermediate Representation of Cyclops Golay Correlator Segment after StateUpdate Node Insertion, No Sub-blocking

5.3.2 Intra-Core Operator Emit Scheduling

An essential step in producing a procedural program from a dataflow graph is determining the order in which operations are written in the resulting C file. C and the executable that is compiled from it are imperative in nature, presented as a series of instructions which a programmer generally views as occurring in program order[12]. Dataflow, as discussed earlier, does not specify the order in which operations are executed but rather specifies the dependencies between operators. The scheduler's role is to take the constraints imposed by the dataflow graph and produce an ordering of operators such that, if each operator was executed one at a time, the correct result would be computed. In other words, it produces a schedule of operators such that no operator is executed before its prerequisites have been computed. This schedule directly translates to the order in which operations are written, as instructions, into the executable file.

If the target CPU is in-order, and the C compiler does not change the order of operations, the scheduler would determine the order in which instructions would be executed. However, this will not be the case for most modern circumstances. As discussed in section 3.1, modern high-performance CPUs such as the ones used in this project tend to be out-of-order, featuring look-ahead windows from which they can select instructions to execute. Because

[12]For more information on imperative languages, including C, and their heritage from von Neumann architectures, see [149], [150].

this type of CPU can change the order in which instructions are executed at runtime, the instruction order in the executable does not necessarily determine the schedule in which operations are executed. However, there is a practical limitation on the look-ahead window size which means that, even if the CPU was performing some optimal form of scheduling, the order in which instructions are present in the executable can still affect performance.

Modern C compilers do have the flexibility to re-order instructions, so long as the underlying semantics of the program are preserved. The parameters of the C compiler optimizations can be controlled via a host of CLI flags. While fine grain control is available, the number of compiler options can be staggering. As such, a popular technique is to use a set of flags which roughly configure the aggressiveness of optimizations from virtually none (`-O0`) to high (`-O3`). Modern compiler can even re-order instruction instructions in potentially unsafe ways[13] via the `-Ofast` flag.

Because the C compilers can re-order instructions, it is important to determine if the ordering generated by the Laminar intra-core scheduler has any impact on the resulting performance. If the C compiler is capable of determining an optimal instruction ordering for a program, the execution time should be the same regardless of the input instruction ordering. To test this theory, two different scheduling heuristics were implemented in an early version of Laminar. A single-thread realization of Cyclops was generated with each heuristic. In addition, a single-thread version of Cyclops was generated with Simulink Coder (circa 2019). These different implementations were generated with a variety of compiler and optimization flags and benchmarked. The results are shown in Figure 5.8

Interestingly, there is a considerable difference in performance between the BFS and DFS heuristics used by the Laminar scheduler, even with aggressive compiler optimizations enabled. Note that the schedule only changed the order in which the operations were emitted in the C file. This is not entirely unexpected as the optimal scheduling problem for parallel machines (which the different execution units in a superscalar CPU can be viewed as) is notoriously NP-hard[14] [151] and it would be infeasible for the C compiler to perform for all but the most simple programs (or small program segments). This does mean, however, that the organization of the C file is important to the achieved performance, even when using an optimizing C compiler. This is a theme which will be re-iterated several times with the Laminar compiler and starts here with the intra-core emit scheduler.

Scheduling Procedure

Having discussed the objective of the scheduler and noting its importance, this section will dig deeper into the mechanics used. There are two main steps when scheduling a partition:

1. Disconnecting State Elements in the Scheduling Graph

[13]Typically, these are transforms and re-orderings which are legal in a purely mathematical sense but are unsafe in a numeric sense when paired with floating point arithmetic.

[14]NP-hardness stems from complexity analysis and is a class of problems for which no known polynomial time algorithm exists.

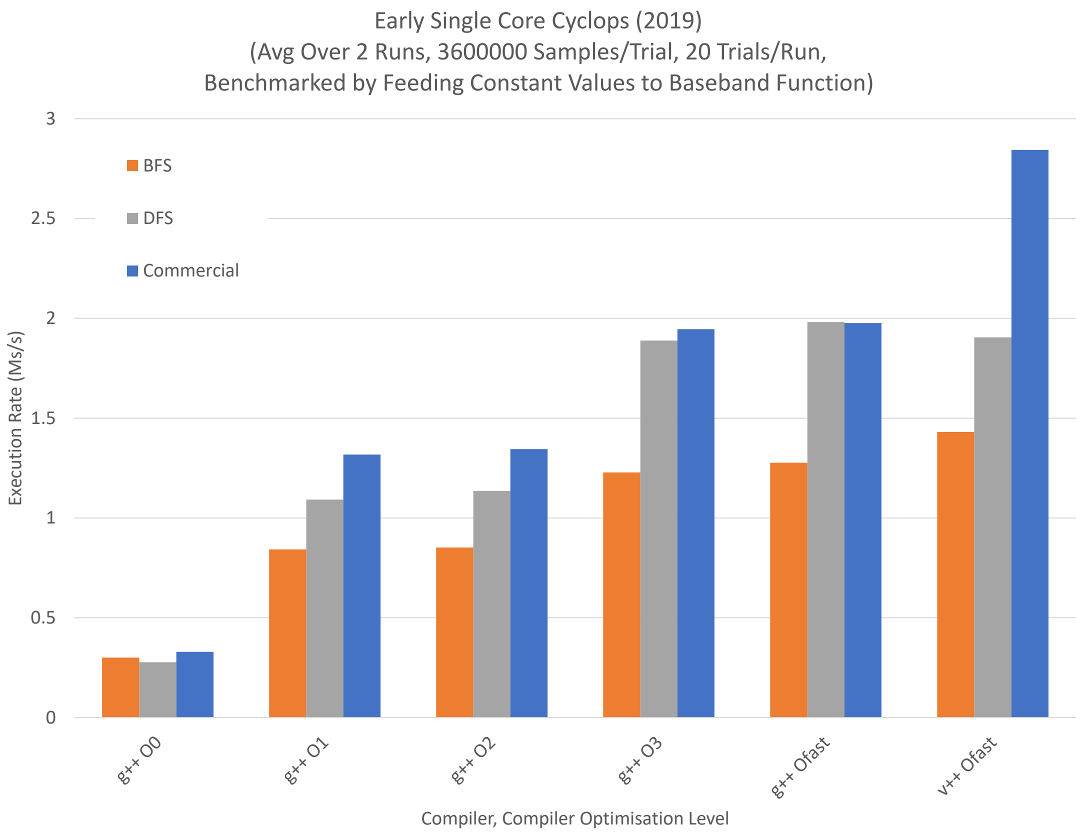


Figure 5.8: Performance of Early Cyclops Single Core Target with Different Generators, Scheduling Heuristics, Compilers, and Compiler Options

2. Generating a Schedule of Operations (Must be a Topological Ordering of Nodes in the Graph)

Scheduling nodes in a graph is a well-known problem in many fields including computing as well as Industrial Engineering and Operations Research (IEOR). An analogous IEOR problems to operator scheduling is assembly line scheduling [12][15]. Owing to its wide applicability across multiple fields, scheduling and its variants are heavily researched. Due to its NP-hardness, practical solutions are restricted to heuristics and approximations algorithms which make certain guarantees on the objective value of a result relative to the optimal. Works such as [151] detail a variety of optimization algorithms discovered for a subset of NP-hard problems.

[15][12] identifies the synchronous dataflow graph scheduling problem as identical to assembly line problems and cites [152], now commonly known as Hu-Level scheduling, as a well known heuristic. Based on a definition of list scheduling as described in [153], [152] can be considered to be in the class of list scheduling algorithms.

The techniques used in the Laminar compiler are variants of the well-known greedy *list scheduling* algorithm. The algorithm, as described in [153] takes the following approach:

1. Find nodes in the graph which are ready to execute (i.e., have no incoming arcs) and place them in a "ready" list
2. While the list is not empty:
    a) Select a node from the list and schedule it.
    b) Identify nodes connected to the output of the scheduled node. These are candidate nodes to be added to the "ready" list.
    c) Remove the scheduled node from the graph
    d) Check the candidate nodes and see if any are ready to execute (no longer have any incoming arcs).
    e) Add newly ready nodes to the "ready" list, if any such nodes exist
3. If all nodes a scheduled, success! If nodes remain, a circular dependency exists in the design and a topological ordering cannot be generated.

The different list scheduling heuristics come from how nodes are selected off the "ready" list. The heuristic only has an impact if there are multiple nodes present in the list at once. The DFS heuristic implemented in Laminar treats the "ready" list as a stack (last-in, first-out). As nodes are scheduled and newly ready downstream nodes are placed on the stack, they will be the next nodes to be scheduled. The BFS heuristic treats the list as a queue (first-in, first-out). This heuristic places newly ready nodes at the end of the queue to be scheduled after all of the existing nodes on the list.

It is important to note that the actual scheduling problem is more complex than the assembly line sequencing problem as described in [152]. Because of the presence of the shared CPU cache, the amount of time each operation/task takes can vary. Because of the limited space in lower-level caches and registers, the order in which operations are executed can determine what values reside in the cache and which were evicted. Re-using these evicted values requires longer delays. This creates the undesirable characteristics that the cost of each operation depends on the execution schedule itself.

One possible explanation as to why the DFS list scheduling heuristic appears to perform better than BFS is that it may make better use of temporal locality in the cache. With BFS, the execution traverses down the dataflow a layer at a time, generating a potentially large volume of intermediate results. As the depth of the tree increases, these intermediate results may not fit into the cache. DFS, by contrast, restricts the intermediate state to values which should be used relatively soon, improving temporal locality which caches serve. There is a downside to this approach, however, in that it may produce a string of dependent instructions that leave CPU resources unused. This effect can be mitigated by wide look-ahead windows in out-of-order CPUs but can become a problem for large chains of independent instructions that exceed the look-ahead window.

Due to its performance relative to BFS and its relatively competitive speeds compared to a commercial dataflow graph to C solution, DFS was selected as the default heuristic for Laminar. However, because of the wealth of scheduling heuristics available and the added complexity of cache interactions, there is likely additional exploration that can be conducted in this space. Adding additional heuristics to Laminar would be relatively simple, involving the modification of the selection criteria used by the list scheduling algorithm. Algorithms outside of list scheduling could also be implemented by replacing the call to the list scheduling function with one of the developer's choosing.

Modifications to List Scheduling Laminar makes some modifications to the list scheduling algorithm by allowing scheduling to occur hierarchically. This allows nodes in conditional execution contexts to be scheduled together, potentially reducing the number of condition checks required. In this case, the contexts are represented as single nodes in the scheduling graph. When a node representing a context is selected, the list scheduling algorithm is called recursively on the nodes within the context. After the last node in the context is scheduled, scheduling is allowed to continue as usual. Nested contexts are allowed with multiple levels of recursion of the scheduling algorithm.

Breaking Dependency Chains

One important requirement of the scheduling problem is that the ordering constraints form a Directed Acyclic Graph (DAG). Surprisingly to some, valid DSP algorithms represented as dataflow graphs are not required to be DAGs. Recalling from section 5.1, while combinational/algebraic loops are not allowed, cycles are permitted so long as a delay exist in the loop. At first glance, this appears incompatible with producing a legal schedule which can be executed on the processor. However, there is a subtle difference between the provided representation of the DSP design and the reality of the software implementation. In software, the objective is to produce a schedule of operations which can be used to process a single sample. This schedule accomplishes what, in hardware, would be computed in combinational logic between clock edges (not inclusive). For a given sample, outputs of delays are effectively constants. The inputs of delays are similar to sinks — their input must be computed before the sample processing has finished. Before the next sample is processed, the value at the sink node of the delay is transferred to the constant node.

Instead of creating two nodes for the purposes of scheduling, Laminar takes a shortcut. Because constants do not need to be scheduled, the outputs of delays are disconnected in the scheduling graph (a replica of the design graph used for scheduling). The delay nodes themselves represent the inputs to the delay and are scheduled. By disconnecting the output arcs, the dependency chains in legal loops are broken and, assuming no combinational loops exist, the resulting scheduling graph is a DAG. A visual depiction of the different delay representations used by Laminar is shown in Figure 5.9.

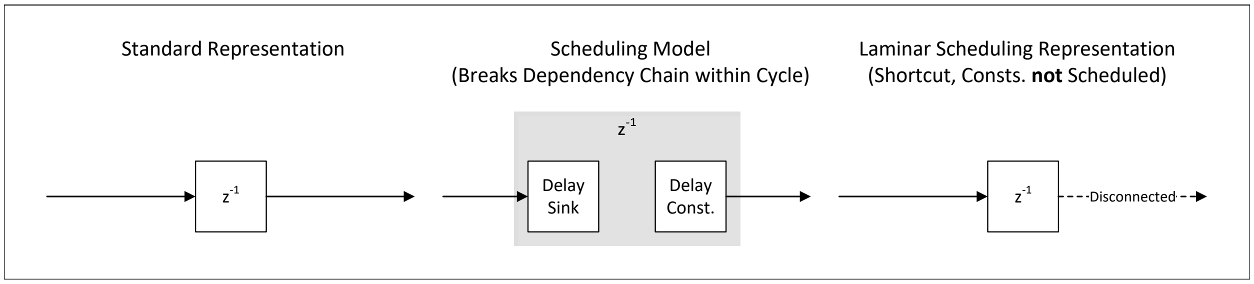

Standard Representation
Scheduling Model
(Breaks Dependency Chain within Cycle)
Laminar Scheduling Representation
(Shortcut, Consts. **not** Scheduled)
z^{-1}
z^{-1}
Delay Sink
Delay Const.
z^{-1}
Disconnected

Figure 5.9: Laminar Delay Representations

5.3.3 Emitter

The Laminar emitter is responsible for producing the various C files associated with the design. Routines exist for generating the thread functions, compute functions, reset functions, I/O functions, startup code, and Makefiles. The emitter itself is broken into several different functions, mostly contained in the `MultiThreadEmit` namespace, and implemented in `src/Emitter/MultiThreadEmit.cpp` and `src/General/EmitterHelpers.cpp` of [130].

The core emitter function, `MultiThreadEmit::emitPartitionThreadC` generates the `.c` and associated `.h` header files for given partition.

Partition State Allocation

The header file, in addition to containing function declarations, also includes a declaration of a structure type which contains all the backing storage for state elements in the DSP nodes contained within the partition. This structure is allocated on the stack by the partition's thread function on startup. Diagnostic output printed by Laminar generated application has shown that, at least in current version of pthreads on Ubuntu Linux, the stack spaces of the different threads are reasonably separated in the virtual memory space. This state structure is passed in calls to the compute function from the partition's thread function. While this method involves de-referencing a pointer to the structure, it has an important advantage to Laminar's original method of handling state: allocating global variables.

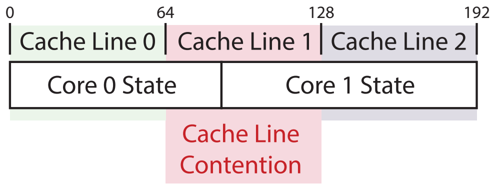


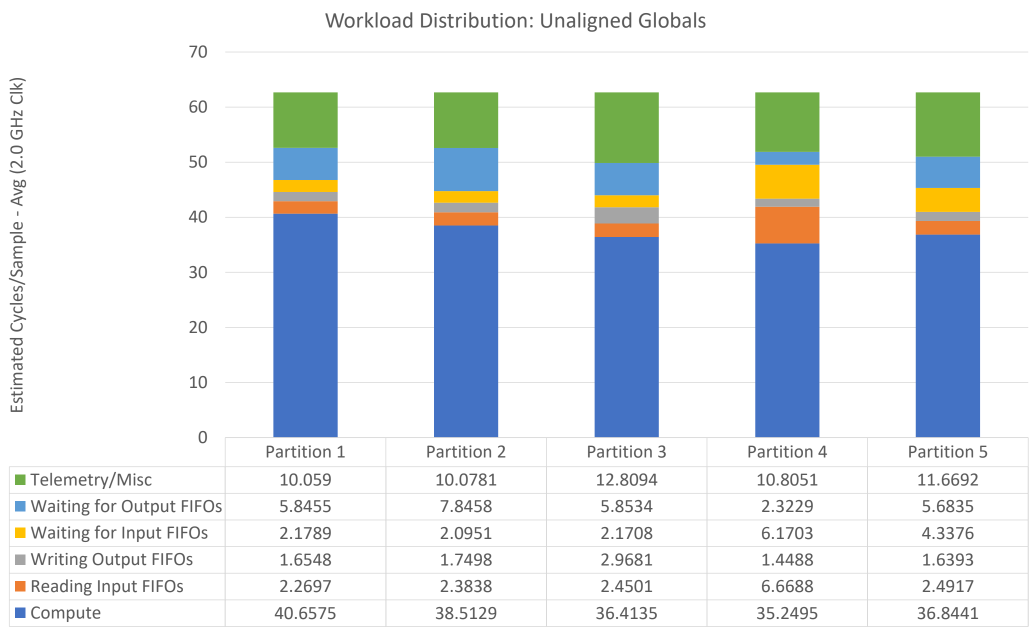
Figure 5.10: Global Cache Line Contention

From the C compiler's perspective, there was nothing special about the global variables declared for each partition. As such, the different global variables were placed close together in memory. There is nothing functionally wrong with this as each thread only accessed and modified its own variables. No synchronization should be required from a functional standpoint on these global variables. However, an important divergence between the C programming model and the hardware is that C (and the underling machine instructions) can operate on bytes while the cache system operates on cache lines. Cache lines are typically many bytes, 64 in the x86_64 CPUs used as the test platform for this project. With the compiler placing global variables close together in memory, it was not uncommon for some of the state variables of one partition to be in the same cache line as another partition. A graphical depiction of this phenomena is shown in Figure 5.10. This leads to a cache line being shared by multiple partitions. This would be OK if no variables were even written to — the line would stay in the *shared* state in the cache of each participating core. However, as these global variables represent state that is updated between samples, writing is a common task. In a MESI style cache coherency protocol, as described in section 3.2, writing to the cache line would require invalidating the entry in other caches and then acquiring exclusive access to the cache line. After writing, other caches would then need to fetch the updated cache line. Because of the cache coherency protocol, the cache line should remain uncorrupted. However, each partition contending for access to the cache line would likely experience performance degradation[16]. Due to different partitions potentially having varying amounts state in the shared cache lines, the complexity of the cache coherency system, and other factors, the performance degradation may not be experienced equally among partitions and may vary run-to-run. This behavior was observed in some simple Laminar generated designs where each partition was given the same FIR filter workload. An example from one of these experiments is shown in Figure 5.11a[17].

Cache line contention on global variables is unnecessary, given that each partition only accesses its own state variables. If state variables for each partition were isolated to their own cache lines, each partition's CPU cache could maintain exclusive access to the lines. Unless evicted from the cache, the lines holding the partition state would not need to be re-acquired. Two methods to achieve this objective include the stack allocated structure, as described above, and using alignment hints for the global variables to ensure that each variable was aligned to the start of a new cache line. Using either the global variable alignment or the locally allocated structure both reduced the workload imbalance with the structure method ultimately being selected due to the potential for poor cache utilization with each variable being allocated on a new cache line.

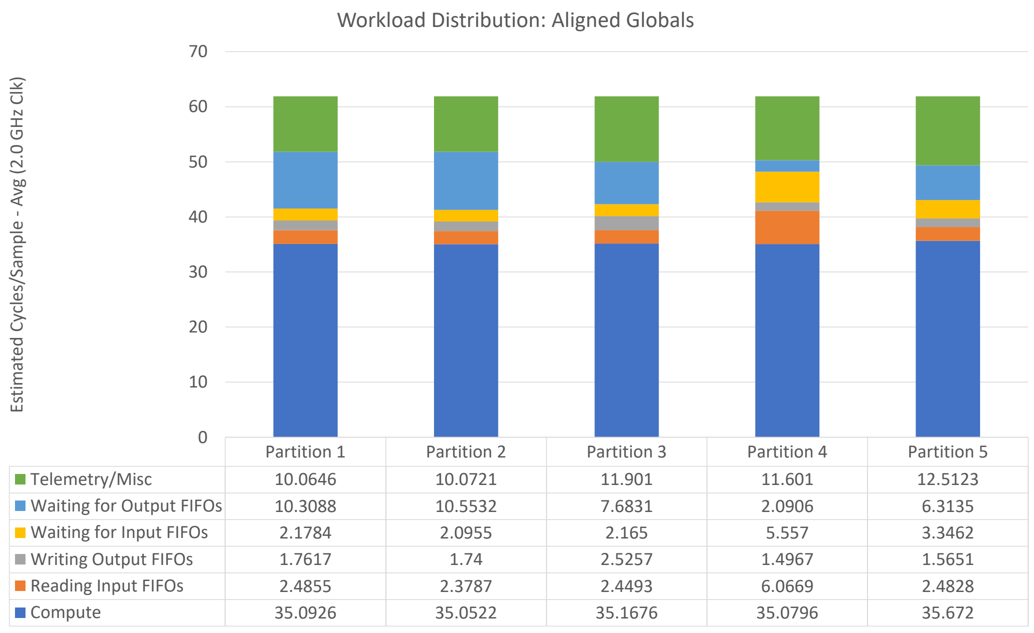
[16]Unless there is sufficient work to do that the increased latency can be hidden by other work.

[17]Note that a L3 boundary is crossed in this experiment which results in non-uniform communication costs between partitions despite having the same computational workload. When comparing state allocation techniques, the compute time is what one should be looking at most closely. Also note that some small measurement error is possible with the telemetry reporting which can result in time being shifted between groups, particularly Telemetry/Misc.



	Partition 1	Partition 2	Partition 3	Partition 4	Partition 5
🟩 Telemetry/Misc	10.059	10.0781	12.8094	10.8051	11.6692
🟦 Waiting for Output FIFOs	5.8455	7.8458	5.8534	2.3229	5.6835
🟨 Waiting for Input FIFOs	2.1789	2.0951	2.1708	6.1703	4.3376
⬜ Writing Output FIFOs	1.6548	1.7498	2.9681	1.4488	1.6393
🟧 Reading Input FIFOs	2.2697	2.3838	2.4501	6.6688	2.4917
🟦 Compute	40.6575	38.5129	36.4135	35.2495	36.8441

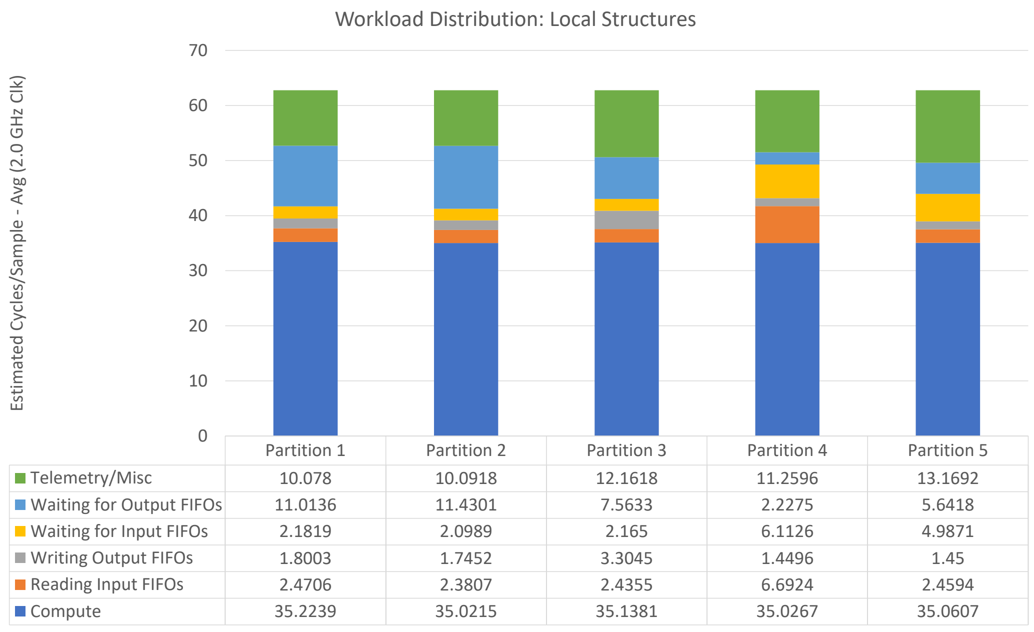
(a) Unaligned Global State



	Partition 1	Partition 2	Partition 3	Partition 4	Partition 5
🟩 Telemetry/Misc	10.0646	10.0721	11.901	11.601	12.5123
🟦 Waiting for Output FIFOs	10.3088	10.5532	7.6831	2.0906	6.3135
🟨 Waiting for Input FIFOs	2.1784	2.0955	2.165	5.557	3.3462
⬜ Writing Output FIFOs	1.7617	1.74	2.5257	1.4967	1.5651
🟧 Reading Input FIFOs	2.4855	2.3787	2.4493	6.0669	2.4828
🟦 Compute	35.0926	35.0522	35.1676	35.0796	35.672

(b) Aligned Global State

Figure 5.11: Effect of Different State Allocation Techniques



	Partition 1	Partition 2	Partition 3	Partition 4	Partition 5
🟩 Telemetry/Misc	10.078	10.0918	12.1618	11.2596	13.1692
🟦 Waiting for Output FIFOs	11.0136	11.4301	7.5633	2.2275	5.6418
🟨 Waiting for Input FIFOs	2.1819	2.0989	2.165	6.1126	4.9871
⬜ Writing Output FIFOs	1.8003	1.7452	3.3045	1.4496	1.45
🟧 Reading Input FIFOs	2.4706	2.3807	2.4355	6.6924	2.4594
🟦 Compute	35.2239	35.0215	35.1381	35.0267	35.0607

(c) Local Structure

Figure 5.11: Effect of Different State Allocation Techniques

Emitting Operators

Operator emitting occurs by running through the ordered list of nodes in the partition, arranged in scheduled order, and calling the *emit* function defined by each node's class. The implementation of this emit function is left up to the individual operator but typically involves querying for statements from its predecessor nodes, producing intermediate statements which are written to the C file, and returning an object representing the resulting C expression for the requested output port. The emit function is called for each output of the node and, if the output type is complex, for the real and imaginary components.

When a new node is emitted, its context stack is checked against that of the previously emitted node. If the node is in a new context, the context conditional logic and output variables (such as the output of a mux) are emitted before the node. If the new node is outside of the context occupied by the previously emitted node, context closing code is emitted first. This mechanism allows conditional statements like if/else and loops to be emitted automatically based on the tracked context.

5.3.4 Debugging and Documentation

Laminar provides multiple debugging and documentation aids for DSP designers and those seeking to modify the compiler. One useful feature is the ability to dump the Laminar in memory intermediate representation of the design to a `.graphml` file which can be plotted or otherwise analyzed. In addition to containing the design structure, the files contain attributes of nodes and arcs in the design. The yEd tool [133], provides a convenient way to inspect exported files by plotting them and allowing the user to map the various node and arc parameters to graphical attributes such as node and arc labels. An example of such a visualization is shown in Figure 5.12.

Currently, intermediate representations are exported, if requested, at the following stages of the compiler:

- Design Before Blocking.
- Design After Blocking.
- Design After Scheduling.

Another helpful debugging feature is the printing of information to the console during compilation. Messages printed to the console include the actions of some optimization passes, such as when a node is pruned from the design. Near the end of the process, a series of reports are printed which give information on the number and type of nodes in each partition and information about communication between partitions including how much initial state is contained in each FIFO. In addition, two "Communication Graph" files are emitted which represent information about inter-core communication in a `.graphml` format which can be easily analyzed by other tools.

For those interested in modifying or extending the functionality of the Laminar compiler, many of the functions and classes are documented using Doxygen flavored comments. HTML and LaTeXdocumentation on the Laminar implementation can be generated via the Doxygen tool. A make target, `docs`, is provided which builds the documentation. An example of the Doxygen generated HTML documentation is shown in Figure 5.13.

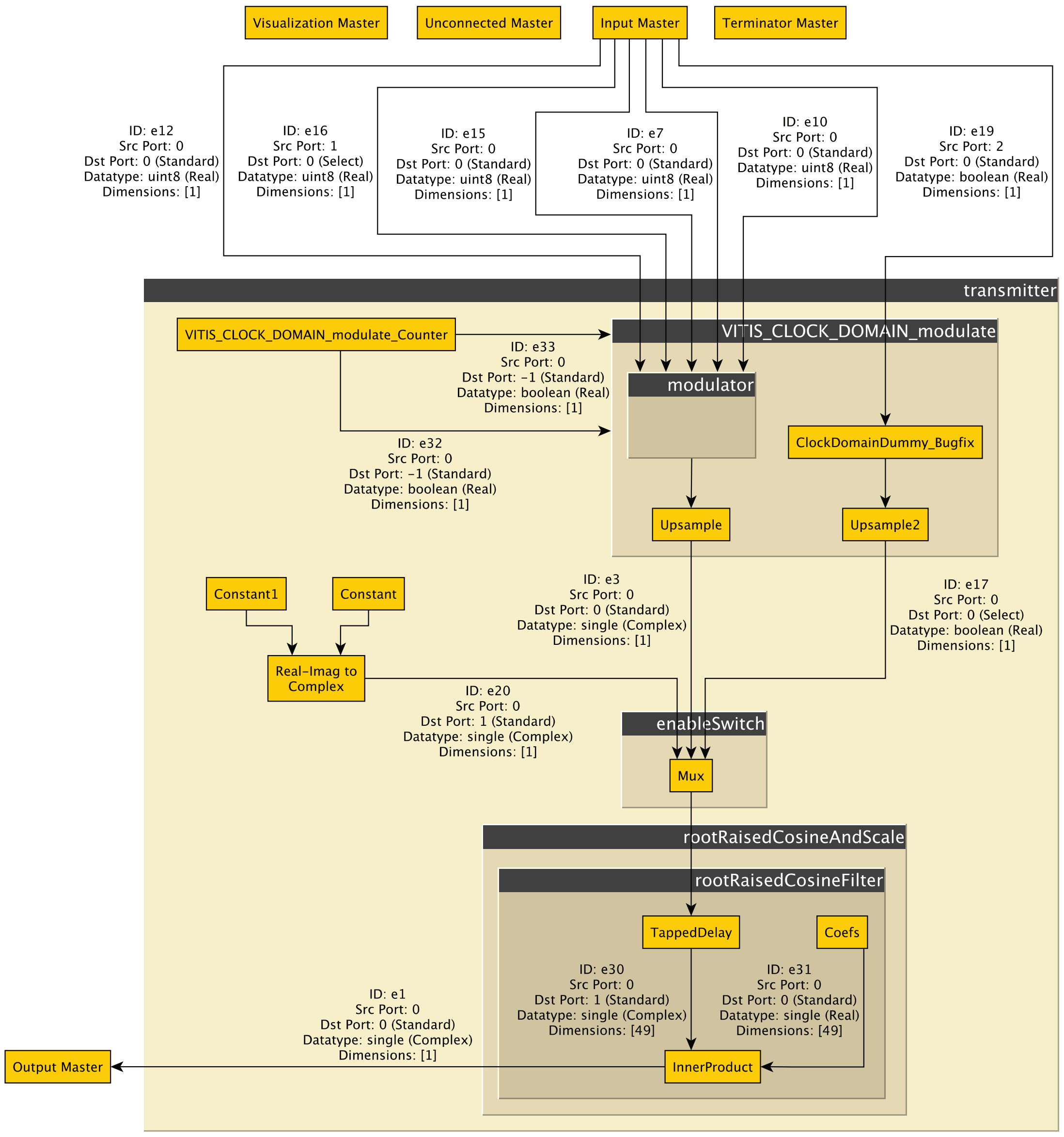
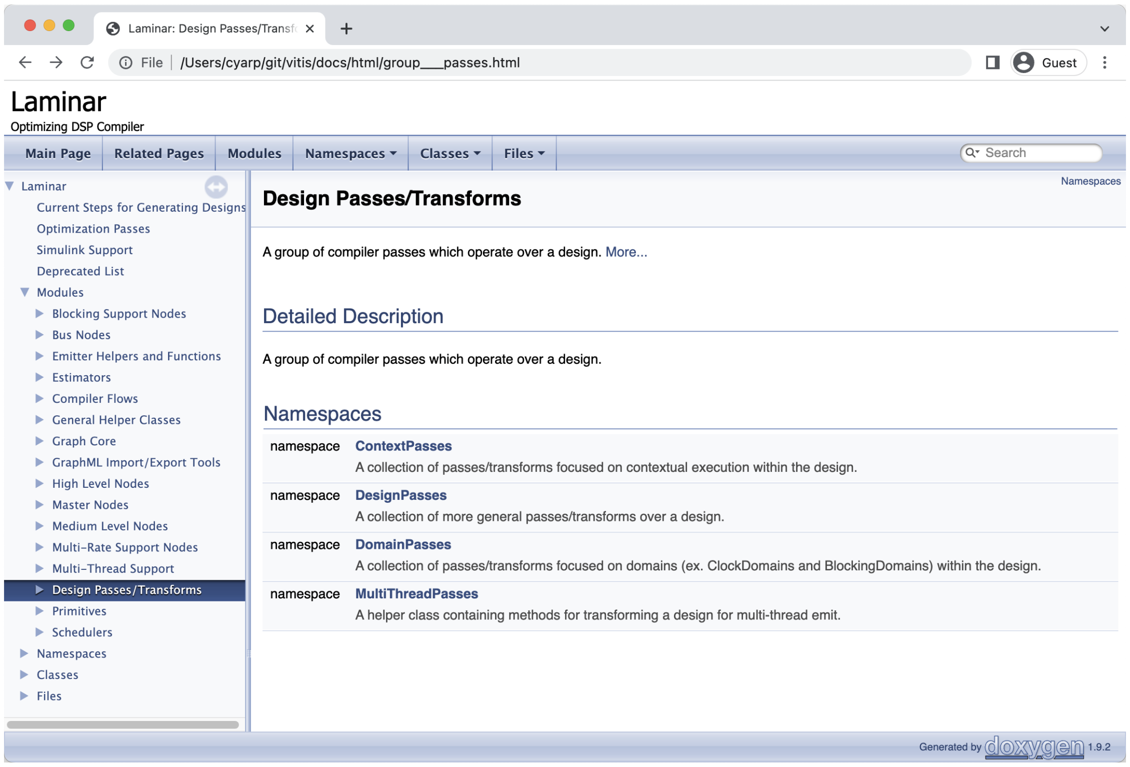


Figure 5.12: Laminar Intermediate Pre-Blocking Representation of Cyclops Tx v1.8, Plotted by yEd



Design Passes/Transforms

A group of compiler passes which operate over a design. More...

Detailed Description

A group of compiler passes which operate over a design.

Namespaces

namespace	**ContextPasses** A collection of passes/transforms focused on contextual execution within the design.
namespace	**DesignPasses** A collection of more general passes/transforms over a design.
namespace	**DomainPasses** A collection of passes/transforms focused on domains (ex. ClockDomains and BlockingDomains) within the design.
namespace	**MultiThreadPasses** A helper class containing methods for transforming a design for multi-thread emit.

Figure 5.13: Doxygen Generated Documentation

Chapter 6

Laminar Optimizations

One of the reasons for developing the Laminar compiler was to facilitate experimentation with different optimization techniques in a structured and repeatable way. Chapter 5 introduced the basics of the compiler and presented a list of passes over the design which are used in the process of converting a DSP dataflow graph to a functioning C program. This chapter will build on the discussion in chapter 5 by presenting the optimization techniques explored. This not only includes explicit optimization passes but also specialized implementations of DSP constructs.

6.1 Working with the C Compiler

There are many different paths one could have taken when producing Laminar. One option was to augment an existing compiler, such as LLVM [154], with domain specific constructs and optimization passes. While this approach would be able to leverage the structure of the existing compiler, it was determined early on that the LLVM IR was not ideally suited for representing dataflow graphs and made constructing the types of optimization passes envisioned for this project more difficult. It appears that this assessment was not without merit as the makers of LLVM have recently launched a new project, MLIR [155], which explicitly mentions better supporting dataflow and machine learning descriptions like TensorFlow.

A distinct disadvantage of integrating into an existing compiler framework is that it potentially limits the ultimate performance of the result with how well the base compiler is implemented. At the onset of this project, LLVM had been steadily improving but GCC was still considered by some to perform better with certain types of optimizations, owing in large part to its mature code base. In addition to these free, open source, compilers commercial offerings such as the Portland Group (PGI) family compilers and Intel C Compiler (icc) were available and promised more optimized results[1]. Integrating into one of these compilers

[1]Since the onset of this project, compilers have undergone some consolidation with icc transitioning to be LLVM based [156] and PGI being absorbed by NVidia [157]. However, vendor optimizing compilers, such as the new version of icc and AMD Optimizing C/C++ compiler (aocc) are still present.

would have precluded taking advantage of advances in other compilers including those in vendor provided optimizing compilers.

These factors lead to the decision to develop Laminar as a source-to-source compiler. By targeting C code, the Laminar result could be compiled with any C compiler, including those specifically tuned by vendors for their underlying CPU architecture. In a similar vein, Laminar attempts to avoid the explicit use of ISA intrinsics and instead leverages optimization passes, such as auto-vectorization, which are present in most widely used C compilers. This allows Laminar generated code to be portable across different vector extensions and ISAs. As CPUs continue to improve and new ISA extensions are added, it is expected that general-purpose compilers will be augmented to support these new capabilities. CPU vendors certainly have a strong incentive to make sure that new features they add to their parts are supported by widely used developer tools. By utilizing continued general-purpose C development, Laminar can avoid required updates to support additional CPU ISAs and new ISA extensions.

While relying on existing C compilers has the advantage of leveraging decades of compiler research and development, there are some challenges that need to be addressed. As shown in Figure 5.8, the input into the compiler can have a strong impact on the ultimate quality of results, despite aggressive C compiler optimization being enabled. In general, it has been observed in this project that the compiler generally performs best with *tight, fixed-bound, loops with easily analyzable dependencies*. This observation is in line with the general wisdom surrounding HPC software development when using advanced compiler optimizations such as auto-vectorization. Also, because general purpose compilers are generally restricted from making transformations which would result in semantically different code, DSP specific optimization that are not legal C transforms need to be performed before passing the generated code to the C compiler. Due to the desire to use existing C compilers which perform best with code emitted in a certain style, Laminar's code generation philosophy involves generating code crafted so that it can be easily analyzed and optimized by the downstream C compiler. Techniques to provide this include:

- Expressing DSP designs with vectorizable constructs, when possible.
- Providing specialized implementations of some DSP constructs.
- Encapsulating existing operators into tight loops (ex. through sub-blocking).

An interesting result of auto-generating code which is intended to be optimized by a downstream compiler is that it often looks different than what some human developers would write. For instance, intermediate arrays are typically produced by the Laminar compiler for vector/matrix operations and are assigned only once. An astute developer may point out that the declaration of many intermediate arrays is unnecessary and that a set of working arrays can be shared among operators. The problem with this approach is that it can complicate dependency analysis for the compiler. With compiler optimizations disabled (`-O0`), the human optimized version indeed performs better, as one may expect. However, in the experience of this project, with more aggressive compiler optimizations (such as `-O3` or `-Ofast`), the intermediate arrays were typically optimized away by the compiler.

6.2 Laminar Optimizations

Having discussed the general Laminar code generation approach above, the following sections describe various ways in which the Laminar compiler optimizes a given design and produces code targeted for downstream, optimizing, C compilers.

6.2.1 Context Discovery and Expansion

A core feature of Laminar, which was touched on in chapter 5, is support for conditionally executed segments of a design. These regions are viewed in Laminar as execution *contexts*. Examples of different types of contexts implemented in Laminar include[2]:

- Muxs.
- Enabled Subsystems.
- Downsample Clock Domains.
- Blocking/Sub-Blocking Domains.

Contexts for each of these node types are eventually encapsulated within container subsystems. In the current version of the scheduler, as described in section 5.3, the nodes within the same context are scheduled together. The Laminar emitter keeps track of the context stack of the previously emitted node and the current node to emit, automatically inserting the context conditional logic as appropriate. For some contexts, such as enabled subsystems, the extent of the context is known at the start of compilation from the design hierarchy. For enabled subsystems, all nodes contained within the subsystem are automatically considered to reside within the conditional context. Other nodes, such as multiplexers, require their context to be discovered.

Mux Discovery

A multiplexer in a DSP or hardware design acts as a switch which passes the value of one of its inputs, based on the value of the selector port, to its output. Without any clock or power gating logic, the logic for the different inputs of the multiplexer would all be active with the final value being selected from one of the inputs. There is power inefficiency with this construction as only the logic at the input port that is ultimately selected needs to run. However, there are downsides to clock or power gating the logic as it adds complexity to the design. Synchronous digital logic timing analysis also needs to conservatively take

[2]Internally, these different contexts inherit from the `ContextRoot` class.

the worst-case scenario through the multiplexer into account when computing the allowable clock rate, even if the worst-case multiplexer input is rarely selected.

This contrasts with typical software implementations where the logic for each mux input could be contained within if/else blocks or switch statements. Because instructions are executed with some degree of serialization on limited execution resources, avoiding the execution of instructions whose values would go unused could potentially improve the overall execution time[3]. In cases where the complexity of calculating different inputs into the multiplexer is vastly different and the lower-complexity inputs are commonly taken, there can be a substantial average performance improvement by conditionally executing the logic to compute each input.

Because branching and looping is necessary in most programs, CPU implementations have included optimizations to try to improve their execution. For example, many modern CPUs (including the ones used in this project) implement branch prediction logic. By observing previous branch behavior, the branch predictor attempts to guess the result of an upcoming branch before it has been determined. If successful, the branch predictor keeps the CPU pipeline filled and avoid stalls. If incorrect, the work computed along the predicted branch is discarded and execution starts again along the correct branch. This results in a *branch misprediction penalty*, which can be quite large. Branch prediction has been extensively researched with multiple prediction schemes implemented with varying degrees of complexity. In general, branches that commonly take one path are often correctly predicted by the branch predictor after initial training. Cases involving simple and repeated patterns taken by branches, such as in fixed bound loops, are also of interest to branch prediction researchers. However, most CPU vendors do not provide all details of their branch prediction logic. For some information on the branch predictor in the AMD CPUs used by this project, see [17]. For more information on branch prediction in general, see [16], [105].

If we assume that the branch prediction logic is correct in most cases after startup, there is little cost to conditionally executing code. Given the potential workload savings, it makes sense to maximize the amount of logic that can be placed in the conditional blocks leading to multiplexers. This is accomplished in a step referred to in Laminar as *Discoverer and Mark Contexts*. In this step, Laminar traces back from each of the input ports into the multiplexer, stopping at state elements, enabled subsystem boundaries, clock domain boundaries, sub-blocking base length boundaries, and explicit context expansion barrier nodes which can be inserted by the DSP designer. It also stops when a node in another partition is discovered. Arcs which are traced back are marked. If a node is encountered with all its output arcs marked, the trace-back recurses back to the inputs of that node. Nodes which have been recursed on are only used to derive the input of the multiplexer, possibly via additional operations. If their output was depended on by another node outside of computing the mux input, the corresponding output arc would not have been marked. This would have prevented

[3]Even with superscalar out-of-order CPUs, executing unneeded instructions could take up slots in execution units which could otherwise be used for required operations, potentially creating a bottleneck for that resource. Though they are often pipelined, execution units in superscalar CPUs typically serially execute instructions from their scheduling queue.

the recursive trace-back from occurring on the node. This is true if the dependency fanout occurred later in the logic chain. Ultimately, the context discovered for a given mux input is the set of combinational logic (contained within a given partition, clock domain, and sub-blocking base) whose outputs are unused outside of computing the value of the multiplexer input. State elements are excluded from the context as state updates occur regardless of which input is selected. An example of the mux context discovery procedure is shown in Figure 6.1.

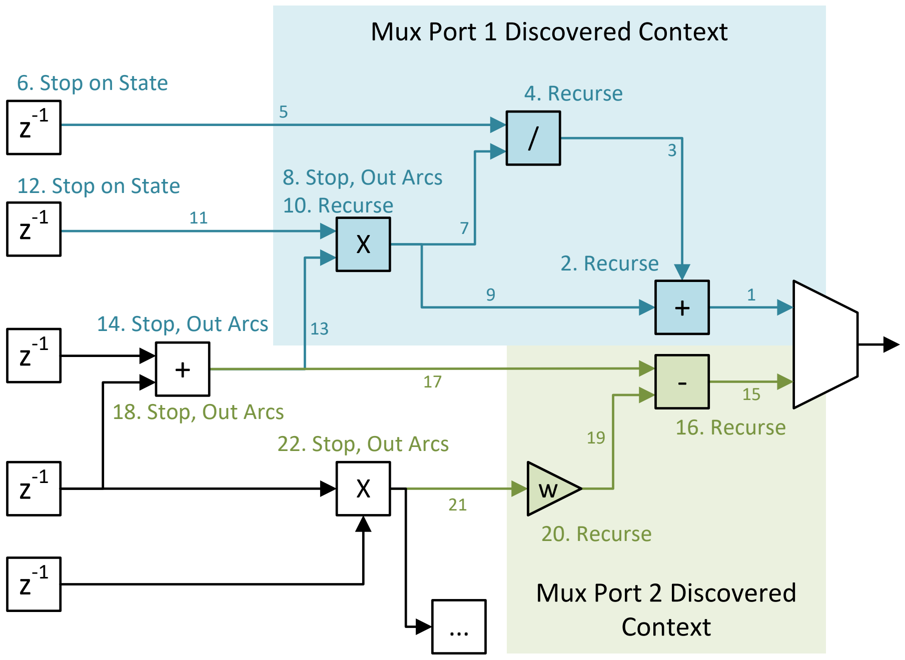


Figure 6.1: Example of Mux Context Discovery Procedure

Enabled Subsystem Expansion

Enabled subsystems are analogous to clock-gated and power-gated sections in hardware designs. Like clock-gated design segments, state elements in enabled subsystems do not update unless enabled. The outputs of enabled subsystems exhibit latch-like logic where the output is passed through if the system is enabled, and the previously computed value is passed if the system is disabled. Because the outputs exhibit this latching behavior and state updates within the subsystem only occur if enabled, the logic within enabled subsystems only needs to be executed in cases where the enabled signal is true. This allows enabled subsystems to potentially take advantage of the same workload reduction advantages as described above for multiplexer context discovery.

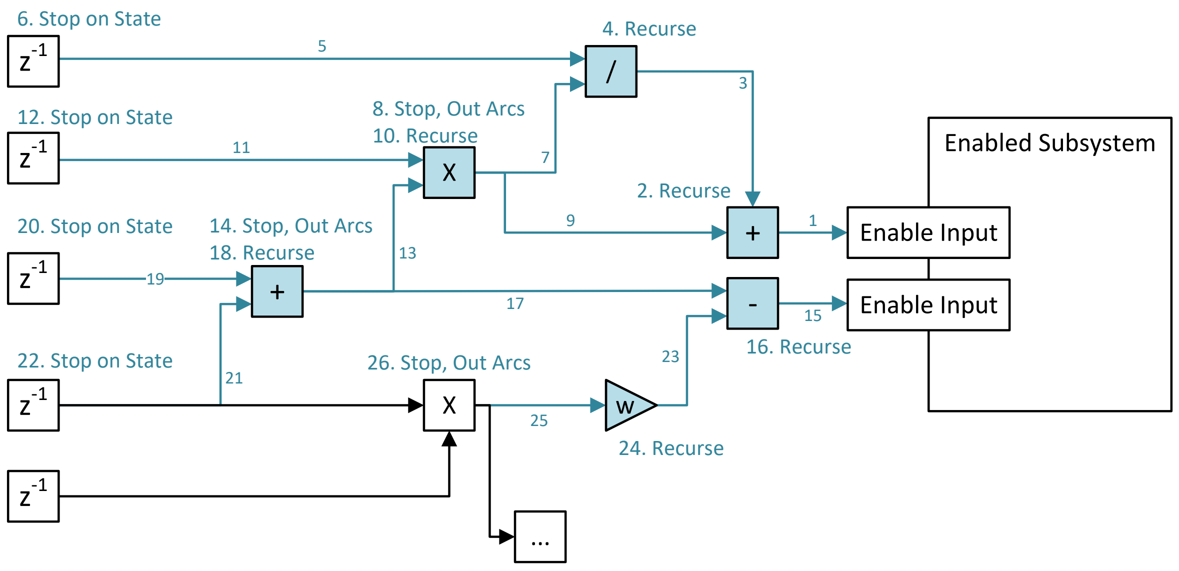
Enabled subsystems, in the Laminar model, are comprised of a specialized enabled subsystem container within which nodes can reside. At the boundaries of the enabled subsystem are *Enable Input* and *Enable Output* nodes. Discovery of enabled subsystem contexts is relatively easy as each node within the enabled subsystem is contained within the context. This includes the enable input and output nodes. In the current version of the Laminar with nodes in contexts scheduled together, the emitted C code for the operators contained within the enabled subsystem are wrapped in a single `if` block. The latching logic is achieved by the enable output nodes, which are also contained in the enabled context, passing their state variable as the output to downstream nodes. If the enabled subsystem is run, the enable output state variables are updated before any dependent nodes are executed. State update nodes for delays inside the enabled subsystem are also contained within the enabled context and are not executed if the subsystem is disabled.

On the input side, the idea behind enabled subsystem expansion is that combinational logic which is only depended on by nodes within the enabled subsystem does not need to execute unless the subsystem is running. In effect, the logic could be pulled into the enabled subsystem. This could potentially reduce the workload on the CPU on average if the enabled subsystem spends considerable time disabled. On the input side, enabled subsystem expansion uses the same trace-back techniques as mux expansion except that the marked arcs are maintained across the different enable input nodes. Once identified, nodes in the expanded context are moved within the enabled subsystem and enable input nodes are created at the new boundary points. A demonstration of this process is shown in Figure 6.2.

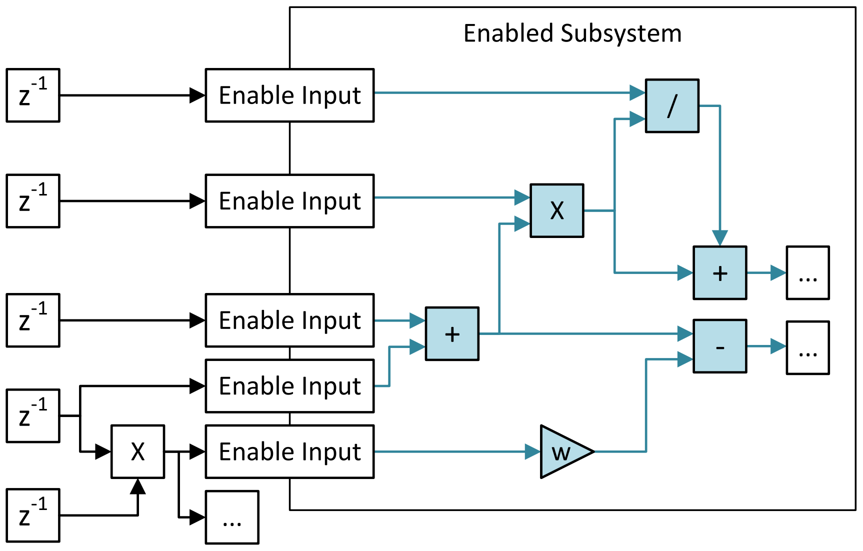
On the output side, there is also the potential that some combinational logic can be moved within the enabled subsystem. In general, this is possible if all the inputs for the combinational logic come from the enabled subsystem. To determine this, a trace-forward version of the same approach used for the input side of the enabled subsystem is used. This trace-forward version recurses only if all input ports to a node have been marked. Tracing stops at state nodes and certain boundary nodes, just like the trace-back variant. After the extended context is discovered, nodes within it are moved into the enabled subsystem and enable outputs are created at the new boundary points.

When Contextual Execution may be Undesirable

While it seems to make sense that reducing the average workload by conditionally executing code can provide performance improvements, there are some scenarios where it hurts performance. One such case is when the condition on which logic is executed is hard to predict. This will result in the branch predictor potentially incorrectly predicting the branch often, incurring regular misprediction penalties. Depending on the magnitude of the misprediction penalty and the complexity of the branch predictor, better performance may be achieved by removing the branch if possible. For example, if the branch was used to mask a certain value, multiplying by zero can have a similar effect in certain situations. Even though more logic may be executed in this case, so long as it is less than the misprediction penalty, there should be improvement.



(a) Extended Context Discovery



(b) After Node Relocation and Enable Input Move

Figure 6.2: Enabled Subsystem Context Expansion (Enable Line(s) Not Shown)

Another case where conditional logic may be undesirable is when vectorization is available. If vectoring across samples, it is possible that a given operator will be enabled for some sample and disabled for others. Vectorizing this type of code typically requires masked vector operations where a mask must first be computed which is then used to limit the vector operation to only impact specific elements. If the branching logic includes an `else` clause, the inverted mask would then be required for operations in the else block. Both branching paths would need to be executed if the vector contained a mixture of enabled and disabled elements. It is possible on some platforms that, if all elements take the same path, the execution of the alternate path is avoided. However, on conventional CPUs, this would likely require branching on the value of the mask, introducing the potential for branch misprediction penalties. Nested branches further complicate matters since each composite path from the two branches would potentially need to be executed. This can quickly negate any benefit from vectorizing. As such, it is sometimes desirable to avoid the use of branches or enabled subsystems in the design when it is expected that vectorization could possibly occur.

6.2.2 Vectorizable DSP Constructs

SIMD units are an exceptionally important resource for accelerating designs as they have the potential to accomplish many computations simultaneously. While it can be challenging for programmers to extract SIMD from their applications, DSP designs are often filled with easily vectorizable constructs. One exceptionally common DSP block, the FIR filter, is composed of a shift register and dot product with one constant operand (as shown in Figure 4.3). The dot product is, itself, a vector operation. Laminar supports emitting dot products in the form of a fixed bound loop with an outer accumulator. Inside the loop, the coefficients and inputs are multiplied and the result accumulated. This operation is so common that hardware exists within many x86_64 CPUs to accelerate the operation. The Fused Multiply-Add (FMA) vector unit is capable of element wise multiplying two vectors and adding the result to a third vector [17], [103]. By splitting the dot product into multiple accumulators, each of which being an element in a vector register, a chain of FMAs can be used to generate partial results which can then be reduced by summation into the final result[4]. This optimization of the loop is possible with more aggressive C compiler optimization levels such as `-Ofast`.

Several other operators are implemented in Laminar to support element-wise vector or matrix operations, such as element-wise sum, product, mux. Additional operators can be converted to support element wise operation by using helper functions within Laminar and the design pattern used in the sum operator to generate inner loops for element-wise operations.

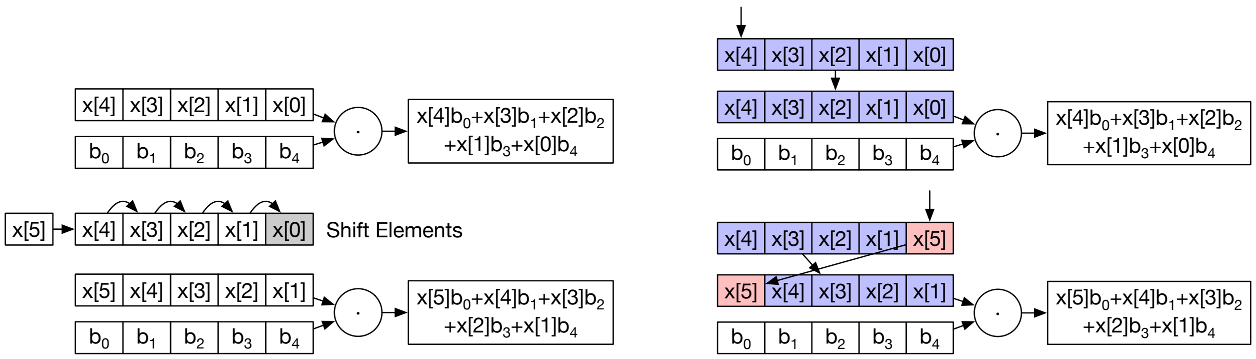
[4]Due to floating point operations not being strictly associative, the modification of the dot product into smaller segments and then reducing the result may lead to different numeric results. In a radio operating with constrained dynamic range on the input and reasonable algorithms, it is not anticipated that this causes many problems. Since radios operate in noisy environments, minor numeric differences often have a limited effect on the ultimately decoded signal.

6.2.3 Tapped Delay and Circular Buffer Implementations

While it would appear that FIR filters are easily vectorizable because the dot product vector operation performs the computation, early implementations performed far below expectations. The problem is that, while the dot product operation itself is clearly vectorizable, the shift register feeding the dot product can inhibit effective vectorization or contribute significantly to the workload of the block.

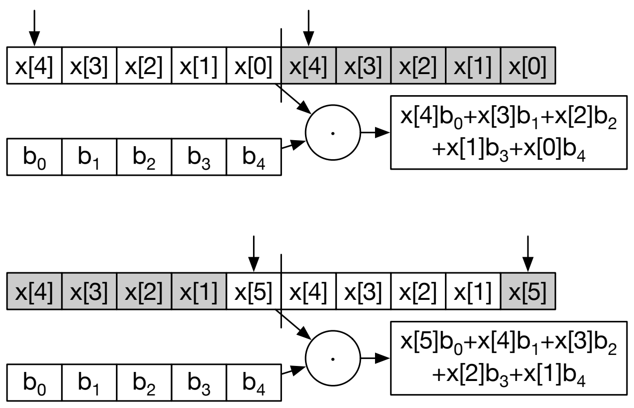
The simplest implementation of a tapped delay in software mirrors a shift register in hardware. The elements in the tapped delay are stored in an array with each element being shifted over a position each iteration. This involves a significant amount of data movement which the compiler cannot easily overlap with another iteration of the FIR filter, due to the data dependencies. A compelling alternative is a circular buffer where the head of the buffer is shifted through the array on each iteration. The new entry overrides the oldest entry in the array. Because the circular buffer is of finite length, the head pointer needs to wrap from the last element in the array back to the first. Care must be taken when accessing elements in the buffer to properly handle wrap-around logic. While the address arithmetic to handle the wraparound is simple, especially for buffer sizes which are powers of 2, the wraparound interferes with the vectorization of the inner product. If the buffer is sized to the tapped delay, the values can be easily read into vector registers only for one position of the head pointer. In all other cases, the discontinuity would need to be specially handled. Disassembling compiled binaries revealed that the C compilers were unable to come up with an elegant solution to the problem on their own with one solution providing a vectorized implementation of the dot product for cases when no wraparound was encountered with the tapped delay but reverting to scalar instructions in the other cases. One attempt to address this problem was to perform an intermediate copy of the circular buffer into a new buffer using either simple for loops or calls to `memcpy`. This did improve performance relative to using the circular buffer addressing logic in the `for` loop implementing the dot product. By providing unit stride access to the elements in the shift register, the compiler was able to auto-vectorize the dot product, as expected. However, this implementation is still inefficient, requiring significant data movement.

From the earlier experiments, it was clear that limiting the amount of memory movement as well as providing stride one access to the elements in the tapped delay were important to achieving high performance. The shift register and re-buffered circular buffer provide the stride one access but require extensive data movement. The classic circular buffer significantly reduces data movement but gives up unit stride access. An alternative scheme was implemented in Laminar which provides both unit stride access and reduced data movement at the cost of a larger memory footprint. In this implementation, the tapped delay buffer is oversized. In the example shown in Figure 6.3c, the buffer is double the required length. The buffer is logically split in half with each half being a replica of the other. When a new element arrives, it is written into both halves of the buffer. With this implementation, a stride one version of the buffer can always be read. In the case where the head pointer is not at the first element in the array, the tail end of the returned segment includes elements



x[4] x[3] x[2] x[1] x[0]
b0 b1 b2 b3 b4
x[4]b0+x[3]b1+x[2]b2 +x[1]b3+x[0]b4
x[5]
Shift Elements
x[5]b0+x[4]b1+x[3]b2 +x[2]b3+x[1]b4

(a) Shift Register (b) Circular Buffer with Re-Buffering



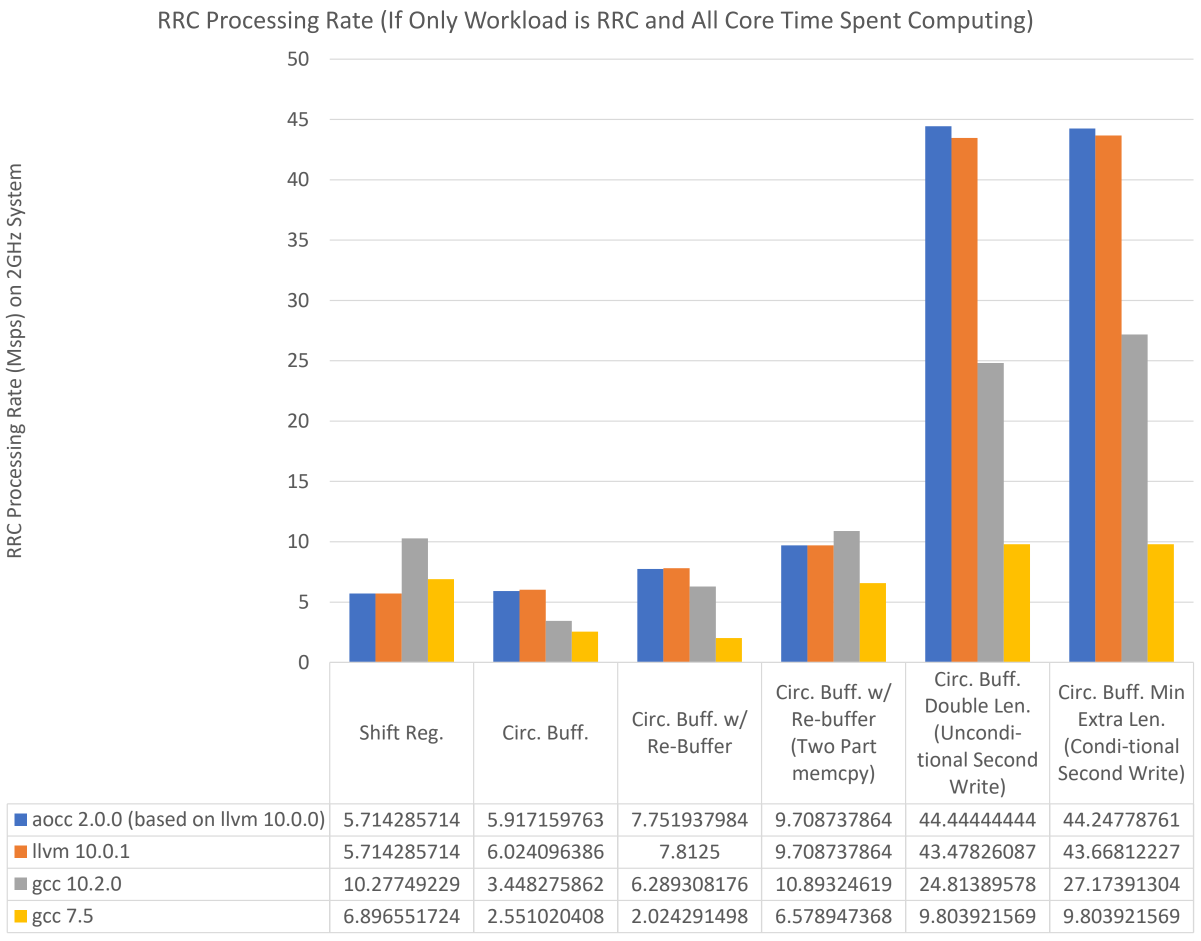
x[4] x[3] x[2] x[1] x[0] x[4] x[3] x[2] x[1] x[0]
b0 b1 b2 b3 b4
x[4]b0+x[3]b1+x[2]b2 +x[1]b3+x[0]b4
x[4] x[3] x[2] x[1] x[5] x[4] x[3] x[2] x[1] x[5]
x[5]b0+x[4]b1+x[3]b2 +x[2]b3+x[1]b4

(c) Over-provisioned Circular Buffer with Replicated Writes

Figure 6.3: FIR Filter with Different Tapped Delay Implementations

from the replica. This version with double the buffer length allows the replica write to occur unconditionally, avoiding a branch. However, if memory footprint is at a premium, the buffer can be sized smaller with a conditional replica write. The size reduction is based on how much delay the buffer is providing and how many elements are returned at a time.

The performance of each of these methods using different compilers was tested against the Root Raised Cosine (RRC) FIR filter used in Cyclops with the results shown in Figure 6.4. The specialized implementation of the tapped delay with over-provisioned buffer sizes and replicated writes performed the best out of all methods as was expected, given its ability to always provide unit stride access to tapped delay elements. For LLVM compilers, there was a speedup of over 7.5x relative to the shift register implementation. The circular buffer implementation with an intermediate copy to a stride one array improved performance over the traditional implementation, particularly when using LLVM based compilers. Interestingly, the LLVM based compilers outperformed GCC in almost all cases except for the shift register implementation. The more recent version of GCC also slightly outperformed the



	Shift Reg.	Circ. Buff.	Circ. Buff. w/ Re-Buffer	Circ. Buff. w/ Re-buffer (Two Part memcpy)	Circ. Buff. Double Len. (Uncondi-tional Second Write)	Circ. Buff. Min Extra Len. (Condi-tional Second Write)
🟦 aocc 2.0.0 (based on llvm 10.0.0)	5.714285714	5.917159763	7.751937984	9.708737864	44.44444444	44.24778761
🟧 llvm 10.0.1	5.714285714	6.024096386	7.8125	9.708737864	43.47826087	43.66812227
⬜ gcc 10.2.0	10.27749229	3.448275862	6.289308176	10.89324619	24.81389578	27.17391304
🟨 gcc 7.5	6.896551724	2.551020408	2.024291498	6.578947368	9.803921569	9.803921569

Figure 6.4: Computation Rate for RRC FIR Filter with Different Shift Register Implementations and C Compilers

LLVM based compilers for the circular buffer implementation with intermediate copy using two calls to `memcpy`. Currently, the Laminar supports the shift register, traditional circular buffer, and over-provisioned circular buffer with replica write implementations of the tapped delay block.

Aside on How the Compiler Affects the Quality of Results

An interesting result of this experiment, outside of finding the best tapped delay representation, was observing how much different compilers, including different version of the same compiler, effected the quality of results. As was mentioned in subsection 5.3.2, general purpose C compilers face very hard optimization problems, some of which are NP-hard. As such, compilers are often constrained to heuristic solutions that are still being improved to this day. Additionally, new CPU releases can introduce new features or change costs that compilers use to drive heuristics. Older compiler versions can lack tuning for newly released CPUs, effecting their resulting performance. The different versions of GCC tested were released relatively close together with version 7.5 released on Nov 14, 2019, and version 10.2 released on July 23, 2020 [158]. Even taking version 7.1 into account, which was released May 2, 2017 [158], there is a little over three-year difference between major versions. Despite the short time difference between releases, the performance of the executable produced by version 10.2 was observed to be over 2x faster than the executable produced by version 7.5. This difference underscores that optimized C compilation is not a solved problem and is still being developed today, over three decades since the initial release of gcc (May 23, 1987 [158]). Because the default compiler is often frozen for Linux distributions, such as Ubuntu LTS[5], users may need to specifically request newer compiler versions from their package management system or install them from source. While a hassle, use of an up-to-date compiler can be an important factor in attaining good performance, particularly when using newly released CPUs.

6.2.4 Blocking and Sub-Blocking

As discussed in section 5.2, transactions between cores/threads already typically transact in blocks of samples to amortize fixed costs associated with FIFO transactions. In the initial implementation of blocking, each thread looped over the samples received from the input FIFOs. Disassembly of the compiled binaries revealed that the C compiler was often unable to automatically vectorize these, often large and complex, loops. As discussed in section 6.1, this is not entirely unexpected as the compiler optimizations typically perform best with small, fixed bound, loops with easy dependency analysis. In addition to the loops containing potentially complex structures, DSP state updates (as discussed in subsection 5.3.1) can present an impediment to auto-vectorization by the C compiler.

One method to address this is to break the incoming block of samples into sub-blocks which are operated on instead of individual samples. In the dataflow graph model, this can be represented by adding (or extending) the dimension of the data types in the sub-graph to be sub-blocked. An example of the general concept when sub-blocking by a factor of 4 is shown in Figure 6.5. Sub-blocking has proven successful in other fields, such as HPC [162][6], where proper sub-blocking allows some algorithms to present much more cache friendly memory access behavior. Sub-blocking can also aid in auto-vectorization or auto-unrolling by presenting tight fixed-bound loops for some operators, particularly ones without

[5]For example, the clang package for Ubuntu 18.04 LTS aliases to clang-6 (version 6.0) [159] while the same package for Ubuntu 20.04 LTS aliases to clang-10 (version 10.0) [160]. Clang version 10.0 is available on Ubuntu 18.04 LTS by explictally installing the clang-10 package [161].

[6]In the HPC context, this technique is often simply referred to as blocking with the algorithm operating on the input data in blocks. The input in the HPC case is equivalent to the input block to the Laminar compute function. Since the data was already blocked to amortize FIFO costs, operating on segments of the block is referred to as sub-blocking in the Laminar contexts to distinguish the two levels of blocking.

state where each element is independent. In addition to vectorizing over the samples in the block, some operators can achieve additional performance and efficiency with specialized blocked implementations. Laminar supports both specialized blocking implementations and automatic encapsulation of operators in fixed bound loops.

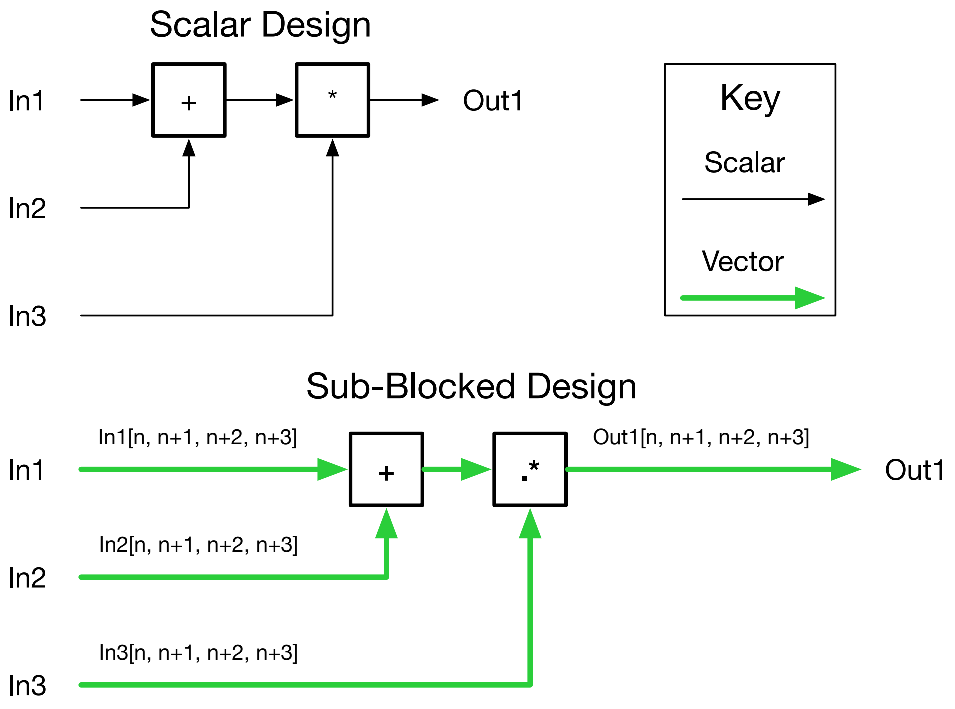


Figure 6.5: Sub-Blocking Concept on a Scalar Design (.* = Element-wise Multiply)

Improvements for Already Vectorized Constructs

Benefits of sub-blocking are not isolated to scalar ops. One of the key features of sub-blocking individual operators, particularly stateless ones, is that it increases the amount of independent work locally available. For scalar operators, this can lead to vectorization, as discussed earlier. For vector and scalar ops, the local availability of independent work can be used to keep CPU execution units occupied in the face of non-trivial operator latency.

A hypothetical execution schedule for the real component of the inner product of a 129-tap RRC FIR filter on the AMD Ryzen 3970X is shown in Table 6.1. T0-7 represents the FMA operations for taps 0 through 7 (inclusive) of the filter. The results of that computation are stored in vector accumulator A0 (as noted in the column header of the FMA schedule). This schedule assumes that memory operands for instructions are fetched before instruction issue but does include the latency of the different operations as described in [17]. Since the RRC filter is a complex filter with real coefficients, the real and imaginary components can be computed separately. While only the real component is shown here, one can assume the imaginary component is running on the other FMA and vector/floating-point sum unit. Because both the real and imaginary components need to be computed, and there are only two FMA and two vector/floating-point sum units, all relevant vector units are involved in the computation of the single filter.

One thing to note is that, while the FMA unit is fully pipelined and able to accept a new operation in each cycle, there are five cycles of latency for an individual FMA operation to complete. To keep the FMA unit busy, this requires five different vector accumulators. With single precision floating point and 256-bit wide vector registers, this effectively means there are 40 single precision accumulators which eventually need to be reduced to a single sum. This presents a problem for smaller FIR filters with less than 40 taps as they would be unable to keep the FMA pipeline full. There is a shorter three-cycle latency for vector and scalar floating-point sums. Despite the decreased latency, the reduction stage of the dot product requires more cycles from the time it begins to the time the final result is available. This is primarily due to cycles lost while waiting for dependent operations to finish. This is particularly acute in the final stages of the reduction where there is a chain of dependent instructions that are latency bound. Also significant is the fact that the FMA unit is generally not busy while the sum unit is working and vice versa. Ideally, multiple dot products would be available to compute so that the FMA unit could begin operating on the next dot product while the previous one was going through the reduction stage.

While branch prediction can potentially provide some additional operations for the FMA units, there are limitations including the size of the partition loop as well as how many outstanding predictions are allowed with tight loops. State updates for each outer-loop iteration can also create false dependencies which can inhibit compiler optimizations, such as unrolling, and prediction logic. To put this concretely, a partition containing an FIR filter with no sub-blocking contains an outer loop over the samples in the block in which the result of the FIR is computed. Within the loop, after the dot product logic has completed, the state of the tapped delay is updated by moving the head pointer(s). On the next iteration, the next sample is written into the buffer, overriding any previous value present. If the tapped delay buffer (even if using the specialized circular buffer implementation with overallocation and replicated writes) is sized such that the overwritten value is needed by the previous dot product operation, the memory write will be prevented from occurring until all instructions accessing it have finished[7]. A similar dependency hazard exists with the head pointer, whose value is relied on to properly index within the array. Inspecting the disassembled binary produced from the Laminar generated code shown in Listing 6.1 revealed that the compiler

[7]Speculative writes to memory are generally not permitted as they effect the state of the system in a way that can be difficult to reverse if the speculation proves incorrect. However, memory write instructions can potentially be executed speculatively if the CPU contains a write buffer that can hold writes until it can be determined that the write can be committed to memory. While isolated to the store buffer, it is easier to revert changes as the old value is still stored in the cache. However, dependency tracking is required to ensure that any instruction that uses the speculated write value can be reverted if the speculative write is later determined to be invalid.

Clk	FMA0 (5 Vector Accumulators + 1 Scalar for Partial Results)						Sum0	
	A0/Misc.	A1	A2	A3	A4	M0		
0	T0-7							
1		T8-15						
2			T16-23					
3				T24-31				
4					T32-39			
5	🟩 T40-47							
6		🟩 T48-55						
7			🟩 T56-63					
8				🟩 T64-71				
9					🟩 T72-79			
10	🟩 T80-87							
11		🟩 T88-95						
12			🟩 T96-103					
13				🟩 T104-111				
14					🟩 T112-119			
15	🟩 T120-127							
16		🟩				T128		
17			🟩				A1+A2	
18				🟩				
19					🟩	🟩		A3+A4
20	🟩						🟩 A0+(A1+A2)	
21								
22								🟩
23							🟩 (A0+A1+A2) +(A3+A4)	
24								
25								
26	vextract						🟩	
27	🟩						Extract+Low	
28								
29								
30	vpermilpd						🟩	
31	🟩						vaddps	
32								
33								
34	vmovshdup						🟩	
35	🟩						vaddss	
36								
37								
38							🟩 vaddss M0	
39								
40								
41							🟩	

Table 6.1: Hand-written Schedule for Real Component of 129 Tap RRC FIR Filter - Assumes Memory Operands Fetched Before Instruction Issue - Green = Operation Complete

did not unroll the outer loop but did vectorize the inner loop with FMA instructions and reductions. The hazards can be easily avoided by allocating additional buffer space so that all sample values required across the different dot products are present and the header index used to compute addresses can remain constant during in a single iteration of the outer loop. This is automatically accomplished by sub-blocking.

Sub-blocking provides a reliable method to increase the amount of parallel work available to fill the holes in the execution unit pipelines. A sub-blocked dot product will be presented with multiple, independent (from the perspective of the dot product operation), input vectors that can be computed in parallel. Depending on the length of the dot product, larger degrees of sub-blocking may be required to fill the gaps left open by the latency of the FMA operations. The addition of multiple independent dot products can also simplify the reduction step by allowing each separate dot product to use fewer vector accumulator registers. For the case of more than 20 independent dot products, it is possible to completely avoid the reduction step with each dot product occupying a single element in a vector register. The five (or more) vector accumulators containing the individual accumulator for the twenty (or more) different dot products allow the five-cycle latency of the FMA unit to be hidden. An example of this occurring with the Cyclops RRC filter sub-blocked by a factor of 24 is shown in Appendix D.

6.2.5 Blocking/Sub-Blocking and Loops

One challenge that exists with both blocking and sub-blocking involves loops. As was stated in chapter 5, combinational loops are not allowed in a design being passed to Laminar. However, loops are legal so long as they contain a delay which breaks the dependency chain when processing individual samples. The same idea is true for blocking except that the units change — instead of requiring a single delay to exist, loops which cross partition boundaries need at least a block's worth of state to avoid deadlock. In fact, at least this much state must exist at one of the FIFOs participating in the loop so that it can be converted to an initial value in the FIFO. Since FIFOs transact in blocks rather than individual samples, this initial state becomes a token in a Kahn Process Network (KPN) [18] or synchronous dataflow [11] view of the design[8]. With only one block of state, at most one partition in the loop can be active at a time. Ideally, multiple blocks of data would be introduced to allow multiple partitions to operate simultaneously.

Sub-blocking has a similar problem to the overall blocking case. Loops which have a delay less than the sub-blocking length cannot be broken apart when sub-blocking — there is a dependency within the sub-block. However, loops with delays equal to or larger than the sub-blocking factor can be broken apart. This is because any node depending on the value of the delay block is dependent on the input to the delay from one or more previously computed blocks. This breaks the dependency chain in the loop at the sub-block level,

[8]The KPN model has theoretically unlimited buffers while the Laminar generated design has fixed length buffers. However, the concept of tokens in loops still holds for Laminar.

```c
void rx_demo_partition1_compute(Partition1_state_t *stateStruct, const float InputSamples_re[120], const float InputSamples_im[120], float OutputSamples_re[120], float OutputSamples_im[120])
{
  for (uint8_t blockingIdx = 0; blockingIdx < 120; blockingIdx++)
  {
    float BlockingInput_re = InputSamples_re[blockingIdx];
    float BlockingInput_im = InputSamples_im[blockingIdx];

    //---- Calculate TappedDelay ----
    memcpy(stateStruct->TappedDelay_state_re + stateStruct->TappedDelay_headIdx, &BlockingInput_re, sizeof(float) * 1);
    memcpy(stateStruct->TappedDelay_state_re + stateStruct->TappedDelay_headIdx - 64, &BlockingInput_re, sizeof(float) * 1);
    memcpy(stateStruct->TappedDelay_state_im + stateStruct->TappedDelay_headIdx, &BlockingInput_im, sizeof(float) * 1);
    memcpy(stateStruct->TappedDelay_state_im + stateStruct->TappedDelay_headIdx - 64, &BlockingInput_im, sizeof(float) * 1);

    //---- Calculate InnerProduct ----
    float InnerProduct_Accum_re = ((float)0);
    float InnerProduct_Accum_im = ((float)0);
    for (unsigned long indDim0 = 0; indDim0 < 49; indDim0++)
    {
      InnerProduct_Accum_re += (((float)(Coefs_re[indDim0])) * (((float)((stateStruct->TappedDelay_state_re + stateStruct->TappedDelay_headIdx - 48)[indDim0]))));
      InnerProduct_Accum_im += (((float)(Coefs_re[indDim0])) * (((float)((stateStruct->TappedDelay_state_im + stateStruct->TappedDelay_headIdx - 48)[indDim0]))));
    }

    //---- State Update for TappedDelay~~~~
    stateStruct->TappedDelay_headIdx = ((stateStruct->TappedDelay_headIdx + 1) % 64) + 64;

    (OutputSamples_re[blockingIdx]) = InnerProduct_Accum_re;
    (OutputSamples_im[blockingIdx]) = InnerProduct_Accum_im;
  }
}
```

Listing 6.1: Laminar Generated Cyclops Rx RRC Partition, Blocking: 120, Sub-Blocking: Disabled, Comments and Variable Names Changed for Clarity

allowing operators within the loop to be sub-blocked, assuming they are not inhibited by some other factor.

Laminar Sub-Blocking Model and Automation

Laminar includes a compiler pass that handles blocking and sub-blocking within a design. As discussed in section 5.3, Laminar creates a global blocking domain to handle blocked I/O and blocked communication between FIFOs. Within the global blocking domain, the design can further be sub-blocked with the requirement that the sub-blocking length be an integer factor of the blocking length. As part of the process, the dimensionality of each arc in the sub-blocked region is expanded with the outer dimensions equaling the sub-blocking length and operators are replaced with their sub-blocked equivalents. For blocks without a specialized sub-blocked implementation, Laminar wraps the block in its own blocking domain with BlockingInput and BlockingOutput nodes at the boundaries. Within the inner blocking domain, the dimensionality of the arcs are left unchanged with the outer arcs connected to the blocking boundary nodes being expanded. The inner blocking domain creates a fixed interval for loop which iterates over the elements in the sub-block. The operator is contained within the `for` loop with the BlockingInput and BlockingOutput nodes handling indexing into sub-blocked inputs and outputs, respectively. A depiction of sub-blocking a single operator is shown in Figure 6.6.

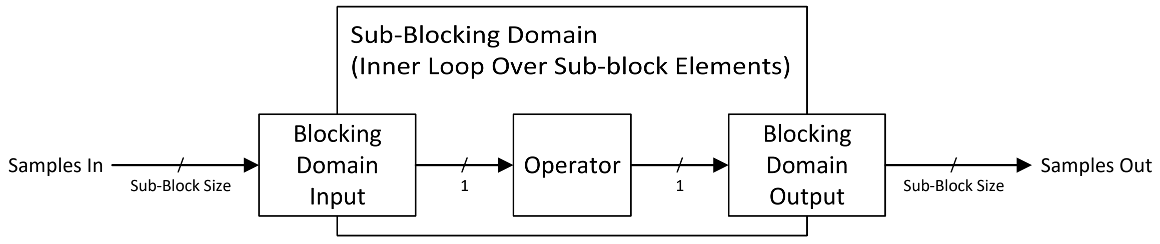


Figure 6.6: Laminar Sub-Blocking Domains

As discussed in subsection 6.2.5, loops can present a problem for automatic sub-blocking. Specifically, loops which have dependencies on samples that are delayed by less than the sub-blocking length cannot be broken and need to be executed on a sample-by-sample basis[9]. To identify the nodes that need to be executed together in the same loop, Laminar performs the following analysis on a copy of the graph:

1. Disconnect delays which are greater than or equal sub-blocking length.

[9]Smaller sub-blocking factors may allow the loop to be broken. However, there is currently a limitation that all sub-blocking within a partition must have the same base sub-blocking length. This is due to the need for more complex width adaptation and scheduling if more than one base sub-blocking length is present.

2. Find Strongly Connected Components (SCC) within the design[10].

Strongly Connected Components in directed graphs are maximal subgraphs in which there is at least one directed path in each direction between each pair of nodes in the subgraph [163]. From this property, each node in the strongly connected component participates in at least one cycle[11]. Because delays greater than or equal to the sub-blocking length were disconnected, breaking cycles with dependencies outside of a single sub-block, nodes contained within strongly connected components have dependencies within the sub-block and with each other and therefore cannot be separated when sub-blocking. For feed-forward segments of the graph which, by definition, do not participate in a loop, each node is its own strongly connected component.

After analysis, Laminar iterates over each identified strongly connected component and either encapsulates it in a blocking domain (with the appropriate Blocking Input/Output boundary nodes) or, if the strongly connected component contains a single node which provides a specialized implementation, replaces it with the corresponding specialized implementation. An example of sub-blocking when dependencies inside a sub-block and outside a sub-block exist are shown in Figure 6.7 and Figure 6.8, respectively. Note that the case containing a loop with delay greater than the sub-blocking factor allows the individual blocks within the loop to be in separate sub-blocking domains.

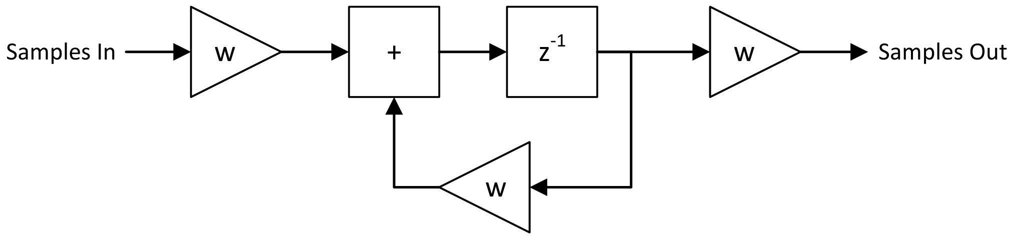
While exceptionally useful, there are some limitations of the automated sub-blocking pass within Laminar. Currently, the base sub-blocking length is required to remain constant within a partition. This avoids the added complexity of handling width adaptation and scheduling segments that operate with different block sizes on a single CPU core. FIFOs are implemented to work as width adapters and are used to support different base sub-blocking lengths in different partitions. Contexts, such as ones discovered for multiplexers and enabled subsystems, are currently restricted to reside in single sub-blocking domains. This is because the condition on what code executes can change on a sample-by-sample basis, complicating the blocking logic by requiring masking operations on the data being passed between operators. There is one notable exception to this rule: clock domains.

Interaction with Clock Domains

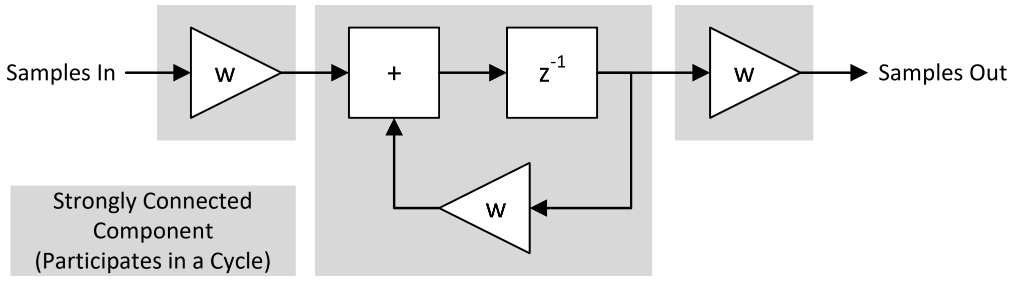
Laminar limits clock domains to having static timing relationships with each other via the use of static rate change blocks such as upsample, repeat, and downsample. Currently, only integer downsample clock domains are supported in Laminar with the standard implementation using a counter and comparator to determine if the logic contained within the clock domain should run. When sub-blocking, this would result in the entire clock domain

[10]Laminar implements the Tarjan's algorithm for finding strongly connected components as described in [163] based on the original paper, [164]. This algorithm is one of several linear time algorithms for finding strongly connected components.

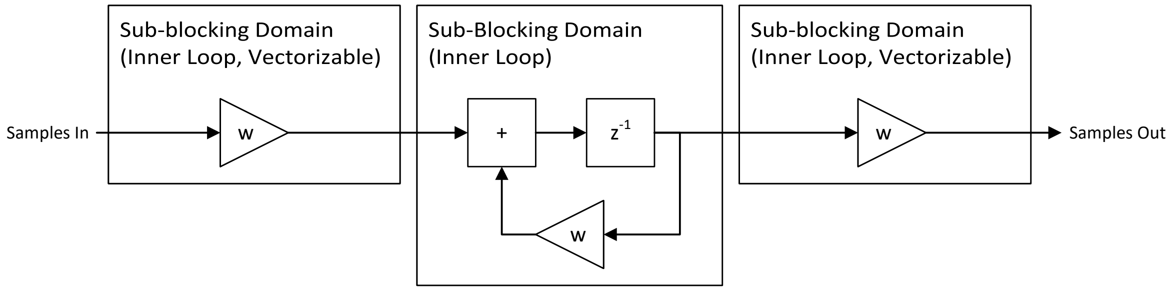
[11]For any pair of nodes A and B in the strongly connected component, by definition, there is at least one directed path from A to B and from B to A. Concatenating these paths results in a cycle containing A and B.



(a) Original Design

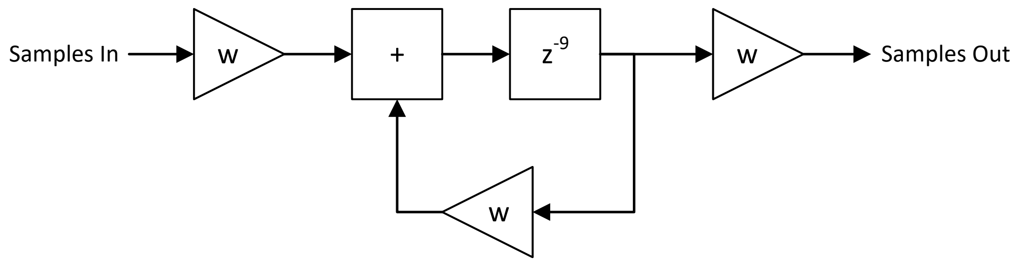


(b) Sub-Blocking Analysis

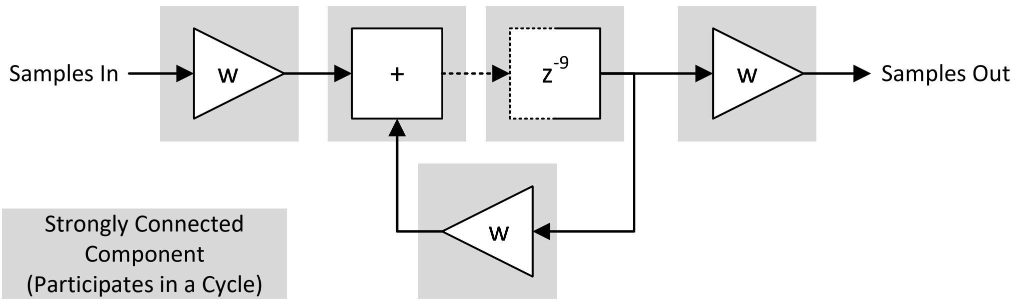

Sub-blocking Domain (Inner Loop, Vectorizable)
Sub-Blocking Domain (Inner Loop)
Sub-blocking Domain (Inner Loop, Vectorizable)
Samples In
w
+
z-1
w
w
Samples Out

(c) Sub-Blocked Design (Blocking Boundary Nodes Omitted for Clarity)

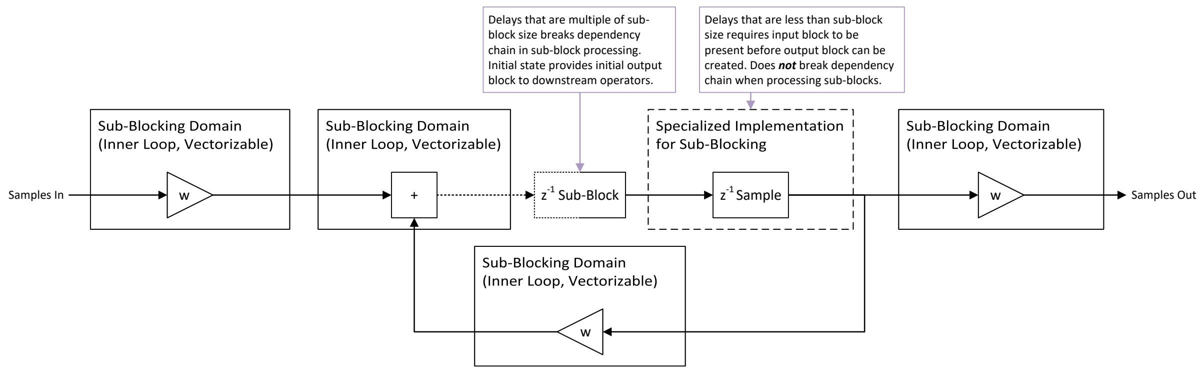
Figure 6.7: Sub-Blocking Example with Dependency Loop within Sub-block



(a) Original Design



(b) Sub-Blocking Analysis


Delays that are multiple of sub-block size breaks dependency chain in sub-block processing. Initial state provides initial output block to downstream operators.
Delays that are less than sub-block size requires input block to be present before output block can be created. Does *not* break dependency chain when processing sub-blocks.
Sub-Blocking Domain (Inner Loop, Vectorizable)
Sub-Blocking Domain (Inner Loop, Vectorizable)
Specialized Implementation for Sub-Blocking
Sub-Blocking Domain (Inner Loop, Vectorizable)
Samples In
w
+
z^{-1} Sub-Block
z^{-1} Sample
w
Samples Out
Sub-Blocking Domain (Inner Loop, Vectorizable)
w

(c) Sub-Blocked Design (Blocking Boundary Nodes Omitted for Clarity)

Figure 6.8: Sub-Blocking Example with Dependency Loop Outside Sub-block

being placed inside of a single sub-blocking domain, an undesirable result. However, for certain configurations of the clock domain, a more elegant solution is possible. When the decimation ratio is a factor of the sub-block size, the clock domain can be executed in every iteration of the outer loop, just with a smaller vector of samples passed to it. For example, a decimation by 3 of a sub-block size of 12 would result in every third sample in the sub-block being passed to logic within the downsample domain. The resulting vector would be of length 4. Likewise, sub-blocks exiting the clock domain can be easily adapted for use in the outer clock domain by either zero-filling or repeating samples as appropriate based on the output rate change block used. Inside the clock domain, the sub-blocking algorithm can continue as normal with strongly connected components identified within the clock domain that are either encapsulated in sub-blocking domains or specialized. An illustration of the result for sub-blocking with compatible downsample domains is shown in Figure 6.9. When operating in this mode, Laminar considers the clock domain to be operating in *vector mode*.

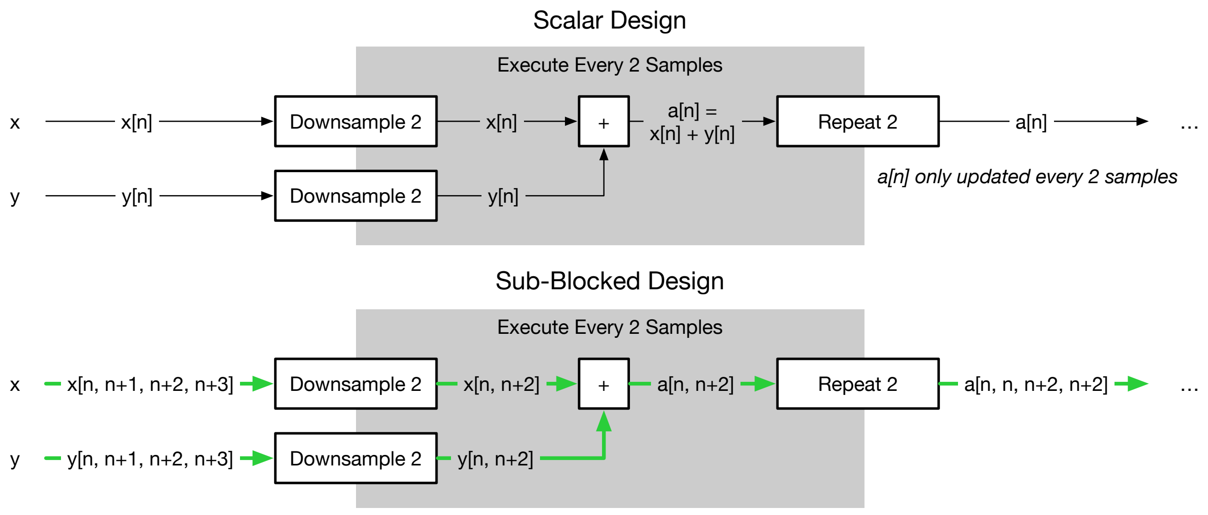


Figure 6.9: Sub-Blocking with Clock Domains Operating in Vector Mode

While the sub-block length changes within the clock domain, this is legal in the current Laminar implementation even though other mixing of sub-block lengths within a single partition is not allowed. To reconcile this conflict, Laminar uses the term *base sub-blocking length* to refer to the sub-block length at the base clock rate of the system. While the actual sub-block length is allowed to change within clock domains, the base sub-blocking length is required to remain constant within a partition. If the clock domain decimation ratio is not a factor of the sub-blocking length, it is encapsulated like any other context when sub-blocked. It is, therefore, important to select the sub-block size so that clock domains can operate in vector mode.

Specialized Blocking Implementations

While wrapping an operator in a sub-blocking domain typically results in an efficient vectorized implementation after being processed by an optimizing C compiler, some constructs perform better with specialized implementations. One such operator is delay. Not only do delays serve a special role in breaking dependency chains in loops, but their performance can also be heavily impacted by their implementation. When operating in a sub-blocked design, standard delays with a delay less than the sub-block length behave similarly to tapped delays since they return vectors of samples, shifted by an amount specified by the delay length. Due to this similarity, delays can take advantage of the different implementation types discussed in subsection 6.2.3 with minor modifications. Instead of ingesting a single sample at a time, the implementations are modified to support the ingestion of multiple samples at once. When using the oversized circular buffer approach, the buffer is sized to accommodate a complete sub-block of samples at the input. For delays that are multiples of the sub-blocking length, they can behave like standard delays operating on vectors of data instead of scalars. Because they are multiples of the sub-blocking length, no blending from samples in different sub-blocks is required at the output — sub-blocks can simply be stored as elements in the delay to be outputted in a later iteration. To help support dependency breaking properties in sub-blocked loops and to take advantage of favorable properties if the delay is a multiple of the sub-blocking length, delays which are greater than the sub-blocking length are broken up into two separate delays, one which is a multiple of the sub-blocking length and one which contains the remainder (as shown in Figure 6.8c).

Another operator that takes advantage of a specialized blocking implementation is the dot product (inner product). When sub-blocked by wrapping in a sub-blocking domain, the outer loop iterated over the elements in the sub-block while the inner loop iterated over the elements in the vector of samples. This ordering of the loops is disadvantageous when computing FIR filters, particularly long ones. When computing FIR filters, the inner loop iterates over all the coefficients in the filter. In the case when the filter is long, there may not be enough architected vector registers to keep all coefficients present in the register file. This results in some coefficients needing to be reloaded between the computation of different inner products over the sub-block. Alternatively, the order of these two loops can simply be exchanged. In this case, each of the dot products is computed for each coefficient. This allows each segment of coefficients to be read into a register once for the computation of a sub-block, shared across each of the dot products. Inspection of the disassembled executable using these two techniques on a 49-tap RRC filter revealed that the compiler was not performing the loop interchange itself, despite it being a well-known technique in HPC[12]. This result was somewhat surprising as it is one of the ideal cases for automated loop interchange with a simple exchange of the loops being possible without any modification of indexing logic within the loop. However, analyzing loop interchange in general can be tricky to statically analyze as cache behavior can be a dominant effect. In this baseband DSP

[12]For example, [165], an optimization guide for Intel processors details the potential benefits of loop interchange.

application, especially when finely partitioned across cores, the amount of required working state is rather small. This can help alleviate some cache challenges that could be a factor in more general loop interchange situations. In the RRC case, providing the specialized inner product implementation resulted in a 17.24% performance improvement over the sub-block domain encapsulation method.

Additional operators which provide specialized blocking implementations include rate changes and constants. These blocks can leverage compressed representations of data which will be discussed below.

Compressed Data Types

While not specific to blocking, opportunities for compressed datatypes are especially abundant when dealing with sub-blocking and rate changes. For example, the repeat rate change block duplicates samples at its output. When operating in vector mode, as shown in Figure 6.9, the resulting vector contains multiple successive copies of each value. Any time there is redundant information in a consistent pattern, there is an opportunity to use a compressed representation. In the case of the "repeat by n" block, the actual size of the resulting vector can be $1/n$ the length with each value only given once. By changing the indexing logic, each value can be associated with multiple indexes. Depending on the repetition factor and the size of the elements, this can result in substantial space savings, reducing the amount of memory operations required and potentially improving cache performance. Laminar contains support for handling compressed repeated data types and can generate the appropriate indexing logic if the destination node uses the provided indexing helper functions. The same technique is used by "constant" nodes when sub-blocked. Since they, by definition, remain constant across different elements in a sub-block, they emit repeated types. Since constants can be vectors or larger arrays, such as when storing constant coefficients for filters, the space savings when using the compressed repeat data type can be substantial.

Another opportunity for compressing data being passed through the design comes when tapped delays are sub-blocked. Before sub-blocking, if the input to a tapped delay is a scalar, its output is a vector. After sub-blocking, the dimension of the output is expanded, in this example to a matrix, with the new dimension being the sub-blocking length. The decision on how the dimensionality should be expanded depends on the language being used with C favoring prepending the dimension to the list. As shown in Figure 6.10, the resulting matrix exhibits a specific symmetry across the primary diagonal. Specifically, the matrix fits the definition of a *Hankel Matrix* where the elements along the skew-diagonals are constant [166], [167], although the matrix arising from the sub-blocked tapped delay is not guaranteed to be square[13]. In addition to having useful properties, the Hankel matrix can be compressed into a vector representation with rows and columns easily provided with modified indexing logic as demonstrated in Figure 6.11. Either row access or column access can be made fast due

[13]Some definitions specify Hankel matrices to be square. Under this stricter definition, these matrices can probably be best viewed as Hankel-like.

to the symmetry; however, the compiler needs to be aware of the access pattern to provide the most effective indexing. Due to the amount of redundant data in the Hankel matrix, the compressed form provides significant space savings and improves performance significantly in FIR filter contexts by avoiding excessive data movement. Laminar supports the output of compressed Hankel matrices from sub-blocked tapped delays. The tapped delays are still able to use the multiple implementations discussed in subsection 6.2.3 with operators such as dot products in FIR filters being able to leverage the compressed representation to reduce memory operations and cache pressure.

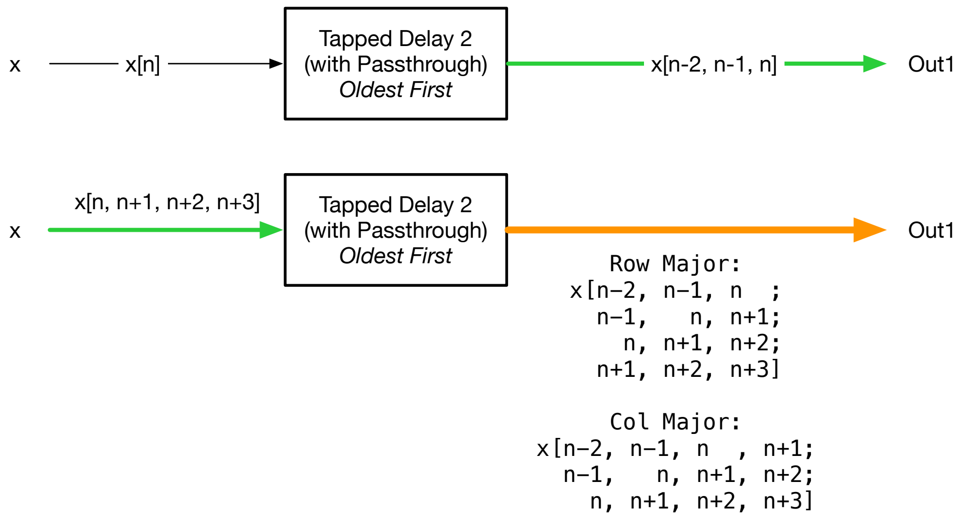


Figure 6.10: Sub-Blocking Tapped Delays

Cyclops Performance Improvements

Due to the complexity of implementing automated sub-blocking, preliminary experiments were conducted where segments of a Laminar generated Cyclops design were sub-blocked by hand. These sections included FIR-heavy partitions, such as the RRC filter and equalization. The results were so promising that the decision was made to automate sub-blocking into a Laminar pass. To demonstrate the impact of this pass, two versions of Cyclops Rev 1.40 are presented in Figure 6.12, one with sub-blocking disabled and one with sub-blocking enabled using annotated sub-blocking values inserted in the design. While a few partitions suffer a small performance drop, several experience substantial improvements with the Timing Recovery Golay Correlator and Peak Detection partition experiencing over 2.8x computation speedup. This approximately doubled the overall system performance.

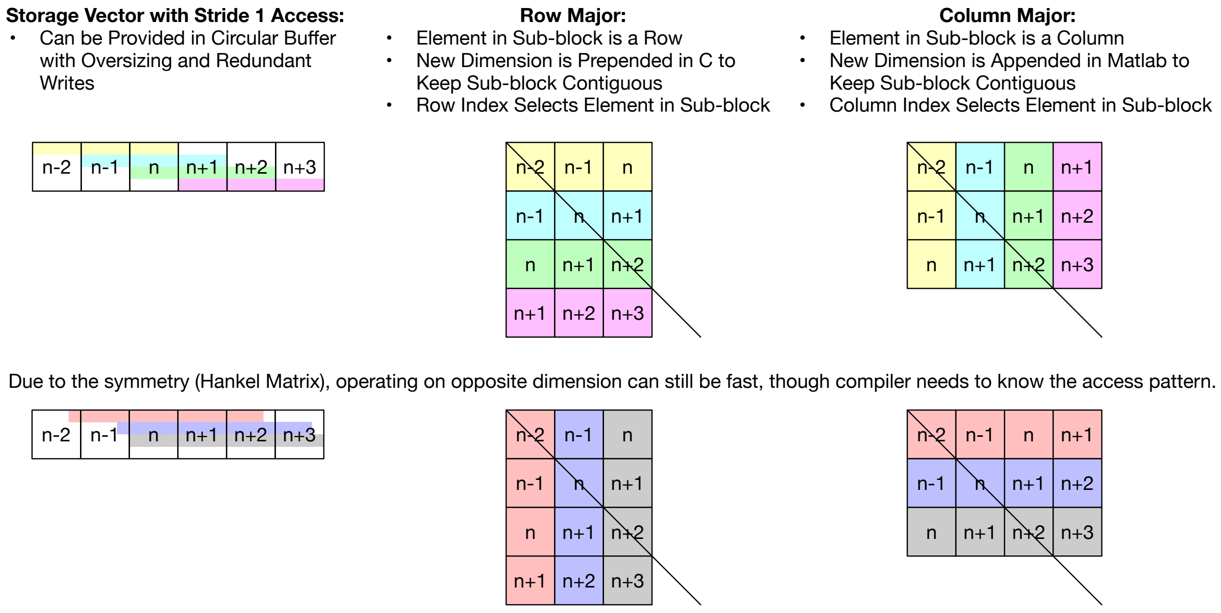
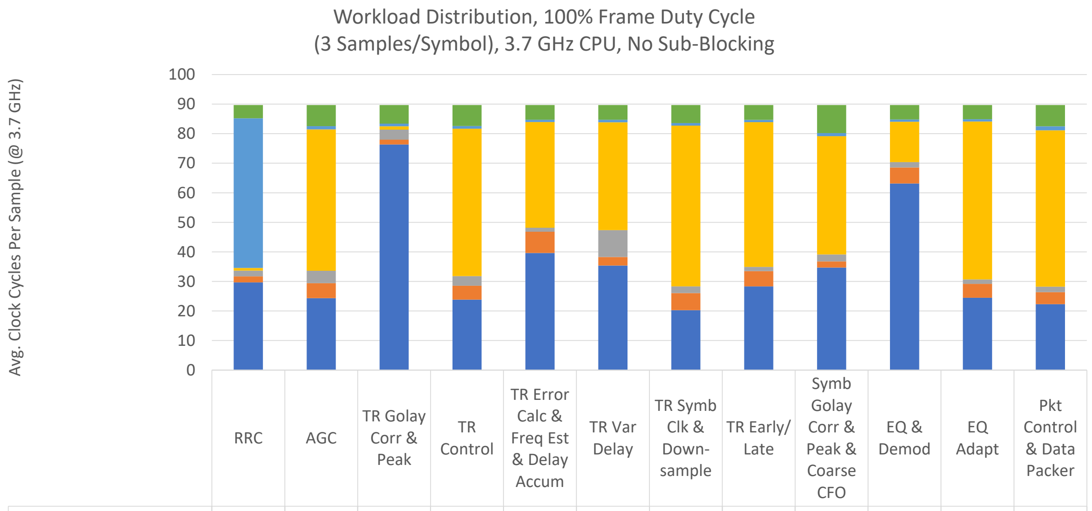

Storage Vector with Stride 1 Access:
• Can be Provided in Circular Buffer with Oversizing and Redundant Writes
Row Major:
• Element in Sub-block is a Row
• New Dimension is Prepended in C to Keep Sub-block Contiguous
• Row Index Selects Element in Sub-block
Column Major:
• Element in Sub-block is a Column
• New Dimension is Appended in Matlab to Keep Sub-block Contiguous
• Column Index Selects Element in Sub-block
n-2
n-1
n
n+1
n+2
n+3
Due to the symmetry (Hankel Matrix), operating on opposite dimension can still be fast, though compiler needs to know the access pattern.

Figure 6.11: Hankel Matrix Compressed Representation

6.3 The Role of Design Modeling

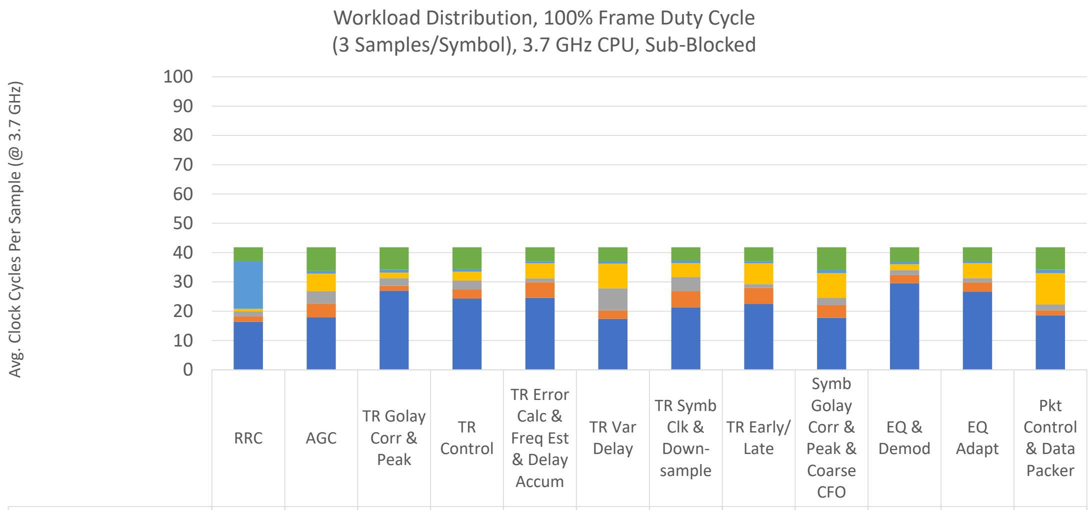
With such a wide range of possible optimization techniques, it can be challenging to determine which directions require the most attention. At several points in this project, a rough model of Cyclops baseband demonstrator was created and compared against the results achieved through the Laminar flow. This model, which was constructed by hand, along with a fine grain partitioning of the radio was used to identify mismatches in expected performance and achieved performance. One case where this analysis was exceptionally helpful was in identifying a lack of expected vectorization speed-up, particularly in the RRC filter. With the baseband executing on an AMD Epyc 7002 series CPU, a vectorization gain of about eight was expected based on eight 32-bit single precision floating-point numbers fitting in a 256-bit vector register. However, as shown in Figure 6.13, the RRC filter accounted for a similar ratio of the total compute time as was predicted from the model for a scalar design[14]. This indicated that the expected vectorization gains were not being realized, which led to an in-depth inspection of how the FIR filters were implemented. This effort culminated in the circular buffer investigation discussed in subsection 6.2.3. After modifying the tapped delay implementation, a speedup of 7.78x was observed in the RRC filter, which is close to

[14]The analytical model provided an estimate of the relative workloads of the different modules in the design, excluding FSMs. The total workload of the model was scaled to the rate of a single-core version of the radio (using shift register tapped delays) compiled by Laminar to produce the expected compute time for each module. The model itself does not estimate execution times but rather the number of scalar operators per module.



	RRC	AGC	TR Golay Corr & Peak	TR Control	TR Error Calc & Freq Est & Delay Accum	TR Var Delay	TR Symb Clk & Down-sample	TR Early/ Late	Symb Golay Corr & Peak & Coarse CFO	EQ & Demod	EQ Adapt	Pkt Control & Data Packer
🟩 Telemetry/Misc	4.4801	7.166	6.2744	7.1204	4.9824	4.9601	6.1194	5.0146	9.514	4.8799	4.8087	7.2156
🟦 Waiting for Output FIFOs	50.644	1.0494	0.9133	0.9013	0.7531	0.8676	0.8621	0.7621	1.0302	0.7846	0.7558	1.3524
🟨 Waiting for Input FIFOs	0.9021	47.8495	1.1157	49.8675	35.7841	36.4996	54.3738	49.0055	39.993	13.6534	53.4451	52.8736
⬜ Writing Output FIFOs	1.9327	4.2288	3.4314	3.2496	1.3332	9.1037	2.2819	1.4148	2.3696	1.7891	1.4977	1.8531
🟧 Reading Input FIFOs	2.0594	5.0694	1.5874	4.7201	7.2059	2.8592	5.7711	5.1713	2.1012	5.46	4.7081	4.0846
🟦 Compute	29.676	24.3312	76.3721	23.8353	39.6355	35.404	20.286	28.3259	34.6863	63.1283	24.4798	22.3159

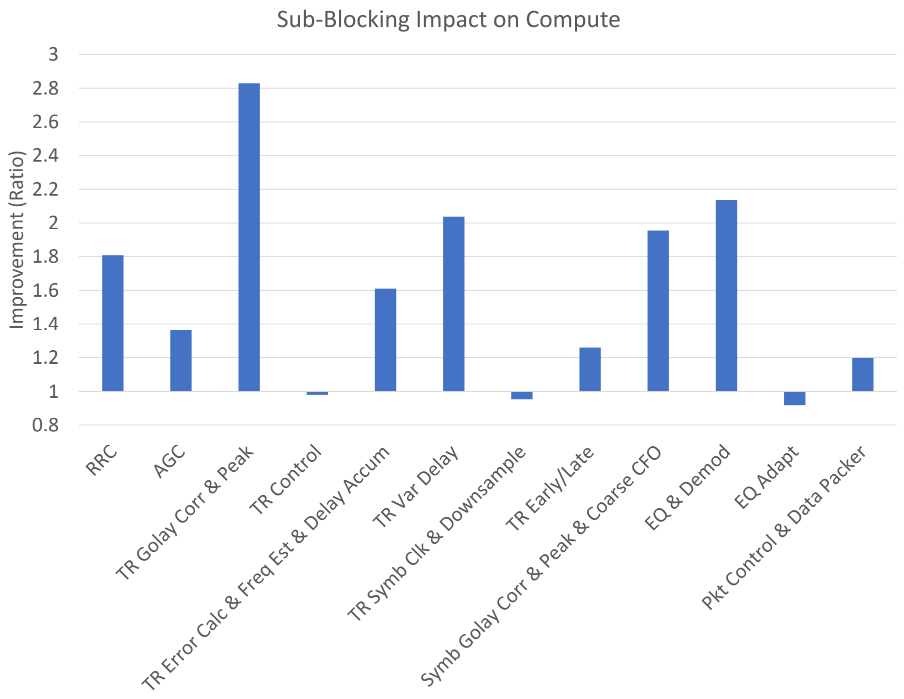
(a) No Sub-Blocking



	RRC	AGC	TR Golay Corr & Peak	TR Control	TR Error Calc & Freq Est & Delay Accum	TR Var Delay	TR Symb Clk & Down-sample	TR Early/ Late	Symb Golay Corr & Peak & Coarse CFO	EQ & Demod	EQ Adapt	Pkt Control & Data Packer
🟩 Telemetry/Misc	4.6651	8.0564	7.6207	7.4229	4.7013	4.6944	4.5057	4.7006	7.7109	4.9855	4.6671	7.6223
🟦 Waiting for Output FIFOs	16.3108	0.9422	0.9913	0.9197	0.7531	0.8591	0.8837	0.7557	1.1197	0.7902	0.7564	1.1858
🟨 Waiting for Input FIFOs	0.9141	5.9057	1.9002	3.0033	5.1348	8.4052	4.7692	7.0911	8.4444	1.9889	5.0808	10.6906
⬜ Writing Output FIFOs	1.6903	4.3032	2.6021	3.024	1.3208	7.5436	4.7698	1.3046	2.388	1.6623	1.4687	1.9737
🟧 Reading Input FIFOs	1.8103	4.744	1.6971	3.1194	5.2756	2.9234	5.579	5.4759	4.3993	2.8078	3.1421	1.704
🟦 Compute	16.4114	17.8501	26.9908	24.3127	24.6164	17.3764	21.2947	22.4741	17.7398	29.5655	26.6852	18.6238

(b) Mixed Sub-Blocking (Annotated)

Figure 6.12: Effects of Sub-Blocking on Cyclops Rev. 1.40



(c) Sub-Blocking Compute Time Improvements

Figure 6.12: Effects of Sub-Blocking on Cyclops Rev. 1.40

the expected vector performance gain.

An important thing to note about the Cyclops computation model is that it takes the expected execution rates of each operator into account when estimating the compute load of each module. The expected execution rate for each operator is determined based on clock domains and the expected conditional execution behavior for a given packet duty cycle. The impact of taking these different factors into account is shown in Figure 6.14. This complicates the automation creation of the design model. While the expected activity factor of operators in fixed-rate clock-domains can be easily determined by inspecting the compute graph, determining how often nodes in conditional execution domains execute can be difficult to gauge without simulating the design with representative test vectors or annotations by the designer. This is a similar problem to what is experienced in EDA tool digital design power estimation. The tool can provide rough power estimates with the expected static power draw of the transistors and some assumed activity factor. However, the tool can provide better estimates when activation factors are given for each transistor based on simulation data with representative workloads for the circuit[15].

[15]For an example of power modelling with different levels of detail in an EDA tool (Xilinx Vivado), see [168].

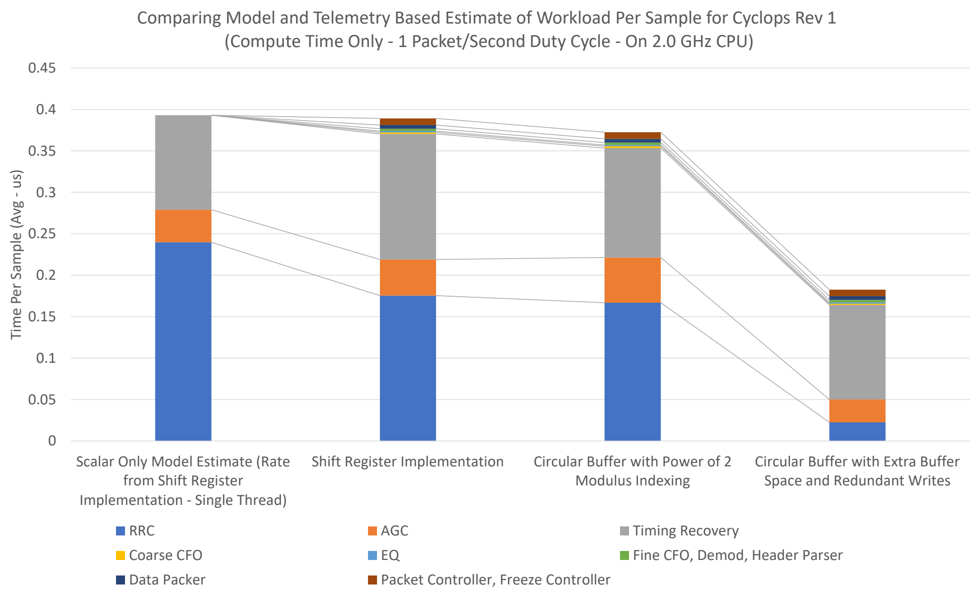


Figure 6.13: Comparing Cyclops (Older Revision) Computation Model and Measured Performance with Different Circular Buffer Implementations

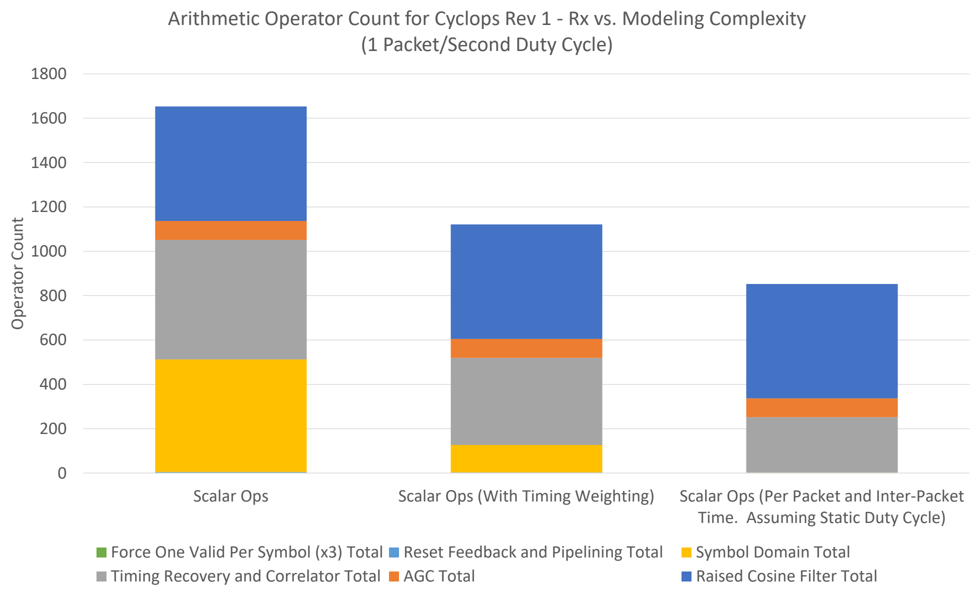


Figure 6.14: Cyclops (Older Revision) Computation Model with Different Modeling Complexity

Chapter 7

Inter-Core Optimizations

The work performed by Laminar-generated programs can be broadly divided into two types: computation and communication. The computational workload involves the execution of the various operators specified in the DSP design and was the primary focus of chapter 6. However, the various compute optimizations within single cores are often not sufficient to attain the levels of performance desired for the radio system, necessitating the partitioning of the design across cores operating in parallel. As discussed in section 5.2, data passing between partitions transit through single producer, single consumer, FIFOs that are automatically inserted by Laminar during emit. A conceptual example of FIFO insertion is shown in Figure 5.6. Because of the dependency of computation on receiving input data, the performance of the FIFO can have a strong impact on overall performance. This chapter discusses the implementation of the partition (core) crossing FIFOs, different options available within Laminar, and ideas on how to further improve performance.

7.1 Lockless x86_64 Single Producer, Single Consumer, FIFO

All FIFOs inserted by Laminar have a single producer (source partition) and a single consumer (destination partition). Values which are communicated to multiple partitions are communicated via independent FIFOs. Constraining each FIFO to only have a pair of threads accessing it simplifies the required synchronization. On typical x86_64 architectures, single producer, single consumer, FIFOs can be implemented without the use of explicit locking constructs thanks to the stricter Total Store Ordering (TSO) memory model [103], [110] mentioned in subsection 3.2.2. One such implementation uses two shared pointers to contain the read and write positions, as well as a shared buffer to hold the data being sent (shown in Figure 7.1). The dequeue operation at the consuming thread takes the following steps:

1. The consuming thread reads the write pointer. If the write pointer was modified by

the producing thread, the new value will be fetched via the cache coherency protocol. If not, the value may already be present in the consuming thread's cache and should be fast to access.

2. Using the current position of the read pointer and the value in the write pointer, the occupancy of the FIFO is computed locally by the consuming thread.

3. If the FIFO is not empty, the corresponding data is read from the shared buffer. This data will have been last written by the producing thread and will be either invalid or not resident in the consumer thread's cache. The values will be pulled into the consuming thread's cache via the cache coherency protocol. If the FIFO is empty, the consumer returns to step 1 and continues to poll the write pointer.

4. Read Pointer Update:

   a) If copied to an intermediate array, the consumer updates the read pointer after all reads have completed. If recently read by the producing thread, the read pointer may be in the shared state with the consuming thread required to obtain exclusive access to the read pointer via the cache coherency protocol before a new value can be written.

   b) If directly using the shared buffer values in the compute function (without an intermediate copy to a local buffer), the read pointer is only updated after the compute function is finished for the current block.

The enqueue operation at the producing thread takes the following steps:

1. The producing thread reads the read pointer. If the read pointer was modified by the consuming thread, the new value will be fetched via the cache coherency protocol. If not, the value may already be present in the producing thread's cache and should be fast to access.

2. Using the current position of the write pointer and the value in the read pointer, the occupancy of the FIFO is computed locally by the producing thread.

3. If the FIFO is not full, data is written into the shared buffer. This data will have been last read by the consuming thread and be in the *shared* state in the cache. The producing thread will need to re-acquire exclusive access to the cache line containing the addresses to be written via the cache coherency protocol. If the FIFO is full, the producer returns to step 1 and continues to poll the read pointer.

4. After the writes to the shared buffer have completed, the write pointer is updated. If the write pointer has been read by the consuming thread, it will be in the *shared* state in the cache. The producing thread needs to re-acquire exclusive access before the update can occur.

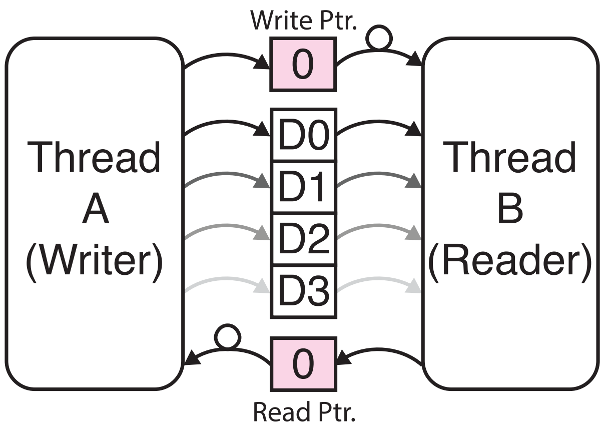


Figure 7.1: Lockless FIFO (Loop in Arc Represents Polling, Red Represents Atomic Access)

While there are no guarantees on when a write will be perceived by other cores, writes should appear in the same order to all the cores in the system under the TSO consistency model[1]. This is important as it means that, if writes to the shared buffer occur before the write pointer is updated, the values in the shared buffer will be up to date from the perspective of the consuming thread once it's reading of the write pointer indicates that data has been enqueued on the FIFO. For this to work in practice, it is also important that the reads and writes to the pointers are atomic (i.e. are perceived to occur at once). Problems can arise if reading/writing the pointer occurs in multiple, separable, steps. In these cases, it may be possible for a thread reading the value to retrieve a stale value for one part of the pointer and an updated value for another part, leading to a corrupted read. Fortunately, x86_64 supports atomic access for data types that are used for the pointers/indexes[2].

It is important for the C compiler to be aware of the memory access requirements of the FIFO. Specifically, the compiler should not re-order reads/writes across the read/write pointer accesses, and it should be aware that the values of the pointers and data can be changed outside of the function. Historically, this would have been achieved by using the `volatile` keyword in C, indicating that a side effect may exist with access to a particular memory location, along with other ISA and compiler-specific tricks [170], [171]. However, the semantics surrounding the `volatile` keyword have historically been somewhat nebulous and can slow down data access if used incorrectly. A more current best practice is to use the *standard atomic* library, available in newer versions of the C language, to formalize constraints on memory access, including memory ordering and atomic operations [170]–[172]. Specifically, Laminar uses standard atomic's release/acquire memory ordering semantics on

[1]There is relaxation of this requirement for string operations, which are discussed in [103].

[2]The C/C++ standard 'atomic' library provides a function, atomic_is_lock_free, which can be used to check if operations on a particular type are atomic [169], [170]. For the x86_64 systems used in this project, each has indicated that the atomic operations on the FIFO pointers were lockless.

the FIFO read/write pointers. Standard atomic prevents the compiler from reordering memory operations across the pointer accesses as appropriate and ensures that accesses to the pointers are atomic. x86_64's stricter memory ordering generally provides acquire/release semantics without the use of locks or explicit memory fence instructions [170], [171], [173]. This should improve performance over schemes which require more complex handshaking or enforced exclusive access to the FIFO. While standard atomic should insert appropriate locking and fence instructions on ISAs with more relaxed memory models, specialized implementations of the FIFO may perform better on those platforms. As this project focused on x86_64 based CPUs, only the FIFO implementation described above is implemented. However, adding additional FIFO implementations should not be overly onerous as Laminar defines an abstract FIFO class to handle most interactions with compiler passes.

7.1.1 FIFO Options

Laminar generated FIFOs support several different configuration options, some of which are revealed to the user via the CLI and some which require a minor tweak to the Laminar source to use. One of the most useful options is the ability to use local copies of the read and write pointers when possible, to avoid unnecessary latency fetching the current pointer value. In this mode, a local copy of the opposing thread's read or write pointer is kept and used to compute the FIFO empty/full status as appropriate for the given thread. If status indicates that the operation (enqueuing or dequeuing) can occur, the operation is allowed to proceed. However, if the current FIFO status indicates that the desired operation cannot occur, the current value of the opposing thread's pointer is read and the local copy is updated. The FIFO status can then be rechecked. For the consuming thread, this reduces latency and cache coherency traffic if multiple items are present in the FIFO at the time the write pointer was last checked. The thread can consume up to the number of items it last knew were in the array before rechecking the pointer. The producing side is similar, with the read pointer having a local copy and the amount of empty space in the FIFO being the critical factor. In Laminar, this mode is known as *index caching* and can be enabled on the consuming side, the producing side, both, or neither across all FIFOs in the design via a command line option.

Another option involves how values are copied between the FIFO and local buffers (if used). Options include classic for loops, calls to memcpy, calls to __builtin_memcpy_inline, or calls to a function using vector intrinsics to perform the copy (fast_copy_unaligned_ramp_in). __builtin_memcpy_inline is a LLVM/clang language extension which guarantees a specialized, inline implementation of memcpy will be used with the requirement that the size of the copy be known at compile time [174]. While Laminar's approach was to generally avoid the use of compiler-specific features or intrinsics, this was one case where coaxing the compiler to perform the desired action was challenging. In many cases, C compilers can detect a memory copy operation implemented with a `for` loop and internally convert it to a memcpy call. This is typically desired, as memcpy often provides the most efficient implementation of memory copy operations on a given platform. However, it was observed in several disassembled binaries that the compiler would sometimes include calls to memcpy rather than inline the operations contained within memcpy. Since all copy sizes are known at compile time, the added delay of the function call along with the more general logic in memcpy resulted in degraded performance relative to when the operations were inlined. The __builtin_memcpy_inline function forces the specialization and inlining of the operations which resulted in improved performance. There can be some occasional interference of __builtin_memcpy_inline with other compiler optimizations and is why the implementation of the Delay block does not use it by default. The style of FIFO copying can be changed by modifying the constructor in the `ThreadCrossingFIFO` class and re-compiling Laminar.

7.2 Communication/Computation Overlap

As can be seen in Figure 6.12b, communication takes a non-negligible amount of the total time spent by each CPU core in a radio system. Currently, there is serialization of FIFO transactions and performing useful computation on blocks. This is in part because the FIFO conditions must be checked before read or write transactions can be performed. It is also partially because data is copied to/from local buffers that the compute function interacts with. While there can be some overlap due to speculative and out-of-order execution, this is limited to the boundaries around the call to the compute function. With large blocks to operate on, the amount of overlap provided would be small.

Ideally, communication and computation would overlap so that execution units in the CPU can be performing productive work while data movement is occurring in parallel. In general, overlapping communication and computation in systems with some degree of load imbalance would mostly affect bottleneck threads as other threads will be forced to either wait for input data to become available or wait for space in output FIFOs to open up[3]. With ideal load balancing, a block always available at each input FIFO, and space always available in each output FIFO, overlapping communication and computation can impact each thread. Several different techniques to introduce more communication/computation overlap were attempted with Laminar, some of which can still be emitted with command line options. However, most attempts yielded either inconsistent or suboptimal results.

In-Place Buffer Access

The default FIFO behavior is for each accessing thread to copy data between a local buffer and the shared buffer in the FIFO. This allows each thread to operate on its local copy of the data without causing potentially unwanted cache coherency traffic. Unintended cache contention can occur if a reading thread and a writing thread are both accessing a single

[3]Since Laminar uses fixed length FIFOs, in steady-state operation FIFOs going into bottleneck threads will fill up on average and FIFOs exiting bottleneck threads will tend to be empty. One exception is when feedback loops have insufficient initial state. In this case, participating threads in the loop can be artificially starved.

shared cache line in the FIFO buffer, possible if the size of an individual FIFO transaction is not a multiple of the cache line size. The Laminar FIFOs support an alternate mode where the threads can read/write directly to the shared buffer. One advantage of this approach is reduced memory requirements as no duplicate copy of the data is required. Also, if data is access by enabled subsystems, providing direct access to the shared buffer can reduce the number of required memory read operations when the subsystem is disabled.

Initial tests of direct FIFO buffer action with a simple cascaded FIR filter design, similar to the one used in the experiments depicted in Figure 5.11, operating on an Epyc 7002 series processor appeared promising with overlap appearing to occur[4]. Results from the test are shown in Figure 7.3 compared to the default FIFO implementation copying between local buffers and the shared buffers shown in Figure 7.2. An early version of Cyclops running with a 5% packet duty cycle, generated by an earlier version of Laminar, and running on the Epyc 7002 series CPU yielded a net speedup. However, the results were not consistent across threads with some experiencing a speedup and some experiencing a slowdown. Threads which contained enabled subsystems appeared to show over 100% speedup, but this was likely due to avoiding reads when the subsystems were disabled. This effect has been largely negated in more recent, sub-blocked, versions of the Cyclops radio which use static decimation for the symbol domain and have reduced their reliance on enabled subsystems to aid in vectorization. When the packet duty cycle was elevated to $\approx 80\%$, overall performance was degraded when using in-place FIFO access. When moving to the Ryzen Threadripper 3970X, use of in-place FIFO access degraded performance for both 5% and 80% packet duty cycles.

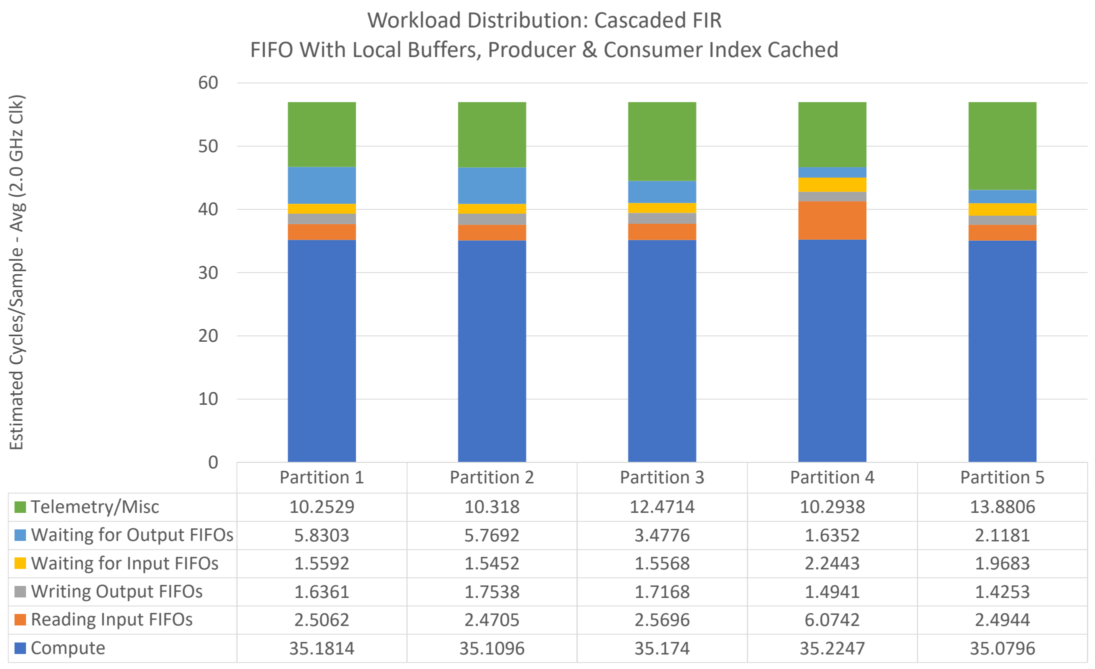
The current Cyclops design with sub-blocking was tested with in-place FIFO access with significant performance degradation observed (shown in Figure 7.4) compared to the version with the default FIFO implementation (shown in Figure 6.12b). Because of the observed performance degradation with the full baseband, the decision was made to offer in-place FIFO access as an option but not to make it the default.

Double Buffering

In an effort to alleviate the problems encountered with in-place access to the FIFO buffer, an alternative approach with multiple local buffers was attempted. This approach uses one local buffer to contain the block currently being processed in the compute function and another to contain blocks in transit. On the input side, a sample is copied into this secondary local buffer during each execution of the blocking loop in the compute function[5]. The idea is that this copy operation is not depended on by any operation in the compute function with out-of-order execution allowing the load to execute in parallel with the compute function. After the compute function completes executing, the local buffers are swapped for the next block. A similar approach is used for the output FIFOs with one local buffer used to store the results for the current block as they are computed, and another local buffer used to hold

[4]When using direct FIFO buffer access, the Laminar inserted telemetry measurements cannot isolate the time spent accessing the FIFO. This time is included in the *compute* measurement.

[5]Sub-blocking was not yet implemented when this experiment was conducted.



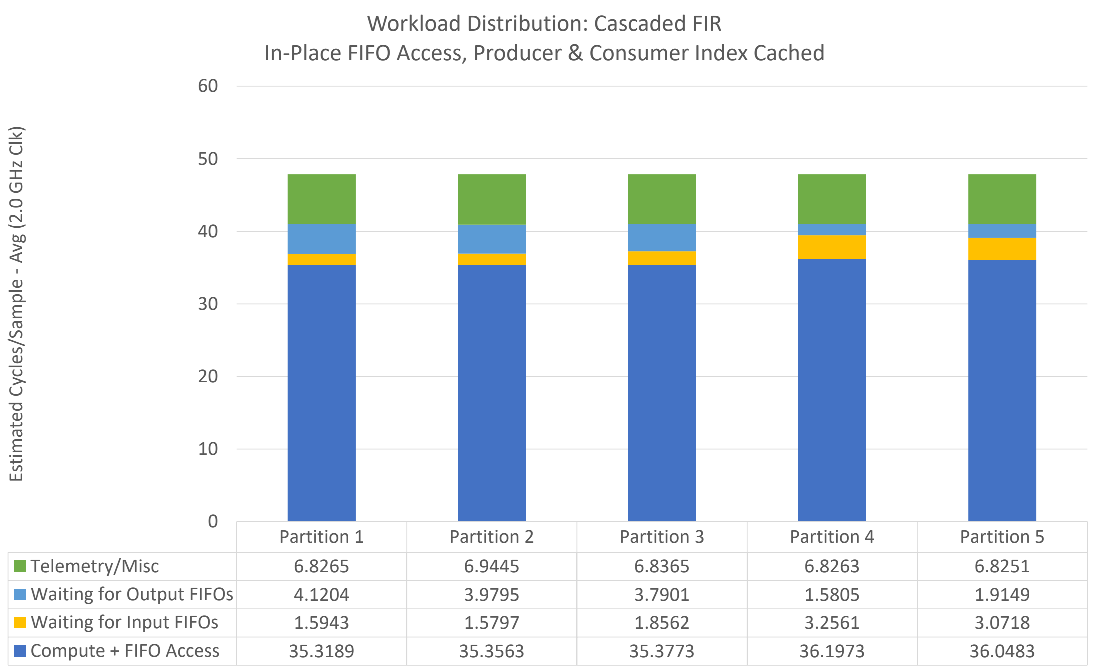
	Partition 1	Partition 2	Partition 3	Partition 4	Partition 5
🟩 Telemetry/Misc	10.2529	10.318	12.4714	10.2938	13.8806
🟦 Waiting for Output FIFOs	5.8303	5.7692	3.4776	1.6352	2.1181
🟨 Waiting for Input FIFOs	1.5592	1.5452	1.5568	2.2443	1.9683
⬜ Writing Output FIFOs	1.6361	1.7538	1.7168	1.4941	1.4253
🟧 Reading Input FIFOs	2.5062	2.4705	2.5696	6.0742	2.4944
🟦 Compute	35.1814	35.1096	35.174	35.2247	35.0796

Figure 7.2: Cascaded FIR Filter Test (Early Laminar Version) - Default FIFO Implementation

the results for the last block. In each iteration of the blocking loop in the compute function, a sample is copied from the secondary buffer to the shared FIFO buffer. Again, there should be no dependencies between this operation and operations inside of the compute function, allowing the out-of-order core to execute the store in parallel with operations in the compute function (subject to the x86_64 memory consistency model).

The hope was that this approach would provide the benefits of computing with local copies of data while being able to perform FIFO accesses in parallel. An initial test with concatenated FIR filters (shown in Figure 7.5) was promising with performance similar to that of the version with direct FIFO buffer access (shown in Figure 7.4). However, performance was less consistent with one showing growth in the *Compute + FIFO Access* time and slightly slower overall performance.

A principal disadvantage of the double buffer technique is the additional state required, both in terms of memory and the data which must be available in FIFOs. The addition of a second local array places increased pressure on lower-level caches, potentially evicting values that are required for computation. If used on inputs, two blocks of input data are required before a result is produced with double buffering. If double buffering is also used on outputs, three blocks of data are required before an output is passed to downstream threads. This is an acute problem for designs which contain feedback loops across partitions with limited state. If viewed similarly to a KPN, each thread would consume two tokens immediately



	Partition 1	Partition 2	Partition 3	Partition 4	Partition 5
🟩 Telemetry/Misc	6.8265	6.9445	6.8365	6.8263	6.8251
🟦 Waiting for Output FIFOs	4.1204	3.9795	3.7901	1.5805	1.9149
🟨 Waiting for Input FIFOs	1.5943	1.5797	1.8562	3.2561	3.0718
🟦 Compute + FIFO Access	35.3189	35.3563	35.3773	36.1973	36.0483

Figure 7.3: Cascaded FIR Filter Test (Early Laminar Version) - In-Place FIFO Implementation

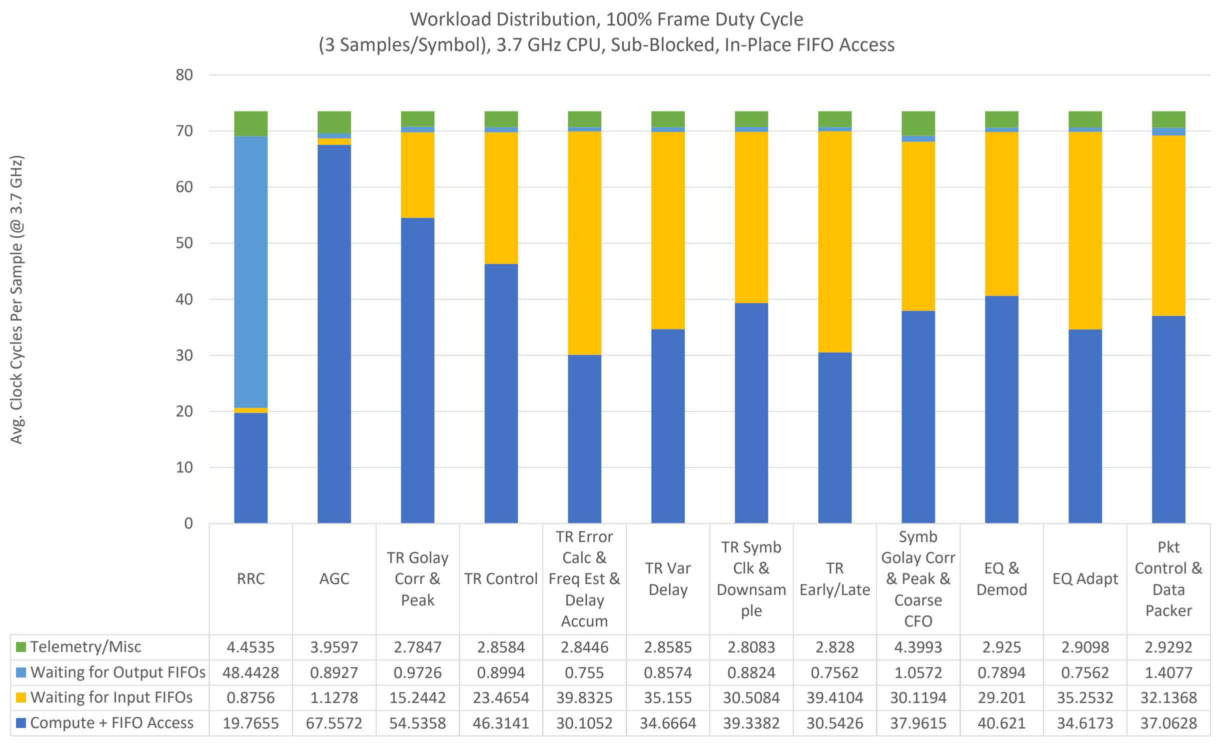
and would only produce a result on each received token after the second.

Because of the increased state requirements, tests on an early version of Cyclops were conducted with reduced transaction block sizes. At 5% packet duty cycle, the *Compute + FIFO Access* time was larger in most threads relative to the direct in-place FIFO buffer access. This was possibly caused by the reduced transaction block size less effectively amortizing the fixed costs associated with FIFO access. However, because the block size reduction was required to employ double buffering on the Cyclops design, the double buffering technique was set aside. Currently, double buffered FIFO implementations are not implemented for sub-blocked designs.

Software Prefetching

Akin to the double buffering approach is the use of software prefetching instructions to request cache lines be pulled into a core's local cache before the read is issued, avoiding latency on the load[6]. Prefetching is a sufficiently important technique that hardware prefetching units are included in many modern CPUs including the ones used in this project, such as the Ryzen Threadripper 3970 [17]. These prefetching units analyze the memory operations issued by the processor and attempt to anticipate future requests, prefetching values for those

[6]For more information on prefetching, including software prefetching, see [16], [175].

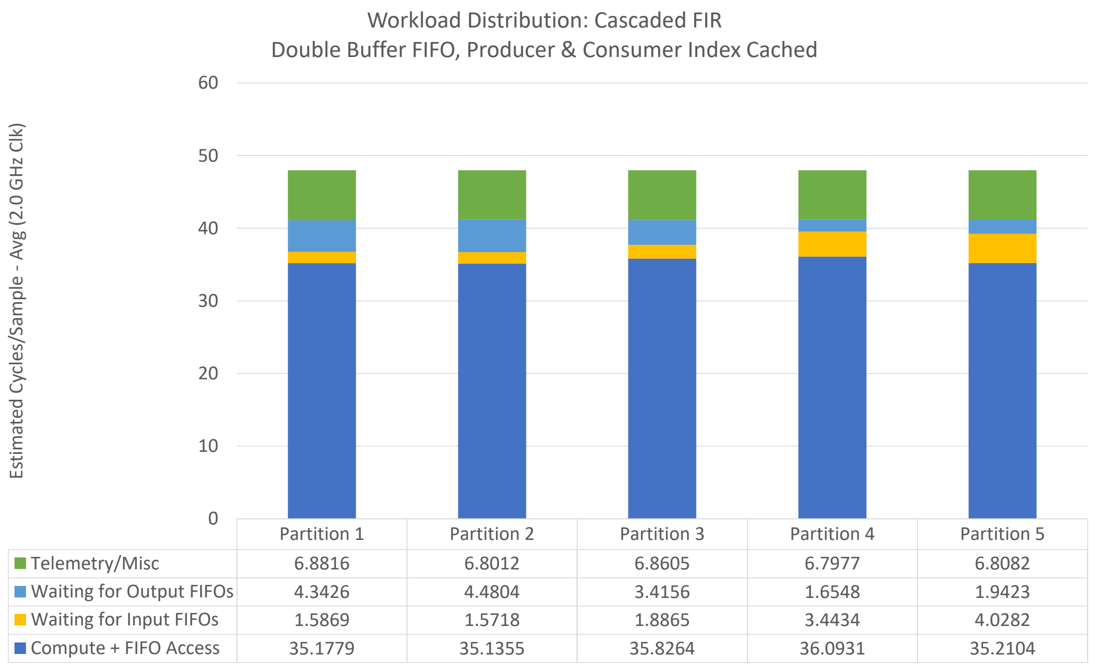


	RRC	AGC	TR Golay Corr & Peak	TR Control	TR Error Calc & Freq Est & Delay Accum	TR Var Delay	TR Symb Clk & Downsample	TR Early/Late	Symb Golay Corr & Peak & Coarse CFO	EQ & Demod	EQ Adapt	Pkt Control & Data Packer
🟩 Telemetry/Misc	4.4535	3.9597	2.7847	2.8584	2.8446	2.8585	2.8083	2.828	4.3993	2.925	2.9098	2.9292
🟦 Waiting for Output FIFOs	48.4428	0.8927	0.9726	0.8994	0.755	0.8574	0.8824	0.7562	1.0572	0.7894	0.7562	1.4077
🟨 Waiting for Input FIFOs	0.8756	1.1278	15.2442	23.4654	39.8325	35.155	30.5084	39.4104	30.1194	29.201	35.2532	32.1368
🟦 Compute + FIFO Access	19.7655	67.5572	54.5358	46.3141	30.1052	34.6664	39.3382	30.5426	37.9615	40.621	34.6173	37.0628

Figure 7.4: Cyclops Rev. 1.40, Sub-Blocked, In-Place FIFO Access

requests ahead of time. Despite its simple premise, prefetching can be very challenging, requiring a fine balance between fetching data into the cache early enough to be useful but not so early that useful information is prematurely evicted to make room for the prefetched value [175]. ISAs like x86_64 often provide software prefetching instructions that act as hints for the CPU about what values will be accessed soon. However, software prefetching has had an unreliable track record in general-purpose computing with its use sometimes degrading performance relative to relying on the hardware prefetcher. The hope was that the more predictable data access pattern in dataflow design would lend itself to software prefetching.

An attempt was made to add prefetch instructions to the compute function operating with direct access to the shared FIFO buffer to prefetch samples ahead of the iteration in which they would be used. However, this approach resulted in degraded performance over simple direct FIFO buffer access. Software prefetching may be worth re-investigating at some point, but it is likely that fine tuning will be required to, at the very least, avoid harming performance.



	Partition 1	Partition 2	Partition 3	Partition 4	Partition 5
🟩 Telemetry/Misc	6.8816	6.8012	6.8605	6.7977	6.8082
🟦 Waiting for Output FIFOs	4.3426	4.4804	3.4156	1.6548	1.9423
🟨 Waiting for Input FIFOs	1.5869	1.5718	1.8865	3.4434	4.0282
🟦 Compute + FIFO Access	35.1779	35.1355	35.8264	36.0931	35.2104

Figure 7.5: Cascaded FIR Filter Test (Early Laminar Version) - Double Buffer FIFO Implementation

7.3 Lockless, Reduced Handshake, FIFO Concept For x86 Systems (Concept)

One challenge with FIFO communication on a conventional shared memory system is that a handshaking procedure is required to determine if the FIFO is ready to support an enqueue or dequeue operation at a given moment in time. While maintaining local copies of the last known read and write pointers (as discussed earlier) can help reduce the number of required handshakes in cases where multiple elements are available (on the consuming side) or multiple empty slots are available (on the producing side), handshaking is still regularly required. One method of addressing this relatively fixed overhead is to transfer multiple samples at a time so that the cost of the handshake is amortized over more elements. This is accomplished by blocking transactions between threads and has been an important tool to improve overall performance.

Conceptually, another way to reduce the overhead associated with FIFO transactions would be to virtually guarantee that data would be available at the consuming side and room was available on the producing side. In this case, accessing the FIFO could occur unconditionally without the required handshaking to check the FIFO status. One way to achieve this would be to ensure that each thread in the system operated steadily at the same rate so that the enqueuing and dequeuing rates of each FIFO were exactly matched. This type of operation is possible in digital hardware designs where different modules can share common clocks. However, it was not clear if this would be possible on modern multicore CPUs which often contain multiple power and clock domains that can be unsynchronized. A series of experiments were conducted to see if this type of system was even possible on a modern CPU and to quantify any performance gain that can be achieved.

The proposed *Reduced Handshake FIFO* is similar in structure to the standard FIFO discussed earlier. Like the standard FIFO, there are shared read and write pointers which track the state of the FIFO. The consuming thread modifies the read pointer, and the producing thread modifies the write pointer. The threads share a buffer space where data to be transacted is written by the producing side and read by the consuming side. The Reduced Handshake FIFO adds another shared variable, called the *rate adjustment* variable. Due to interrupts, kernel housekeeping, and the possibility of unsynchronized clocks, it is anticipated that minor adjustments will need to be made to the speed of the communicating threads. The consuming thread periodically reads this value and modifies its execution rate accordingly. Rate adjustments were implemented using NOP insertion into the execution loop but modifications to the core clock via DVFS is a compelling alternative. Each block being transacted also optionally contains two ID fields. These ID fields, combined with acquire/release semantics and the Total Store Ordering (TSO) x86_64 memory model, are used to detect overflow or underflow errors. A separate shared variable is used for the reader to signal that it has detected an overflow or underflow condition.

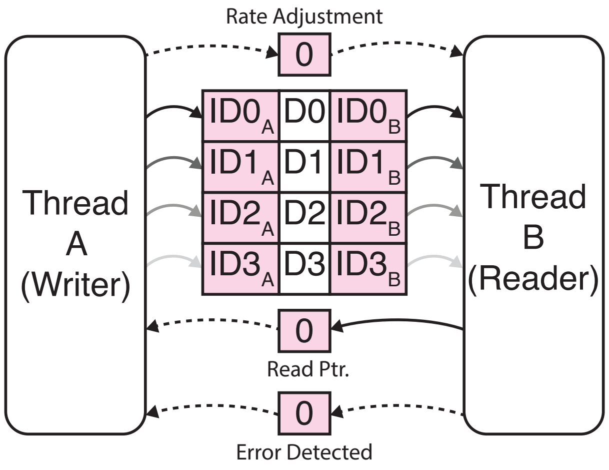


Figure 7.6: Reduced Handshake FIFO (Dotted Line Represents Occasional Access, Red Represents Atomic Access)

The FIFO is initialized to be half-full, as this provides the maximum headroom for an overflow or an underflow. Before starting, the reading and writing threads touch the values in the buffer and the shared flags in the same order they would if the system was in steady-state. This sets the cache state to be as close as possible to what is present during steady-state operation and helps prevent a large transient startup rate imbalance between the producer and consumer.

In this experiment, the producing thread acts as the speed controller. It can control its own rate by inserting NOPs in its regular computation and can communicate the requested rate changes to the consuming thread via the shared *rate adjustment* flag. Occasionally, the writer checks the read pointer value and computes the FIFO occupancy. It then checks how far away the occupancy is from the desired set point, half-full, and computes rate adjustments for itself and the consuming thread. This computation attempts to keep the rate high while alleviating any imbalance. Because the occupancy check occurs occasionally, the read pointer is allowed to remain in the exclusive state in the consuming thread's cache, allowing fast writes. Because the consuming thread does not need to check the write pointer, the pointer becomes a local variable of the producing thread and can be accessed quickly. Occasionally, the reading thread checks the rate adjustment flag and reacts accordingly. Because this is an infrequent check, it should have a relatively low impact on average performance. The rate at which it needs to check is governed by how closely the rates of the two threads can be maintained and how long the FIFO is (how much headroom exists before an overflow or underflow).

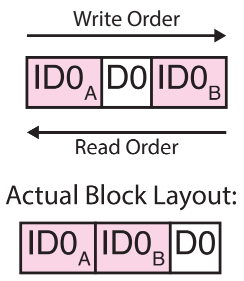


Figure 7.7: Reduced Handshake FIFO Block Access Order and Organization

In steady-state operation, the producer unconditionally writes blocks to the shared buffer. It can optionally include IDs which can be used to check if an overflow or underflow occurred. One method of generating these IDs is to increase the ID by one each time, with two IDs written for each block. When writing, ID_A is written first, followed by the data (no constraint of write ordering within the data), and then ID_B. The consumer unconditionally reads blocks from the shared buffer. If IDs are present, ID_B is read first, followed by the data, and then ID_A. Because of the write ordering preservation of the TSO memory consistency model, if ID_B and ID_A match what is expected, the data contained in the block should be valid. ID_B provides a mechanism for detecting underflow or overflow by comparing the ID with the expected ID at the consumer. ID_A guards against an overflow condition in which the producer begins writing to the block after the reader has begun reading. If an error is detected, the consumer can signal to the producer via the *error detected* flag. Depending on the semantics of the system, this may require the program to reset and restart.

Multiple rate adaptation algorithms can be used in this closed-loop control system. Purely open-loop (no adaptation), bang-bang control, and PI control were all investigated. For each of these methods, efforts were taken to reduce jitter from the OS as it can cause momentary rate imbalances, requiring larger buffers to absorb. Most of these options are configurable as boot options to the Linux kernel or can be set at runtime. They include:

- Isolating cores from the Linux Scheduler (isolcpus) and leaving at least one core for the OS.
- Offloading interrupts from isolated cores (setting smp_affinity_masks).
- Locking core clock frequencies (or at the very least enforce the same upper bound).
- Disabling CPU boost.
- Offloading RCU callbacks to OS core.
- Disabling watchdog timers.
- Bringing isolated CPUs offline and online to migrate kernel tasks off of them.

For the rationale behind some of these options, see the following application notes on reducing OS jitter: [141], [142], [176]–[178].

For open loop, a Linux driver (sir: Simple Interrupt Reporter) was created to both report interrupt events and to disable interrupts on cores [179]. In addition, the kernel was recompiled with support for NO_HZ_FULL enabled. The idea behind NO_HZ_FULL is to reduce the number of Local Timer Interrupt (LOC) events by only scheduling hardware interrupts when required on CPUs with only a single runnable task [176]. However, using the telemetry from the sir driver, it was observed that NO_HZ_FULL tended to cause clusters of Local Timer Interrupt events. Clusters typically appeared around low-resolution software timer events and appeared to partially be due to timer servicing occurring in the softirq handler. The interrupts clusters were problematic because they would interrupt the operation of threads long enough to cause an overflow or underflow condition to occur. It was determined that avoiding NO_HZ_FULL resulted in better stability because, even though potentially more interrupts would occur, they were spread out in time and occurred with approximately the same frequency on the two cores participating in the experiment. Surprisingly, the open-loop experiments, with proper tuning and interrupts disabled through the sir driver, were able to run on the order of minutes without failure.

Of the two closed-loop control systems, the Proportional/Integral (PI) controller was the most versatile and reliable. With parameter sweeps, good initial delays were set for each participating thread in the communication. The P and I parameters were also set in the producing/controlling thread. The results were promising with the system running to a test limit of $2x10^6$ transactions. The performance of the reduced handshake FIFO compared to a traditional FIFO running on a Ryzen Threadripper 2990WX is shown in Figure 7.8.

The reduced handshake FIFO does provide a speedup, especially at lower block sizes. As the block size increases, the advantage decreases as the traditional FIFO can increasingly amortize the fixed transaction costs. The plot in Figure 7.8 is noisy due to the vector intrinsic copy function used in the experiment coupled with varying block sizes that are sometimes not multiples of the vector length. A linear fit model based only on vector aligned block sizes is shown as a smooth line on the plot.

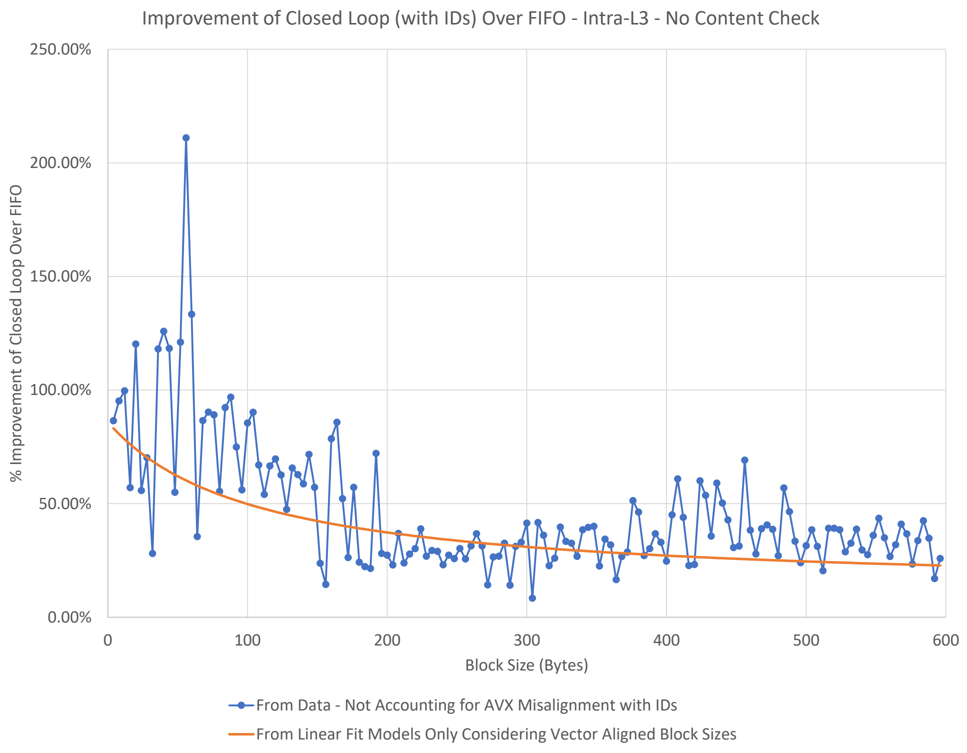


Figure 7.8: Reduced Handshake FIFO vs. Standard FIFO (using fast_copy_unaligned_ramp_in Copy Function) - Intra-L3 on Ryzen Threadripper 2990WX

While promising, especially at low block sizes, there are some challenges in using the reduced handshake FIFO in practice. In a practical DSP system split across several cores, there will be a network of connections which will all need to be rate adapted relative to each other. Interconnected local control systems can lead to oscillations that can be challenging to handle. An alternative would be to have some central controller with global knowledge of FIFO occupancies which would then issue rate change commands to each thread. With global knowledge, it would potentially be easier to avoid over-correcting in one FIFO and causing another to enter an error state. Another problem with practical systems comes from the initialization of the FIFOs to the half-full state to provide a runway for load imbalances to be corrected before an overflow or underflow occurs. While not an issue for feedforward segments, this potentially requires extensive state in loops crossing multiple cores. Because of these challenges, this concept has not yet been integrated into the Laminar flow. However, it likely deserves to be re-visited as it provides tangible performance benefits, especially with small block sizes. The code used to drive this experiment is included in [180] along with early versions of core-core benchmarks.

7.4 Inter-Core Communication Benchmarking

Communication between cores is such a significant component of the Laminar compute model that understanding the expected performance is important to obtain the desired results. To aid in understanding, a series of benchmark suites were produced to test the mechanisms used for core-core communication on the target platform. This includes estimating the latency for two threads to exchange values via a single address in memory and other patterns of data access including FIFOs. Early benchmarks are contained in [180]. These benchmarks provided useful information on the operating characteristics of the memory subsystem and were exceptionally important when evaluating performance jitter when evaluating the reduced handshake FIFO concept described earlier.

As the FIFO implementation used in Laminar matured, the C++ based benchmarking used in [180] was largely superseded by a suite of C based tests in [181] which more closely paralleled the Laminar FIFO implementations. These benchmarks are used to evaluate the expected performance of the cache coherency system under different operating conditions. Scenarios include whether a single pair of threads or multiple pairs are active at once as well as if paired threads are within the same L3 cache domain or reside in different domains. The benchmark suite also includes memory bandwidth measurements to check the efficiency of the memory copy methods as well as to compare against the advertised memory bandwidth specifications for the part.

The FIFO test results, shown in Figure 7.9, highlight the different performance Laminar programs experience when communicating within an L3 versus between L3s. It also shows that the workload present can have a strong impact on performance. For example, there is a significant throughput gap between a single pair of cores within an L3 communicating (top blue line) versus when two pairs of cores within the L3 are communicating (orange line)[7]. There is also a significant difference between a single pair of cores in different L3 domains communicating compared to when multiple cores are communicating via the L3.

Comparing the inter-L3 results with what is expected from the memory bandwidth (shown in Table 7.1), there is a gap. This could suggest that either the method used to copy data to/from the cache is inefficient or that the interconnect system becomes congested with cache coherency traffic (or a combination of both). Using memory benchmarks, which read and write large arrays in memory that are much larger than the cache size, an estimate

[7]The gray line is when all cores across the system, except for two L3s which are hosting the OS during the test, are paired up within their L3. This is to test if there is a significant cross L3 performance impact when almost all communication is contained within L3 domains. The measurements suggest that there may be some cross L3 effects but that it is small on average.

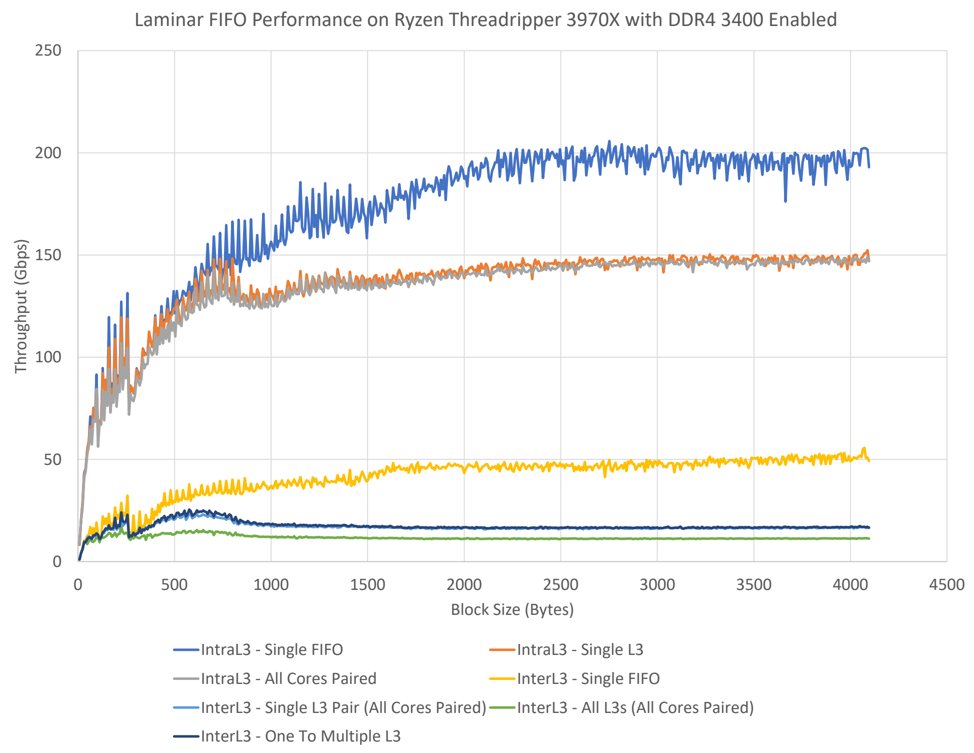


Figure 7.9: Measured Throughput of Laminar FIFOs on Ryzen Threadripper 3970X with 3200 DDR4 - CPU Clock Limited to 3.7 GHz, NPS_AUTO

of the achieved memory bandwidth with the same memory access technique in the FIFO transfers.

Based on DDR4-3200 DRAM, the maximum expected bandwidth per channel is $8*3200 = 25$GB/s. With 4 channels, the expected maximum aggregate memory bandwidth is 819.2 Gbps. This is close to the performance measured using 24 cores and suggests that the memory access technique (memcpy variant) is performing well[8] One possible method to reduce cache coherency workload is the use of special non-temporal instructions which provide a hint to the CPU that the data should not be cached [103], [104]. Stores using these non-temporal instructions should have the advantage of avoiding cache pollution and maximizing memory bus utilization [104]. However, there is a latency price to pay with this technique, as stored data needs to completely transit to DRAM with subsequent fetches requiring a full DRAM

[8]Performance could potentially be improved by changing the NPS setting in the UEFI/BIOS to NPS4 as described in the AMD Tuning Guide [182]. It should also be noted that only 24 of the 32 cores in the system participated in the test. One L3 domain, which hosts the OS processes, was not included to avoid polluting the measured performance. A second L3 domain was not included because, in the other test cases, it would have been paired with the L3 domain hosting the OS processes. This was carried over to the DRAM tests to facilitate a closer comparison to the other test cases.

Mode	FIFO Throughput (Gbps - Aggregate)	Memory Read (Gbps)	Memory Write (Gbps)
Single FIFO Between L3s	50.1	170 (Single Core)	129 (Single Core)
4 FIFOs (Between 2 L3s)	66	343 (Single L3)	179 (Single L3)
4 FIFOs Per L3 Pair * 3 L3 Pairs	135	552 (3 L3s)	230 (3 L3s)
Memory Transfer with 24 Cores (DRAM Limited)		700 (24 Cores)	209 (24 Cores)

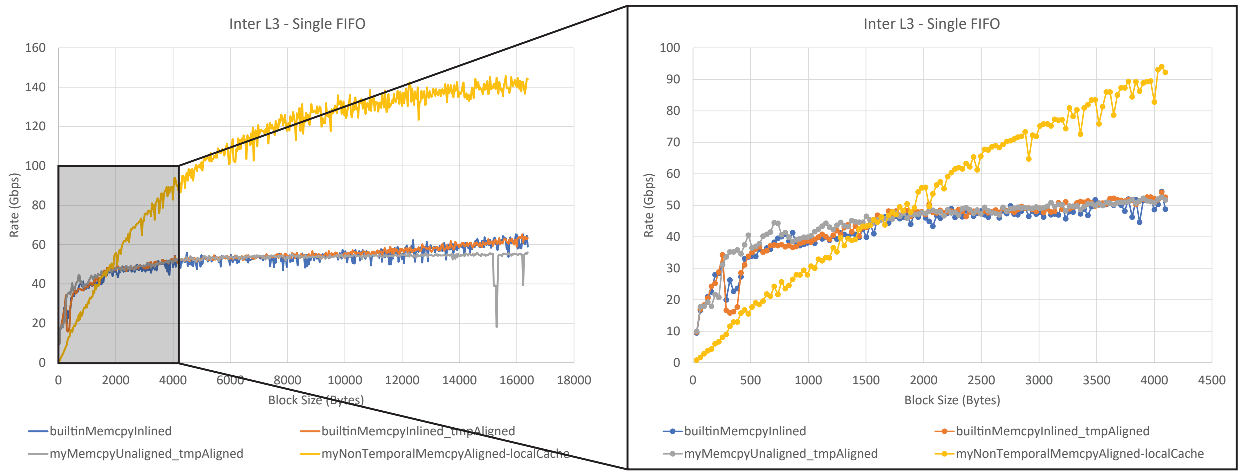
Table 7.1: Measured Throughput on Ryzen Threadripper 3970X with 3200 DDR4 - CPU Clock Limited to 3.7 GHz, NPS_AUTO, Harmonic Mean Over Block Sizes [3400, 4096] Bytes

read. These non-temporal instructions are also weakly ordered and write combining [104], presenting a significantly weaker consistency model than the TSO ordering provided for most x86_64 memory operations. As such, explicit use of memory fence instructions is required to force the memory consistency model expected by the FIFOs.

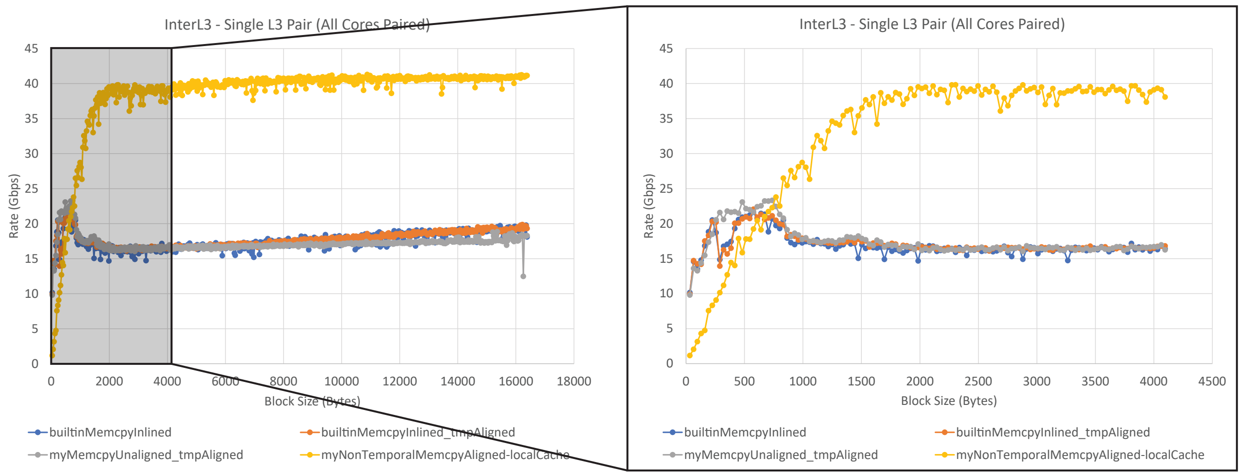
A version of the FIFO using non-temporal instructions was created and tested against other implementations. As expected, there was substantial performance degradation using non-temporal instructions when the communicating threads were within the same L3 domain (i.e., shared a common high-level cache). However, performance did improve for sufficiently large block sizes when the communicating threads resided in different L3 domains (shown in Figure 7.10). The crossover for when the non-temporal approach is better depends on the specific scenario but ranged from around block sizes of around 400 to 150 Bytes. These block sizes fall close to most of the block sizes in the Cyclops implementation making it uncertain if moving to non-temporal FIFOs would be beneficial. Currently, non-temporal FIFOs are not integrated into Laminar but could easily be added as a configuration option. However, control should be available on a per-FIFO basis as use of non-temporal instructions when threads in the same L3 domain are communicating results in significant performance loss.

7.5 Partition-Core Mapping

A significant result of the Laminar FIFO benchmarking analysis in section 7.4 is that the mapping of threads to cores can have a large impact on the overall system performance. Because the bandwidth is significantly lower and the latency is higher going between L3 domains versus within them, the performance of the overall system is typically maximized by trying to constrain as much communication as possible to occur within L3 domains. Even though Laminar has a fixed block size throughout the system, only being modified when entering clock domains, the block size is in terms of samples. Depending on the datatypes of

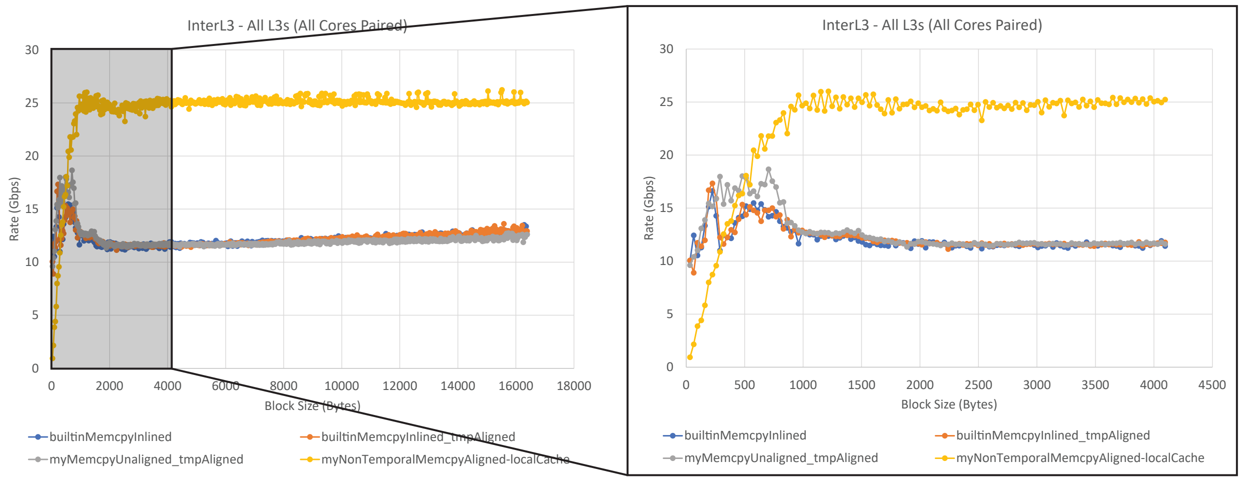


(a) Inter-L3 Single FIFO



(b) Inter-L3 Single L3 Pair

Figure 7.10: Non-Temporal FIFO Throughput Compared to Other FIFO Implementations



(c) Inter-L3 All L3s (Except 2 OS L3s) Paired

Figure 7.10: Non-Temporal FIFO Throughput Compared to Other FIFO Implementations

data being sent between threads, the block size in bytes varies, affecting the effective transfer rate.

Currently, the mapping of threads to specific cores is performed by the designer. To help the designer in analyzing the communication patterns between partitions, an analysis tool [183] was written using the NetworkX library [132]. The tool plots the different partitions, the connections between them, and the volume of data moving through the design per block. The designer uses this, along with the benchmarking results shown in section 7.4 to inform the proposed mappings. These mappings can be quickly compiled and tested, creating a design iteration loop that can be used to refine the placement. The communication analysis for Cyclops Rev 1.40 running on the AMD Ryzen 3970X is shown in Figure 7.11. While there are some mappings that appear suboptimal, they correspond well to the compute workload imbalances present in the design to create a fast overall system.

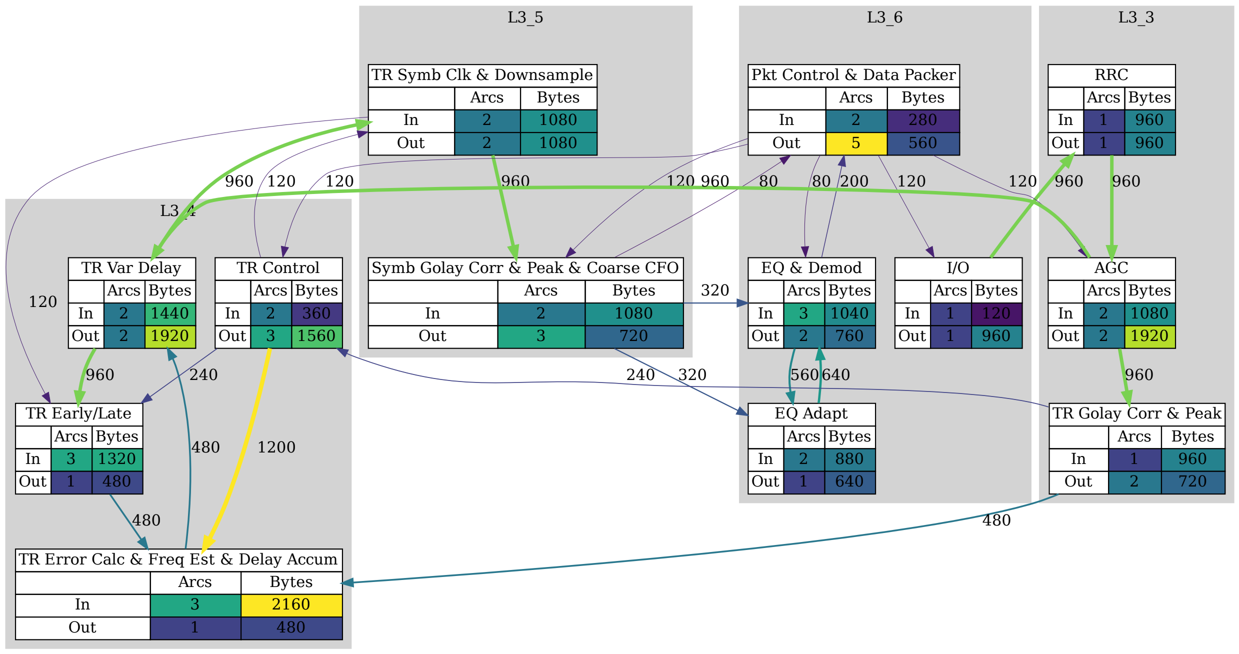


Figure 7.11: Communication Analysis of Cyclops Rev. 1.40 Receiver, Units: Bytes/Block

Chapter 8

Software Radio Co-Design Considerations

Chapters 5 - 7 discussed different methods employed by Laminar to optimize existing DSP designs. As these techniques were integrated and tested, it became clear that modifications to the source baseband design had the potential to improve performance on CPUs. While some of these techniques are also applicable to hardware designs, they can have an outsized impact when paired with the constraints imposed by the CPU platform. Each of the techniques discussed in this chapter were employed in revision 1.40 of the Cyclops receiver.

8.1 Ensuring Sufficient State in Loops

As was discussed in depth in chapter 7, blocked transfers are an important tool to amortize the fixed costs of a FIFO transfer. One consequence of operating on blocks is that initial state in FIFOs can only be formed by delays which are complete integer multiples of the block size being absorbed into the FIFO. When communication cycles exist among partitions in the design, at least one block of samples is required to avoid deadlock. However, multiple blocks of initial conditions are required to keep all partitions in the loop active at once. This is the same case as fixed tokens traveling around a loop in a Kahn Processes Network (KPN) [18] or synchronous dataflow design [11]. As a result, care must be taken to ensure that sufficient state exists in communication loops to avoid artificially limiting performance by forcing some partitions to sit idle while waiting for input data along one of the arcs in a loop.

A cycle analysis tool [183], was written to help analyze communication cycles within a design and to report how many blocks of state are available in each loop[1]. The summary analysis for Cyclops Rev. 1.40 is shown in Figure 8.1 while information about a single loop is shown in Figure 8.2. The minimum effective blocks of initial conditions per core (initial conditions in cycle / number of cores in cycle) for Cyclops Rev. 1.40 is one, in the

[1]Some of this information is also available textually as a report generated by Laminar.

tight Equalizer loop and one of the Timing Recovery loops. By inspecting these reports throughout the development process, Cyclops was modified to include additional state in loops, especially in slow changing non-critical control loops such as the reset loop.

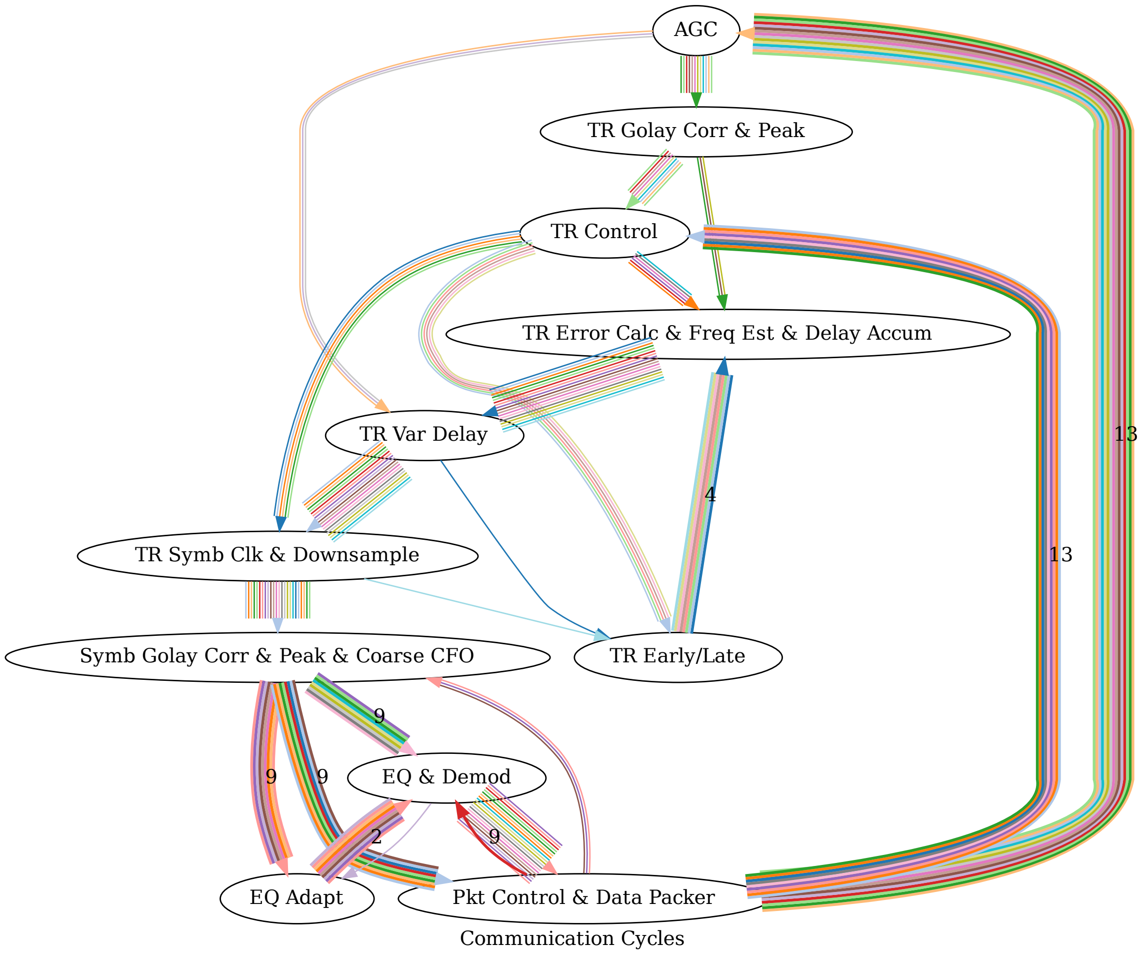


Figure 8.1: Cyclops Rev. 1.40 Receiver - Cycle Analysis Summary, Units: Blocks of Initial Conditions in FIFO, Each Color is a Separate Loop

8.2 Coarse/Fine Control Systems

Adding additional delay to loops in a DSP design can be challenging. Many loops in the design exist as part of feedback control systems whose dynamics change as additional

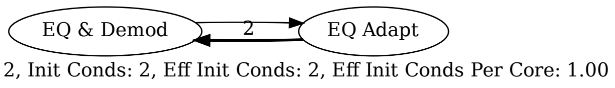

EQ & Demod
2
EQ Adapt
Nodes In Cycle: 2, Init Conds: 2, Eff Init Conds: 2, Eff Init Conds Per Core: 1.00

Figure 8.2: Cyclops Rev. 1.40 Receiver - Cycle Analysis for EQ Loop, Units: Blocks of Initial Conditions in FIFO

delay is inserted. This is a problem in both hardware and software. For example, multipliers often need to be pipelined in hardware/FPGAs to attain reasonable clock rates. However, the sheer amount of state required to accommodate blocked transfers in software presents a substantial challenge. One established solution to this problem in radio baseband design is the use of control systems separated into coarse and fine stages. The coarse phases are often feedforward with the error signal being estimated from the incoming data, a correction computed based on the estimate, and the correction being applied to the data. One downside of feedforward estimation is that any error in the estimate will propagate to error in the corrected values. The upside is that feedforward systems are easily pipelined or, in the case of Laminar generated designs, can be spread across multiple cores that operate in parallel without modification[2]. Feedback systems have the advantage that they can adapt their correction to the remaining error detected after the correction is applied. Their self-correcting properly allows them to adapt to changing circumstances. The downside is that feedback control systems, by their very nature, require a feedback path from the post-correction values, through the error estimation/correction computation block and back to the point where the correction is applied. Adding delay to the feedback path without re-tuning can cause the system to become unstable. One method to address this is to reduce the loop bandwidth, which slows the rate of adaptation. A complication of this approach is that it can cause the system to take a long time to correct large errors and to be unable to adapt to rapidly changing errors.

Combining a coarse feedforward correction with a fine feedback correction (as shown in Figure 8.3) helps alleviate some of the problems of both approaches when used in isolation. The feedforward correction can potentially provide a solution quickly. However, residual error out of the coarse correction is possible, especially if the feedforward estimation relies on some component of the underlying signal (such as the preamble) that is not always available. The feedback control system can correct any residual error that makes it through the coarse correction block. Assuming that the coarse correction block did a reasonable job correcting the error, the feedback control system's loop bandwidth requirements are reduced, facilitating the required pipelining if the feedback control loop has sufficient work

[2]The FIFOs behave similarly to pipeline stages from an execution standpoint, allowing each partition in the feed-forward segment to execute in parallel in steady state operation. With no loops among partitions, the initial state in FIFOs is not a limiting factor.

that it needs to be split across partitions.

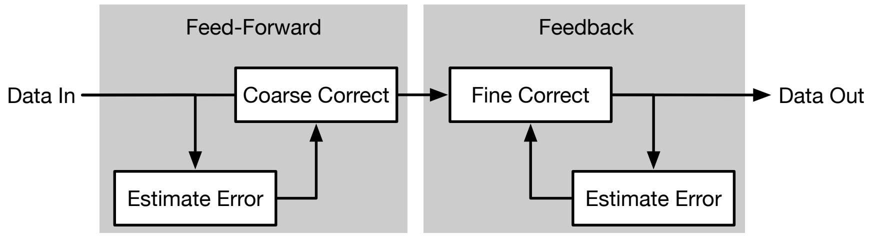


Figure 8.3: Coarse/Fine Control System Structure

The coarse/fine control system design was used to update the Timing Control subsystem which had became a significant bottleneck in the design. The timing error is estimated during the preamble in a feedforward fashion which is then held after the preamble has concluded. After the preamble, the system moves to a feedback scheme that corrects residual error from the earlier correction via an early-late detector. The same technique is also used in the equalizer where the CFO segment of the preamble is used to perform the initial training of the LMS algorithm. In the header and payload portions of the packet, the equalizer changes to a decision-directed feedback mode. Finally, the CFO correction now effectively uses the coarse/fine technique by performing a coarse feedforward estimation during the preamble, which is used to drive an NCO. The fine correction is effectively performed by the equalizer. For more details on how these blocks operate, see subsection 2.2.2.

8.3 Blocked and Block-Aware Algorithms

A related technique to the coarse/fine control system technique is to aggregate operations into blocks. By explicitly operating on blocks of data, it may be possible to obtain better performance than what is available when relying on the automated sub-blocking provided by Laminar alone. An example of this technique is the Cyclops equalizer, which uses Block-LMS [93], [94]. The algorithm gathers error estimates into a block of n symbols from which a new set of correction coefficients are computed every n samples. This is accomplished internally by placing the coefficient computation in a downsample domain with a *repeat* output. Thanks to the compressed datatype, the FIR portion of the equalizer can use constant coefficients for a block's worth of inputs, reducing the required work to load new coefficients for each input. Also, the blocked computation allows the coefficient correction accumulation to be heavily vectorized even though it relies on a loop with state to hold the previous coefficient values. Instead of the coefficient accumulator running every cycle and passing its output to the correction FIR filter only every nth symbol, the accumulator is only modified every n iterations with the aggregate correction coming from the block of n symbols. The block size of the Block-LMS algorithm is set to be the same as the sub-blocking length to facilitate these improvements.

Keeping the eventual sub-blocking length in mind can also help extract the most performance from the Laminar sub-blocking pass. For example, the AGC originally had a tight feedback path which forced the entire loop to be contained in a single sub-blocking domain. By slightly increasing the delay in the loop to a multiple of the sub-blocking length, the sub-blocking pass was able to break the loop apart and sub-block each component, leading to a significant performance gain.

8.4 Fixed Decimation

Early versions of Cyclops made extensive use of enabled sub-systems. They formed the basis for the symbol domain and were used in other locations in an attempt to reduce workload. As the Laminar compiler became more sophisticated, the enabled subsystem became a source of performance problems.

Prior to the introduction of explicit downsample domains in Laminar, the Timing Recovery system generated a strobe signal which was used to indicate when a valid symbol was available at the output. This strobe signal was used to drive an enabled sub-system surrounding the symbol domain of the design. At the time, the enabled subsystem provided a degree of performance improvement because the logic contained within it would not run for every sample. However, because Laminar transacts in fixed-sized (in terms of samples) blocks and it was not known a priori which samples would be processed by the symbol domain, all samples were passed from the Timing Recovery block to CPU cores handling the symbol domain. With n samples per symbol, this resulted in approximately[3] $n-1$ unused samples being communicated per symbol on average. This is a significant inefficiency that, when paired with limited core-core rates, can result in noticeable performance overhead. By moving to a static decimation, the compiler now knows a priori how many samples will make it through the downsample operator for each block. This allows only the samples which will be processed by the symbol domain components of the design to be communicated between cores.

As discussed in subsection 6.2.4, enabled subsystems can also interfere with vectorization. Since the decision on when the subsystem executes is determined on a sample-by-sample basis, some form of masked vector operation is required. Currently, Laminar's automated sub-blocking pass constrains enabled subsystems to be contained within a single sub-blocking domain due to the complexities brought on by the masking requirement. Replacing the symbol domain enabled subsystem with a fixed decimation clock domain allowed sub-blocking to operate on structures contained within it, resulting in performance gains across many of the partitions in the symbol domain.

[3]Timing frequency offset can cause slightly more or less samples being unused on average.

8.5 Use of Floating Point

The original Cyclops design was aimed at FPGAs which did not include floating-point hardware units. To better fit the target platform, numbers were represented in fixed-point form. This allowed the design to take advantage of the integer multipliers in the DSP units and the specialized integer adder hardware. By contrast, the x86_64 CPUs used in this project not only have floating-point units; they have more floating-point multipliers than integer multipliers. Combined with the fact that floating-point automatically handles the re-normalization operations, which would otherwise need to be performed with shift and masking instructions after a subset of fixed-point operations, made floating-point arithmetic especially appealing. It was generally a simple process converting from 14 and 16-bit fixed-point numbers to 32-bit single-precision floating-point due to an increase in resolution from those types. However, there were components including a specialized rolling average[4] that reacted poorly to the numeric properties of floating-point arithmetic. These blocks were removed and replaced with more conventional alternatives that were insensitive to floating-point associativity.

8.6 Other Laminar Modifications

In addition to the changes described above, Laminar underwent additional modifications to improve its overall performance. Additional changes include:

- Leveraging the delay insensitivity of reset and parameter freeze lines to introduce additional state into control feedback paths.
- Removing debugging I/O ports when possible, to reduce the required communication between cores.
- Replacing Stateflow FSMs with implementations more suitable to Laminar optimization passes.
- Replacing Ln and Exp operators with low-resolution Lookup Tables (LUTs) to reduce computation workload.
- Removing enabled subsystems from Timing Recovery Early-Late and AGC to help facilitate vectorization with sub-blocking.

[4]The rolling average used an accumulator and input history to subtract elements that exited the rolling average window. When a large input entered the system, the accumulator would be pushed out of the resolution of smaller samples. New samples would have no impact to the accumulator but would be recorded in the sample history. After the large element was subtracted from the accumulator, the smaller elements would still be in the history. These smaller items would eventually be subtracted, resulting in a biased estimate.

- Reducing Root Raised Cosine (RRC) filter order.[5]
- Reducing LMS equalizer length.

8.7 Increased Complexity Modifications

The success of the Laminar optimizations and Cyclops design modifications made it possible introduce some additional logic to the Cyclops radio to improve its efficiency and add additional features. One of the largest changes was a redesign of the Timing Recovery block to be more reliable. The changes to timing recovery, along with the addition of a decision feedback mode to the equalizer, allowed the implementation of 256 QAM support. This increased the maximum number of bits per symbol from 4 (16 QAM) to 8 (256 QAM), quadrupling the maximum data rate possible in the payload portion of packets. Changes to the Timing Recovery block also made it possible to change from operating with 4 samples per symbol to operating with 3 samples per symbol. This change increased the workload in the symbol domain for a given sample rate but also increased the bandwidth used from 1/4 to 1/3 of the total captured bandwidth.

[5]If increased order is required resulting in the RRC becoming the bottleneck, the RRC can be easily partitioned thanks to relatively little interaction with other partitions and being purely feedforward.

Chapter 9

Experimental Results

One objective of the various Laminar optimizations and Cyclops baseband revisions discussed throughout this document was the demonstration of a high-performance software radio. This chapter details the culmination of these techniques in the Revision 1.40 Cyclops demo and the framework used to evaluate performance.

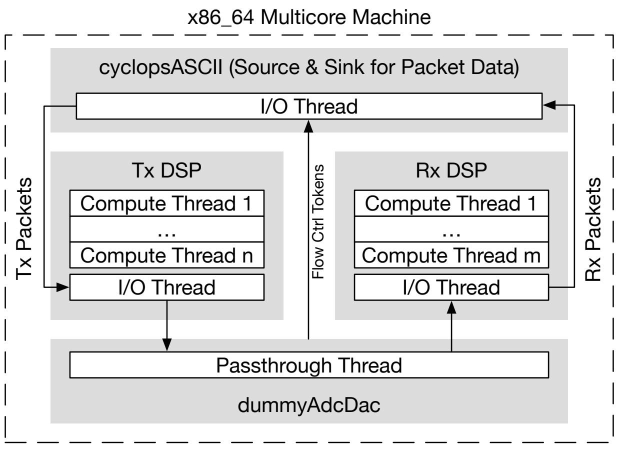
9.1 Cyclops Demo

In isolation, radio baseband signal processing does not perform useful work. Only when data is available at the Tx, transmitted as a frame, and decoded by the Rx is the radio performing its desired function. It is therefore important when demonstrating the generated designs to produce relevant stimuli capable of driving the Tx and Rx and to provide a mechanism to interpret the results. To accomplish this task, a suite of applications and scripts was written [184] to provide a unidirectional end-to-end demonstration of the Cyclops radio and to report/record the performance achieved while running. This suite, referred to as `cyclopsDemo`, is composed of seven main components:

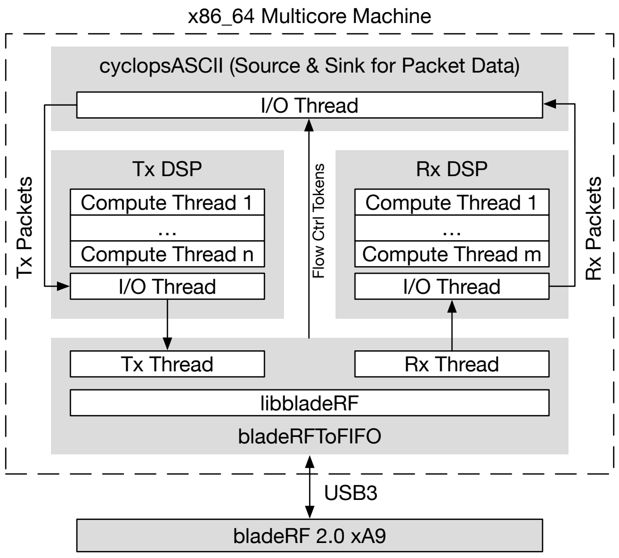
- The Laminar generated Cyclops Tx Baseband Application.
- The Laminar generated Cyclops Rx Baseband Application.
- cyclopsASCIILink [185]: An application which generates valid Cyclops Frames for Tx and displays received Cyclops frames.
- bladeRFToFIFO [186] / uhdToPipes [187] / dummyAdcDac: A program which handles communication with an external radio platform or acts as an internal passthrough from Tx to Rx.
- Web Based Telemetry Dashboard: To showcase real-time performance in a human readable form.

- Scripts to facilitate the building and cleanup of the demo applications, including driving the Laminar compiler.
- Scripts to run the demo either interactively or in a headless environment with performance results collected.

The various applications are linked via named POSIX pipes or shared-memory FIFOs [146] that present an interface similar to that of POSIX pipes. Relatively early in the project, it was determined that the POSIX pipes presented a performance bottleneck. All recent demonstrations use shared-memory FIFOs. The structure of the demo is shown in Figure 9.1 with cyclopsASCIILink acting as the source for the Tx baseband and the sink for the Rx baseband. The Tx and Rx basebands are each connected to a program which either facilitates communication with an external radio frontend or passes samples from the Tx to the Rx. bladeRFToFIFO and uhdToPipes both facilitate connections to bladeRF and USRP hardware, respectively. dummyAdcDac provides a mechanism to test the Cyclops baseband without an external ADC/DAC connected. While the execution rate can be limited by the speed of the ADC/DAC for the bladeRFToFIFO and uhdToPipes drivers, dummyAdcDac presents no such limitation and passes samples from Tx to Rx as fast as possible. When reporting the speed throughout this document, the dummyAdcDac driver was typically used as it allows the speed to become limited by the capability of the signal processing rather than the capability of the radio frontend.



(a) When Using Dummy ADC/DAC Loopback



(b) When using BladeRF Frontend

Figure 9.1: Cyclops Demo Architecture

cyclopsASCIILink provides both the stimulus to the Tx baseband as well as the sink for the Rx. Due to the fixed block size in the Laminar execution model, framing for the Tx and de-framing for the Rx both occur in the cyclopsASCIILink application rather than in the generated DSP code. When generating the frames, it pulls sections from an ASCII text string ([188] from Project Gutenberg used as the example text). Because the performance of the Laminar generated signal processing applications is what is under investigation, random frames are produced before data starts streaming to the Tx baseband to avoid artificially introducing a bottleneck in the cyclopsASCIILink test application. After a frame is completely sent to the Tx baseband, the next frame is randomly selected from the pre-assembled frames. The duty cycle of frames sent to the Tx (the amount of blank space between packets) can be set as a parameter to cyclopsASCIILink. For most Cyclops Rx results presented in this document, the duty cycle was set to 100% or back-to-back packets.

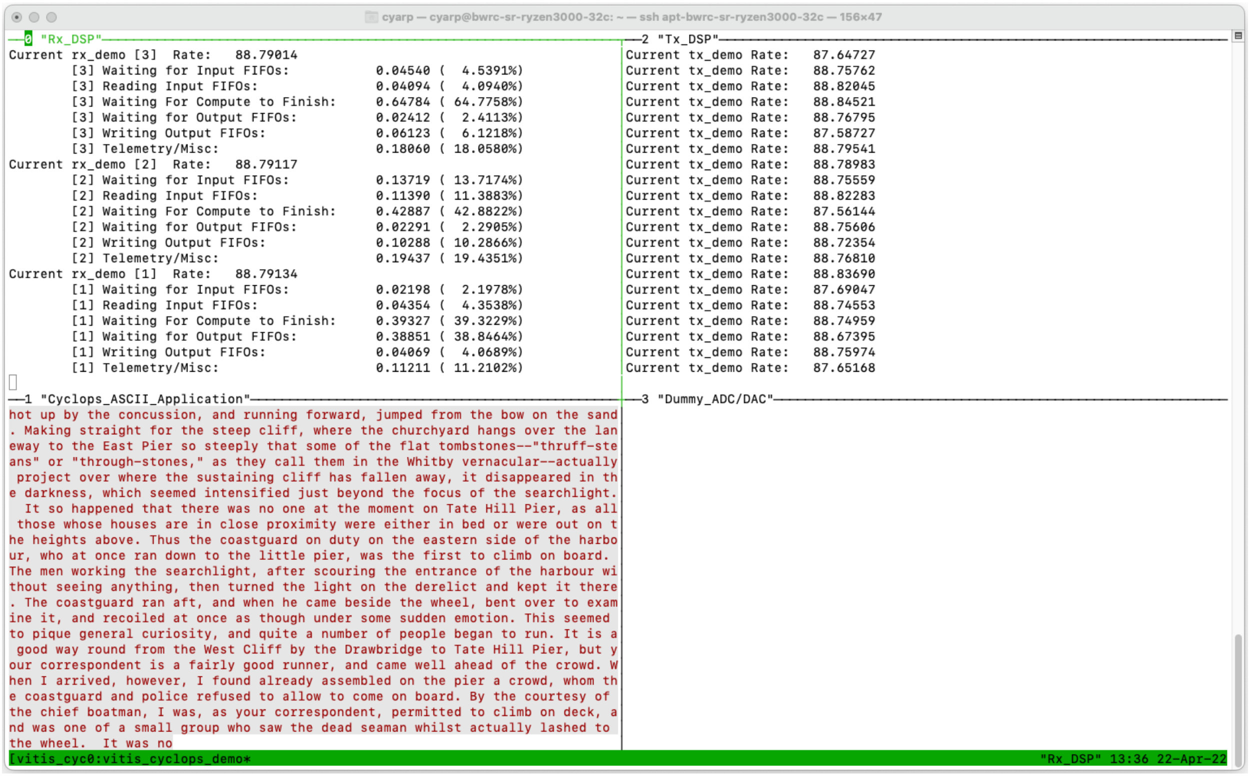
One of the scripts included in `cyclopsDemo`, `build.sh`, manages:

- Compiling the Laminar compiler.
- Generating the Cyclops Tx and Rx baseband sources using Laminar.
- Compiling the generated Tx and Rx basebands into executables.
- Compiling cyclopsASCIILink, bladeRFToFIFO, uhdToPipes, and dummyAdcDac.

A script, `runDemoTmuxSharedMem.sh`, sets runtime options for the demo and starts the various components in order. The programs are run from within a tmux session, which allows interactive inspection of the different programs involved in the demo (as shown in Figure 9.2). An optional web-based dashboard [189] is provided to help visualize the telemetry emitted by the Laminar generated executables in real time. An example of the interface is shown in Figure 9.3.

In general, demos and tests are conducted after a clean reboot of the system to return it as close to a known state as possible. To facilitate this, a simple job queue and runner [190] was written which reboots the system between jobs. This provided a mechanism for running jobs in a repeatable way after enforcing a system reboot. When operating in a headless non-interactive mode, the `runDemoAndCollectResultsSharedMemory.sh` script is executed which runs the demo for a specific amount of time (3 minutes in most configurations) and then saves results and run information for later retrieval. When booting, the system uses a series of kernel options provided by grub to reduce OS jitter and to present as little interference as possible to the signal processing and test applications. An example of the grub configuration is shown in section B.2. One of the most important options is `isolcpus` which specifies a list of CPUs to be isolated from the OS scheduler. All except one core are isolated with one core left to handle regularly scheduled OS tasks. Tasks are scheduled on the isolated cores by specifically setting the process CPU affinity masks. For the Cyclops revision 1.40 demo, the number of cores assigned to each process are:

- Cyclops Tx Baseband: 2 Compute + 1 I/O (3 Total).
- Cyclops Rx Baseband: 12 Compute + 1 I/O (13 Total).



```
cyarp — cyarp@bwrc-sr-ryzen3000-32c: ~ — ssh apt-bwrc-sr-ryzen3000-32c — 156×47
```

```
0 "Rx_DSP"
Current rx_demo [3]  Rate:   88.79014
        [3] Waiting for Input FIFOs:          0.04540 (  4.5391%)
        [3] Reading Input FIFOs:              0.04094 (  4.0940%)
        [3] Waiting For Compute to Finish:    0.64784 ( 64.7758%)
        [3] Waiting for Output FIFOs:         0.02412 (  2.4113%)
        [3] Writing Output FIFOs:             0.06123 (  6.1218%)
        [3] Telemetry/Misc:                   0.18060 ( 18.0580%)
Current rx_demo [2]  Rate:   88.79117
        [2] Waiting for Input FIFOs:          0.13719 ( 13.7174%)
        [2] Reading Input FIFOs:              0.11390 ( 11.3883%)
        [2] Waiting For Compute to Finish:    0.42887 ( 42.8822%)
        [2] Waiting for Output FIFOs:         0.02291 (  2.2905%)
        [2] Writing Output FIFOs:             0.10288 ( 10.2866%)
        [2] Telemetry/Misc:                   0.19437 ( 19.4351%)
Current rx_demo [1]  Rate:   88.79134
        [1] Waiting for Input FIFOs:          0.02198 (  2.1978%)
        [1] Reading Input FIFOs:              0.04354 (  4.3538%)
        [1] Waiting For Compute to Finish:    0.39327 ( 39.3229%)
        [1] Waiting for Output FIFOs:         0.38851 ( 38.8464%)
        [1] Writing Output FIFOs:             0.04069 (  4.0689%)
        [1] Telemetry/Misc:                   0.11211 ( 11.2102%)
```

```
2 "Tx_DSP"
Current tx_demo Rate:   87.64727
Current tx_demo Rate:   88.75762
Current tx_demo Rate:   88.82045
Current tx_demo Rate:   88.84521
Current tx_demo Rate:   88.76795
Current tx_demo Rate:   87.58727
Current tx_demo Rate:   88.79541
Current tx_demo Rate:   88.78983
Current tx_demo Rate:   88.75559
Current tx_demo Rate:   88.82283
Current tx_demo Rate:   87.56144
Current tx_demo Rate:   88.75606
Current tx_demo Rate:   88.72354
Current tx_demo Rate:   88.76810
Current tx_demo Rate:   88.83690
Current tx_demo Rate:   87.69047
Current tx_demo Rate:   88.74553
Current tx_demo Rate:   88.74959
Current tx_demo Rate:   88.67395
Current tx_demo Rate:   88.75974
Current tx_demo Rate:   87.65168
```

```
1 "Cyclops_ASCII_Application"
hot up by the concussion, and running forward, jumped from the bow on the sand
. Making straight for the steep cliff, where the churchyard hangs over the lan
eway to the East Pier so steeply that some of the flat tombstones--"thruff-ste
ans" or "through-stones," as they call them in the Whitby vernacular--actually
 project over where the sustaining cliff has fallen away, it disappeared in th
e darkness, which seemed intensified just beyond the focus of the searchlight.
  It so happened that there was no one at the moment on Tate Hill Pier, as all
 those whose houses are in close proximity were either in bed or were out on t
he heights above. Thus the coastguard on duty on the eastern side of the harbo
ur, who at once ran down to the little pier, was the first to climb on board.
The men working the searchlight, after scouring the entrance of the harbour wi
thout seeing anything, then turned the light on the derelict and kept it there
. The coastguard ran aft, and when he came beside the wheel, bent over to exam
ine it, and recoiled at once as though under some sudden emotion. This seemed
to pique general curiosity, and quite a number of people began to run. It is a
 good way round from the West Cliff by the Drawbridge to Tate Hill Pier, but y
our correspondent is a fairly good runner, and came well ahead of the crowd. W
hen I arrived, however, I found already assembled on the pier a crowd, whom th
e coastguard and police refused to allow to come on board. By the courtesy of
the chief boatman, I was, as your correspondent, permitted to climb on deck, a
nd was one of a small group who saw the dead seaman whilst actually lashed to
the wheel.  It was no
```

```
3 "Dummy_ADC/DAC"
```

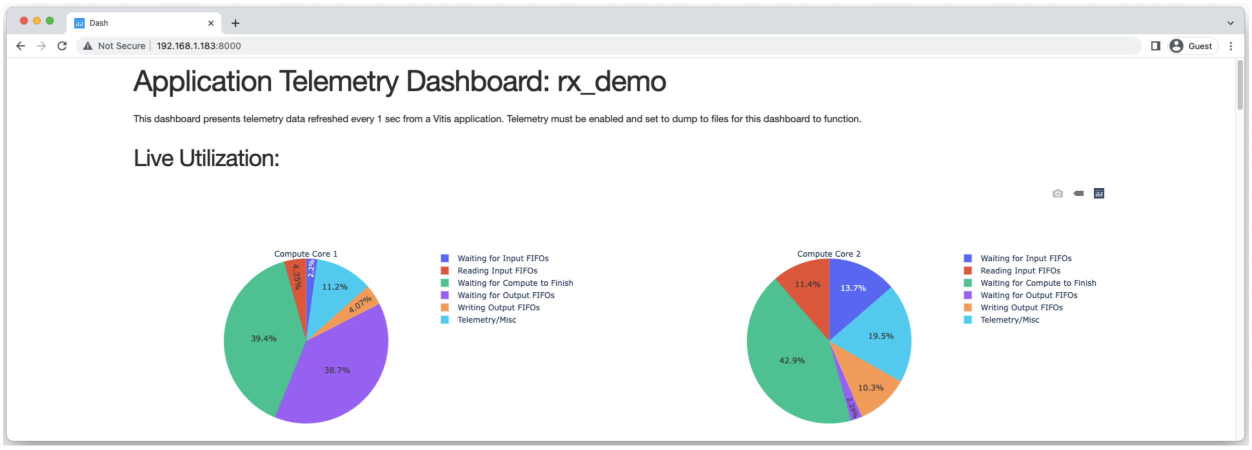
```
[vitis_cyc0:vitis_cyclops_demo*                                  "Rx_DSP" 13:36 22-Apr-22
```

Figure 9.2: Cyclops 1.40 Tmux Demo - Dummy ADC/DAC

- cyclopsASCIILink: 1 Total.
- Radio I/O: dummyAdcDac - 1 Total, bladeRFToFIFO - 2 Total, uhdToPipes - 3 Total.
- OS + Misc: 1 Total.

In addition to the non-default kernel options, a series of scripts [191] are run on boot via a `systemd` service to set runtime configurable options. The scripts:

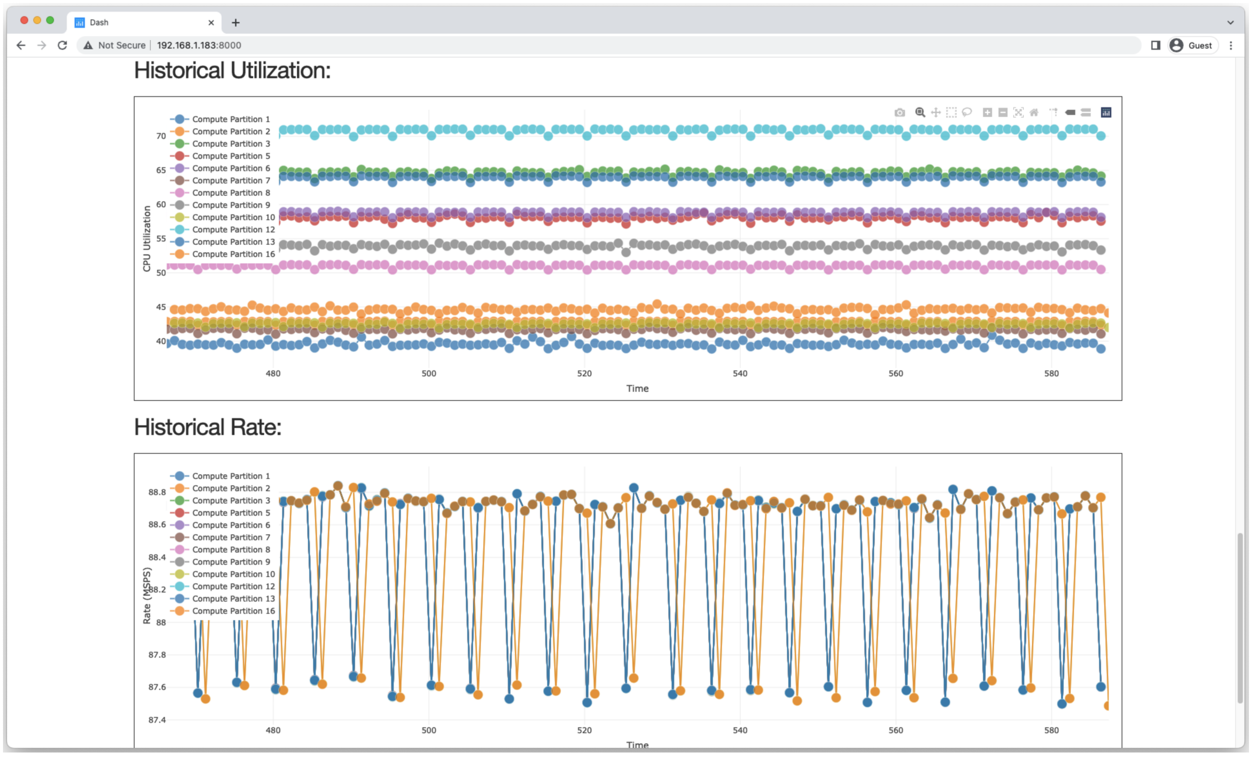
- Set the CPU governor to "userspace" control and set the desired frequency to a constant value (the stock clock speed of the CPU for experiments in this document).
- Disable boost mode on all CPUs.
- Set the real-time (RT priority) run time to unlimited.
- Disable the IRQ balance service.
- Move interrupts off isolated cores (when possible) by setting. smp_affinity_masks.
- Offline and online isolated CPUs to migrate kernel tasks off them.


Application Telemetry Dashboard: rx_demo

This dashboard presents telemetry data refreshed every 1 sec from a Vitis application. Telemetry must be enabled and set to dump to files for this dashboard to function.

Live Utilization:

(a) Live Workload Breakdown Per Thread


Historical Utilization:

Historical Rate:

(b) Historical Compute Utilization (Percent of Workload Spent Computing) and Rate

Figure 9.3: Cyclops 1.40 Demo - Live Telemetry Dashboard

For more information on these techniques to reduce OS jitter see the discussion in section 7.3 and [141], [142], [176]–[178].

9.2 Cyclops Performance

Detailed results of running cyclopsDemo with revision 1.40 of the Cyclops receiver on a Ryzen Threadripper 3970X operating at 3.7 GHz are shown in Figure 9.4. In this run, Laminar was configured to insert additional telemetry collection in the Cyclops Rx threads which included how much time is spent in various stages of execution (on average). This is accomplished by inserting clock reads at various points in the outer thread function. At the end of each phase, the difference between the timer values is taken and is added to an accumulator for the time spent in that given phase of execution. Compiler barriers are placed around the timers to prevent the re-ordering of the timer instructions. However, there is some slop in the timer measurement with the adjacent operations due to out-of-order execution in the CPU. Periodically, the counters are written to a log file. In addition to the localized timer reads, the time between telemetry reports is also tracked. The difference between the sum of the individual execution region times and the overall time is reported as *Telem/Misc* as this unaccounted time should, in theory, be the time spent collecting and writing telemetry. Plots like Figure 9.4 presented throughout this document are stacked bar plots generated by an analysis script in [192] and plotted via Excel.

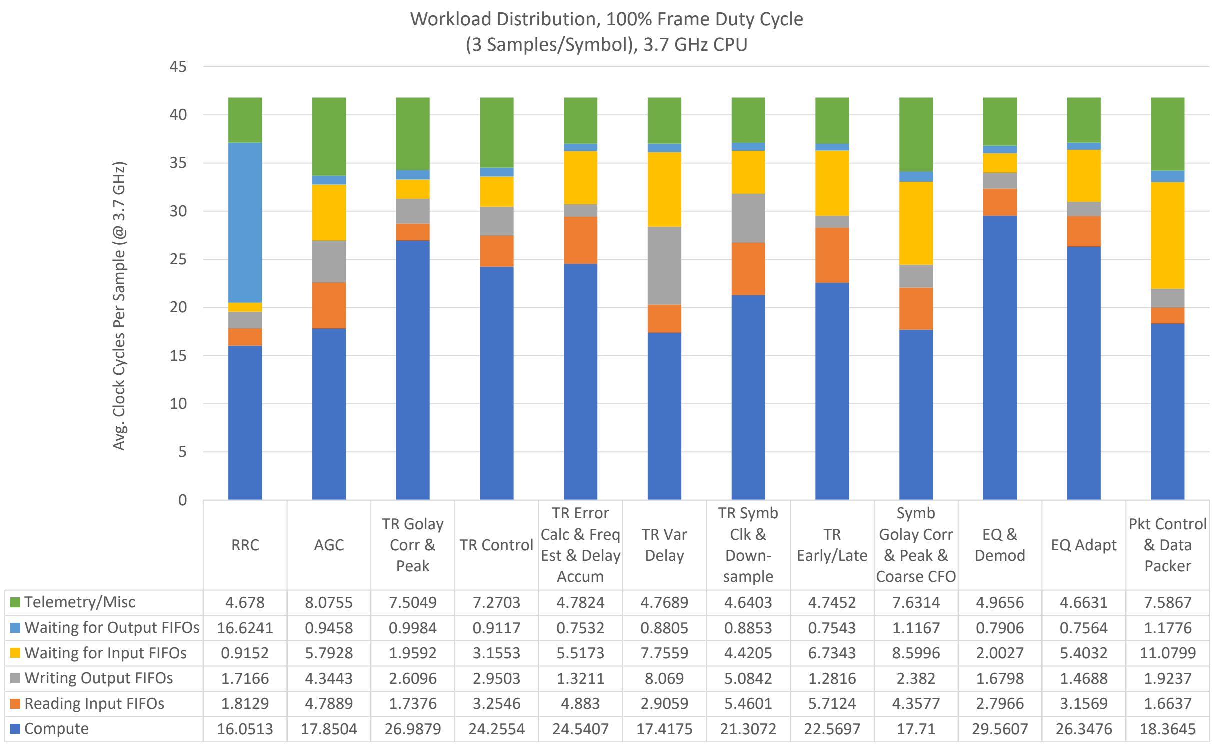
The results for Cyclops revision 1.40 are exceptionally encouraging, with the average execution speed reaching 88.52 Msps with 100% frame duty cycle using 12 compute cores and one I/O for the receiver DSP. When the level of reported telemetry is reduced to just report the achieved rate in the I/O thread, the average execution rate increases to 104.5 Msps. This is a giant leap forward compared earlier versions of the Cyclops baseband which ran at approximately 2 Msps with 4 samples per symbol on a single core (see Figure 5.8).

Telemetry Lvl	Msamp/s	Msymb/s	Mbps Payload (256 QAM)
Fine Grain	88.52	29.51	236.1
I/O Only	104.5	34.83	278.7

Table 9.1: Cyclops 1.40 Performance, Dummy ADC/DAC

9.2.1 Using a Real Radio Frontend: bladeRF 2.0 xA9

An earlier version of Cyclops supporting 256 QAM was demonstrated at the Berkeley Wireless Research Center (BWRC) Fall 2021 Research Retreat using a bladeRF 2.0 xA9 [193] radio interface. The bladeRF xA9 was connected to the host computer using USB3 and was configured for an external loopback via cable and attenuator from one Tx port to



	RRC	AGC	TR Golay Corr & Peak	TR Control	TR Error Calc & Freq Est & Delay Accum	TR Var Delay	TR Symb Clk & Down-sample	TR Early/Late	Symb Golay Corr & Peak & Coarse CFO	EQ & Demod	EQ Adapt	Pkt Control & Data Packer
🟩 Telemetry/Misc	4.678	8.0755	7.5049	7.2703	4.7824	4.7689	4.6403	4.7452	7.6314	4.9656	4.6631	7.5867
🟦 Waiting for Output FIFOs	16.6241	0.9458	0.9984	0.9117	0.7532	0.8805	0.8853	0.7543	1.1167	0.7906	0.7564	1.1776
🟨 Waiting for Input FIFOs	0.9152	5.7928	1.9592	3.1553	5.5173	7.7559	4.4205	6.7343	8.5996	2.0027	5.4032	11.0799
⬜ Writing Output FIFOs	1.7166	4.3443	2.6096	2.9503	1.3211	8.069	5.0842	1.2816	2.382	1.6798	1.4688	1.9237
🟧 Reading Input FIFOs	1.8129	4.7889	1.7376	3.2546	4.883	2.9059	5.4601	5.7124	4.3577	2.7966	3.1569	1.6637
🟦 Compute	16.0513	17.8504	26.9879	24.2554	24.5407	17.4175	21.3072	22.5697	17.71	29.5607	26.3476	18.3645

Figure 9.4: Cyclops 1.40 Performance, Dummy ADC/DAC, Breakdown Telemetry Collected

one Rx port. While the bladeRF board is capable of a 61.44 Msps sample rate [193], it is limited to an aggregate of 80 Msps across all ports as reported by libbladeRF [194]. For a single board operating in full duplex mode, which was required for the demo, this limited the achievable rate to 40 Msps (40 Msps allocated to the Tx channel and 40 Msps allocated to the Rx channel). The ADC/DAC rate of the bladeRF limited the rate of the Cyclops baseband to 40 Msps, which was reported by the telemetry retrieved from the Tx and Rx baseband applications.

This the sample rate limitation can be sidestepped by using two independent bladeRF boards connected to separate USB3 interfaces on the system to avoid sharing bandwidth between the boards. However, doing this would require fine-tuning the bladeRF FIFO transaction properties to avoid underflow and overflow.

Some RF impairments were encountered when testing with the bladeRF 2.0 out of the box. These impairments included I/Q imbalance, DC offset, and increased noise and reduced gain when sampling at high rates. The DC offset shifts the constellation while the I/Q imbalance turns the normally square constellation and warps it to resemble an elongated rectangle or parallelogram. These distortions are hard for the EQ to correct and should ideally be handled by calibrating the RF frontends. While some degree of calibration is possible using an external loopback, the use of lab quality signal generators is best as it provides known signals to calibrate against. A note from the bladeRF vendor shows the effect that proper DC offset and I/Q calibration have on the signal [195]. IQ imbalance compensation is performed in bladeRFToFIFO if provided with calibration information for the specific bladeRF board. The RFIC the bladeRF is based on (AD9361) also requires a set of internal filters to be properly set to provide the highest quality signal [196], [197]. By modifying the filter settings used by the libbladeRF driver, the noise when capturing larger bandwidths was reduced. However, there was still a reduction in overall gain which is a problem when operating with the imited resolution ADC. As a result of the impairments, Cyclops performs best on the bladeRF when operating in 16QAM mode.

9.2.2 Performance with Simultaneous Multi-Threading (SMT)

To achieve the high levels of performance desired in this project, a significant emphasis has been placed on presenting the CPU with enough independent work to fully leverage its parallel resources. This has taken the form of Laminar compiler optimizations and baseband co-design. One area which has remained challenging is the overlap of communication and computation. Multiple methods were discussed in section 7.2 with none providing the performance boost desired. One potential method which had not yet been explored was the use of the Simultaneous Multi-threading (SMT) CPU feature, discussed in subsection 3.1.5. SMT allows a core which would otherwise be unable to fill schedule slots with useful work from a given program to execute instructions from another program. This can apply to both arithmetic instructions as well as memory access instructions.

For most of this project, SMT was disabled as it can introduce contention for shared resources such as the caches, register files, and certain scheduling resources[1]. For threads which provide sufficient Instruction Level Parallelism (ILP) to keep the CPU compute resources busy, enabling SMT and assigning another compute heavy thread to the same core would probably not provide much benefit and could potentially reduce performance due to resource contention. However, with the standard FIFO implementation described in section 7.1, each compute thread stalls while waiting for FIFO communication to occur. Because future compute operations depend on the FIFO data being retrieved, they cannot be executed, leaving the compute resources idle. With SMT enabled and two threads scheduled to run on the same physical CPU (scheduled on the two logical CPUs aliased to the same physical core), it is possible for one thread to continue executing compute instructions while the other is stalled waiting for FIFOs.

To test the effect of SMT on running multiple instances of the radio on the same CPU, the cyclopsDemo was modified to run two instances of the baseband simultaneously [198]. The partitions of each instance were mapped to logical CPUs such that the identical partition from each instance would run on the same physical core. For example, the RRC of both instances ran on physical core 12. The cyclopsASCIILink and dummyAdcDac instances were

[1]See [17] for information on what resources are statically partitioned and which are shared when SMT mode is enabled on the Ryzen Threadripper 3970X.

mapped to different physical cores to avoid any additional contention in these programs becoming a potential bottleneck. Due to the different mapping, the performance of the individual instances can be affected. The two instances were benchmarked running solo as well as operating simultaneously. The results are shown in Table 9.2. If no overlap (arithmetic or memory) was achieved, one would expect that the two instances would operate at half the rate, yielding the same aggregate rate as a single instance. However, the results showed a higher aggregate rate with the two instances running on the same cores with SMT. The improvement was 16.7% when fine telemetry collection was enabled and 13.12% when only the I/O thread rate was collected. Looking at the bottleneck thread, part of the EQ, the expected performance improvement if FIFO reading / writing overlapped with the computation (with fine-telemetry collection enabled) was 15%. The result of this test suggests that some degree of overlap occurred. The fact that the performance improvement was close to what was expected with communication/computation overlap also suggests that, at least in the bottleneck thread, there was sufficient ILP to generally keep the core busy. With these results, additional research on the use of SMT, including its use in a single instance of the baseband, is probably warranted.

	Telemetry Breakdown	I/O Telem Only
Solo Instances		
Inst1	88.05806206	103.4664566
Inst2	88.41301277	102.6829808
HARMEAN Avg Rate	88.23518044	103.0732299
Dual Instances - Simultaneous		
Inst1	51.89351776	58.1872536
Inst2	51.25444853	58.40461047
Aggregate Rate	103.1479663	116.5918641
Percent Improvement	**16.90%**	**13.12%**

Table 9.2: Performance of Two cyclopsDemo Instances Running Simultaneously on the Same Physical Cores with SMT Enabled

9.3 Projecting Multi-Instance Performance

When combined with the multi-channel technique discussed in section 4.3, there is the potential to increase the aggregate throughput of the system beyond 278.7 Mbps in the payload. The number of basebands that can fit on a given platform depends both on the number of cores available as well as assumptions about the number of cores which need to be allocated to non-DSP tasks such as I/O or running a demo. As implemented, the current Cyclops demo requires at least 19 cores to run the complete transceiver (Tx and Rx), the local loopback, the stimulus generation and reporting, and the operating system. Because of the discrete nature of cores, the existing radio cannot be simply scaled to run on the remaining CPUs, it would need to be re-partitioned. However, there are some potential optimizations which could be made including merging the Tx and Rx I/O threads into a single thread. In a production system, an application such as cyclopsASCIILink may not be required[2]. It also may be possible for the Tx and Rx to be split across two different systems (ex. in a unidirectional link with limited feedback) or across two sockets in a multi-CPU server. Depending on the assumptions, rough projections of the achievable data rates on different CPUs are shown in Table 9.3. This assumes that the performance of a single instance scales perfectly with the change in stock clock rate. It *does not* include improvements provided by the Zen 3 microarchitecture in the Epyc processors shown in the table. It is likely that a single instance of the radio will achieve higher performance on those platforms. Note that the expected performance is best on the 64 core systems despite them having a lower stock clock rate. This is both a function of the fixed number of cores required by this version of the Cyclops baseband as well as the additional compute capability of these 64-core parts. Note that, while the clock rate of the 64-core parts is lower than that of the 32-core parts, it is not half the rate when the number of cores is doubled[3]. Assuming that the interconnect is similarly scaled to support the additional cores, the 64-core CPUs should be able to deliver more aggregate radio throughput but possibly spread over a larger number of reduced width channels.

[2]Framing would need to be integrated into the Tx but could potentially be merged into the Tx I/O thread

[3]This make sense if power limited. Each of these parts has the same Thermal Design Power (TDP) of 280W [107], [199]–[201]. Using the concepts in [202], parallelizing a design while lowering the clock rate and supply voltage accordingly yields power and energy savings. In the case of these CPUs where power is kept constant, the non-linear relationship of dynamic power ($P_{SW} = \frac{1}{2}\alpha C V_{DD}^2 F$ where α is the activity factor, C is the capacitance which increases with additional logic, V_{DD} is the supply voltage, and F is the clock frequency) allows the amount of logic to be doubled without needing to half the clock frequency to maintain the same dynamic power, so long as the supply voltage is adjusted accordingly.

	Threadripper 3970X	Threadripper 3990X	Epyc 75F3	Epyc 7763
CPU Clk	3.7	2.9	2.95	2.45
Cores	32	64	32	64
Expected Single Instance Performance With Perfect Clock Scaling				
Baseband Sample Freq	104.50	81.91	83.32	69.20
Rx Only - Single Laminar I/O Thread for All Instances (12 Cores/Instance + 4)				
Cyclops Instances	2	5	2	5
Max Data Rate (Payload)	557.33	**1092.07**	444.36	**922.61**
Required Cores	24	60	24	60
Rx Only - Laminar I/O Thread for Each Instance (13 Cores/Instance + 3)				
Cyclops Instances	2	4	2	4
Max Data Rate (Payload)	557.33	**873.66**	444.36	**738.09**
Required Cores	26	52	26	52
Tx & Rx - Laminar I/O Thread for Each Tx & Rx (16 Cores/Instance + 3)				
Cyclops Instances	1	3	1	3
Max Data Rate (Payload)	278.67	655.24	222.18	553.57
Required Cores	16	48	16	48

Table 9.3: Extrapolated Rough Cyclops Performance Estimate on Similar CPUs

Chapter 10

Conclusion

Broadly speaking, this project aimed to evaluate the realities of modern software radio. Given the power of modern CPUs and the restricted domain of radio signal processing, software radio should to be competitive with specialized hardware solutions. However, experience has shown that producing a high-performance software radio is challenging. By writing the Laminar compiler and optimizing the Cyclops radio, the hope was to provide insight into what it takes to produce a fast and efficient software radio, what limitations still exist, and what can be done to make it easier for radio designers to use general-purpose CPUs.

This project delivered on these goals by demonstrating the acceleration of a complex PHY layer receiver from a single core instance operating at around 2 Msps to a multi-core vectored implementation running at over 100 Msps. With the use of multi-channel techniques and a larger CPU, multiple instances of this baseband could potentially operate with an aggregate rate close to 1 Gbps in the payload portion of frames. The effective mapping of the radio design to the parallel resources available on the CPU was essential to achieve the speedups observed. The Laminar compiler provided a framework to test different techniques for exposing parallelism in such a way that it could be leveraged by a modern C compiler to produce a fast executable file. By encoding these optimizations in Laminar, optimizations that can be tedious for a designer to handwrite can be applied more consistently to a range of designs, improving designer productivity. Inter-core communications benchmarking and design analysis tools provide context and feedback to the designer to achieve better partitioning and mapping results.

At the start of this project, there was a question if a specialized compiler, such as Laminar, would be able to turn any radio design into a high-performance implementation. As this project progressed, it became clear that designer awareness of the underlying execution platform and the optimizations available in the compiler can lead to algorithmic modifications that yield significant performance improvements. This is similar to how designs must often be modified from their golden reference to fit the constraints of FPGA or ASIC design. There is a desire to expect that, because of the power of modern CPUs and the maturity of modern compilers, that software radio can sidestep careful consideration by a designer.

While the optimization passes in Laminar and the underlying C compiler certainly have a strong impact on performance, they perform best when the baseband designer expresses the design in such a way that the compiler can detect and best exploit the parallelism which is available.

Dataflow graph representations present a compelling representation which explicitly expresses parallelism and matches what hardware and DSP designers are accustomed to. Because of its close relationship with RTL descriptions, a design expressed as a dataflow graph can be either fully or partially mapped to software with a subgraph mapped to a hardware. Just like when mapping to software, obtaining the best results when mapping the subgraph onto hardware would likely require modifications by the designer to fit the constraints of the FPGA or ASIC flow.

10.1 Remaining Challenges

While the project was successful in demonstrating high-speed software radio, several challenges with platforms and radio designs were identified that can act as impediments to achieving desired results.

10.1.1 Platform

Modern CPUs are a marvel of digital design with an excellent amount of computing resources connected in such a way that many of the details can be hidden from general software developers. However, when writing code which pushes the boundary of the platform, understanding these systems becomes essential. The core-core communication benchmarks discussed in section 7.4 show some of the complexities of memory subsystems and the challenges which are faced on many-core systems. While the throughput is high and latency is relatively low when communicating within an L3 domain, communicating between L3 domains incurs a latency and performance hit. Although this is not unexpected given the complexity of constructing an interconnect across multiple L3 domains spread across multiple physical dies, it does introduce an added challenge when partitioning and mapping a DSP design. The impact of this can be reduced by increasing the bandwidth of the interconnect links or by increasing the number of cores per L3 domain. In fact, AMD's Zen 3 microarchitecture increases the number of cores per L3 domain from 4 to 8 [203] which, along with other microarchitectural changes in the Zen 3 core, would likely provide an uplift to radio performance.

In general, the complexity of the cache coherency system can present challenges for radio designers. Even given specifications for a given part, such as memory bandwidth, it can be unclear what performance is achievable and what configurations provide reasonable results. Questions arising from this uncertainty include:

- Is all memory bandwidth accessible to a single CPU core or must multiple cores be involved to saturate the memory link?

- What is the effect of multiple sets of cores communicating and generating cache coherency traffic?
- What block sizes are required to reach a specific level of performance?
- Is the use of alternate memory modes or instructions warranted? If so, when?

By performing inter-core communication and memory benchmarks, such as in section 7.4, some of these questions can be addressed, guiding the designer on how to analyze the resulting telemetry from running radio designs. However, the best solution would be increased visibility and control over the cache and interconnect system, given the stylized nature of communication used by streaming DSP. Given the latency associated with a consuming thread checking FIFO status and fetching a block to process, another potential improvement to inter-core communication would be the addition of an out-of-band streaming mechanism between cores which could operate without explicit involvement of the participating cores beyond starting the transfer. Recent developments from Intel on Data Streaming Engines targeted at "optimizing streaming data movement and transformation operations" [204] suggest that changes to the memory subsystem to better support data movement may be on the horizon.

10.1.2 Radio

As was shown throughout this document, one of the keys to a high-performance realization of radio signal processing is the exploitation of parallelism. In purely feedforward systems, multiple techniques can be employed to introduce parallelism at the potential expense of latency. However, most sophisticated radio systems are not entirely composed of feedforward execution and involve loops. Some of these loops may not be timing critical and can have delays inserted without issue. However, feedback control systems present a problem as inserting delays results in changed control system behavior. The coarse/fine method discussed in section 8.2 can be exceptionally helpful in reducing requirements on the feedback control system, but it can require extensive rework of an algorithm to employ and may increase the workload on the system. There are also some loops, such as Finite State Machines (FSMs) which are difficult to insert delay into, especially if they need to react quickly. This can lead to an Amdahl's Law [105] problem where the speedup of the radio is limited to largely sequential FSM and control decisions. Because of this effect, radio designers may find they need to spend significant effort in either simplifying control decisions, localizing control systems closer to where their decisions are used, and potentially replicating control decisions to avoid excessive data transfer.

Another challenge when accelerating radio systems comes when a single design spreads across many cores. As the number of cores/partitions increases, the fewer instructions execute per core and the more likely it is for additional intermediate results to need to be sent between cores. With fewer operations per core, load balancing becomes more challenging with moving only a few operations potentially producing large imbalances (by ratio) between cores. Additional communication between cores also puts stress on the cache subsystem and interconnect, potentially increasing the communication overhead for multiple cores. Techniques such as multi-channel operation can help reduce the impact of this effect by allowing multiple basebands instances with fewer cores per instance to operate simultaneously on a many-core system.

10.2 Lessons Learned

With experience with baseband DSP on previous iterations of the Cyclops radio along with knowledge about computer architecture and software design, there were some preconceived notions of what to expect when embarking on this project. Some of these instincts were proven correct, some were strengthened, and some were challenged.

One of the earliest questions posed by several individuals was whether the project was necessary with modern C/C++ compilers that have matured over decades of development. The belief that general purpose compilers have become so advanced that careful design by a software developer is unnecessary is not an uncommonly held belief. This may hold a kernel of truth for applications which can afford to sacrifice performance for developer productivity. However, the use of intrinsics, specialized frameworks, and languages by the HPC and GPU fields provides a hint that this is not the case for applications that push the platform near its limit. Several experiments were conducted throughout this project to test the effect different representations of a design had on the performance of executables generated by the C compiler. It became clear early on that even the order in which operations were presented to the compiler had an impact on the achieved performance, regardless of the use of aggressive compiler optimization flags. This was later reinforced with the sub-blocking effort and the inner product loop interchange experiment. While the general C/C++ compiler can provide substantial optimization to programs (such as automatic vectorization and unrolling), it performs best when presented with easily analyzable code broken down into digestible segments. This is not altogether unsurprising as many of the problems faced by compilers have no known polynomial algorithm to solve optimally. However, it underscores that, while general purpose compilers have advanced significantly over time, they still do not handle all aspects of high-speed software development.

Another interesting discovery came from considering the representation of numbers in the design. Most hardware radio designs use fixed-point arithmetic. There are several reasons for this, including the ability to use integer multipliers and adders which are generally simpler than their floating-point equivalents. Operations such as shifting, truncation, and expansion are also extremely low cost on hardware as they generally involve changing which set of wires are passed downstream. Dedicated hardware designs also allow integer, fixed-point, and floating-point datatypes to be sized according to the resolution and dynamic range requirements of the signal, potentially reducing the amount of logic required. This level of flexibility is drastically reduced when moving to a standard CPU. Unlike FPGAs or ASICs, processors are typically built around processing fixed width data in standard representations.

While there are multiple standard datatypes, the selection is generally limited to integer multiples of bytes. Implementing fixed-point operations requires the use of shifting and masking operations to re-normalize the result. Unlike in hardware where these operations are almost free, changing what wires are routed between operations, each of these operations requires an instruction to be run on CPU execution resources. While re-normalization can sometimes be deferred to after a series of integer operations if there is sufficient headroom in the type, it adds additional work to a software implementation that does not exist in hardware. By contrast, modern high-performance CPUs have been heavily optimized for floating-point arithmetic. In fact, there are more floating-point execution units capable of multiplication on the Ryzen 3970X than there are integer execution units capable of multiplication [17]. Floating-point operations such as sum, multiply, and FMA are often fully pipelined, allowing one floating-point operations to complete every cycle. Additionally, floating-point units automatically handle the required re-normalization operations. For these reasons, the fixed-point representations used in earlier versions of Cyclops were converted to use single precision floating-point, which containes more than enough resolution to represent signals in the design. The expanded dynamic range even helped the performance of some algorithms, such as the AGC. There are still some advantages to fixed-point representations, including a potentially smaller memory footprint and the potential to fit more elements into fixed-width vector registers. However, there are practical limitations in signal processing algorithms on how small the datatype can become while maintaining the resolution required. For 12 or 14-bit ADCs/DACs, at least 16-bit type is required to contain the full resolution captured by the data converter. While Cyclops used 32-bit single precision floating-point, interest in reduced precision floating-point representations, such as half-precision float which contains an effective 11 bits of precision [205], presents an interesting opportunity in future systems to potentially gain the size advantage currently only possible with integer types while maintaining some of the advantages afforded by floating-point types.

One concern, especially when trying to optimize inter-core transfers in the reduced handshake FIFO discussed in section 7.3, was the suitability of a general purpose Linux OS when running a latency and jitter sensitive real-time application. Surprisingly, with the right selection of kernel boot options, isolation from cores from the scheduler, and offloading of functions such as interrupt handlers, the amount of performance jitter was reduced to the point where the reduced handshake FIFO was able to operate reliably in closed loop mode. With interrupts disabled on cores participating in an open-loop reduced handshake FIFO (via a custom kernel module), the system was able to run on the order of minutes without failure. These results suggest that high performance software radio can be run successfully on a standard operating system without excessive modification, provided that the user has significant control over the target system.

Initially, much of the focus of this project was to obtain performance improvements by splitting basebands across the many cores of a modern CPU. While it was always known that the exploitation of SIMD/vector units would be important to obtain the desired performance, there was a hope that the general-purpose compiler would be able to handle many of the vectorization tasks. It became clear as the project progressed, however, that specialized representations of SIMD/vector parallelism were essential for the compiler's optimization passes to provide the expected results. The realization resulted in a reevaluation of how vectorization was considered and led to the explicit representation of vector constructs, vector-focused operator implementations, and sub-blocking. It also led to inspecting Forward Error Correction (FEC), which can be notoriously difficult to parallelize. Using the Viterbi algorithm as a litmus test showed that, while challenging, tricks are available to reveal some degree of parallelism. Techniques such as interleaving can be employed when further acceleration of a single FEC instance becomes infeasible.

Finally, it should be noted how important DSP design changes can be to the performance of the radio. When Cyclops was initially converted from targeting FPGAs to targeting processors, pipeline stages that had been inserted to increase the achievable clock rate on the FPGA were removed as they appeared to be superfluous for a software implementation. This was fantastic, from a DSP design standpoint, for modules like feedback control systems which could be made much more aggressive. However, many of these pipeline stages had to be reinserted when it became clear that the feedback loops were too large to fit in single cores. The added delay was needed to provide the initial state of FIFOs within the loop. As it became clear that blocking transactions was essential to obtain high speed transfers, the amount of state required increased. Without modifications to the DSP including the use of coarse/fine control systems, the performance of the resulting system would have been severely limited. Pipelining was also reinserted into some long strings of combinational logic to provide added ILP in those sections to better load the multiple execution units in the CPU. As additional optimizations such as sub-blocking were introduced, tweaks to the design to better match the passes and abstractions provided by Laminar resulted in large performance gains.

10.3 Future Work

While this project was successful in demonstrating high-speed software radio there are several opportunities for continued research and development.

It would greatly improve the user's productivity if Laminar included additional automation, especially automated partitioning and delay re-timing. Automated partitioning has been an interest of the project since the beginning with several different possible mechanisms proposed including the use of an approximation algorithm for the capacitated k-cut problem [206]. This proposed heuristic would use the capacity constraint as a mechanism to enforce load balancing with the workload estimates of operators used to set the size of nodes[1] with either communication or a slack measure used as the arc parameter to optimize the cut for[2]. The partitioning could then be refined by iteratively moving nodes between partitions to balance load more equally. Another option would be to use an existing graph

[1] If nodes are required to have unit size, larger operators are created with strings of nodes with high costs between them.

[2] If cut maximisation and non-negative arc weights are required, then the cost can be inverted.

partitioning tool, such as METIS [207] to perform the partitioning on an exported graph from Laminar. Note that the load-balanced graph partitioning problem is NP-hard [207] meaning that, for any large graph, heuristics or approximations will be the only feasible solution. It is also important to note that, on modern CPUs, the costs associated with transfers depend not only on the location of the two communication threads but also the activity around them. This results in the costs used in the partitioning algorithm itself depending on the partitions and mappings chosen. Related to automated partitioning is the requirement that delays be available at partition boundaries participating in loops so that they can be shifted into FIFOs as the initial state. This initial state is essential to avoid deadlock between the threads in the loop and to provide enough input to keep them working in parallel. Currently, this is performed manually by the designer when annotating partition boundaries. However, with automated partitioning, this would need to occur in an automated fashion. Unlike traditional VLSI style retiming, the primary goal of this pass would be to move delays to the partition boundaries participating in loops. For segments of the design not participating in these communication loops, standard retiming to create ILP via balanced software pipelining would be a good secondary objective. In a similar vein, providing additional mechanisms to automate parameter explorations and sweeps with the Laminar compiler would help facilitate finding configurations options which would maximize performance for a given design (autotuning). In cases where perfect load balancing cannot be achieved with the partitioning tool, the use of Dynamic Voltage and Frequency Scaling (DVFS) to slow down threads with slack and potentially speed up the clock of bottleneck threads would be a compelling method to close the performance gap.

As was discovered in subsection 5.3.2, the scheduling heuristic used in the emitter is still relevant to performance despite the loss of precise scheduling control in out-of-order CPUs and the flexibility of the compiler to rearrange instructions. Investigation of more sophisticated scheduling heuristics, including ones which view superscalar CPUs like a VLIW CPUs from a scheduling perspective [208], [209], should be further investigated with the Laminar emitter. Likewise, investigating what representations of a designs fit best with a C compiler's optimization passes will likely be an ongoing process as features are added and heuristics are changed overtime. As mentioned before, many of the problems C compilers are trying to solve are NP-hard, meaning that they typically need to resort to heuristic methods to accomplish these goals. As new CPUs are released, compilers will adapt with them. The author expects that the general principle of presenting compilers with tight, easily analyzable code segments will be a reliable method to extract good performance out of compilers as they continue to develop for the foreseeable future. However, getting the best performance may involve some additional tweaks to the way code is emitted.

While there is an impressive set of features supported and optimization passes implemented within Laminar, there continues to be opportunities for improvement. One possible direction is supporting variable base sub-blocking sizes by allowing multiple partitions to execute in a round robin fashion on a single core (to handle the width adaptation). Support for upsample clock domains and general rational resampling would be another useful feature to add, along with a general cleanup of the codebase.

Supporting GPUs would be another potential area for Laminar to expand. However, the latency associated with GPU operations with the CPU in the loop could cause challenges. As a result, GPU support would likely be best suited for a system with integrated GPUs or future GPUs which are capable of spawning threads on their own and passing data between threads without the explicit involvement of the CPU.

The recently released Xilinx Versal ACAP architecture also presents a very compelling future target platform for Laminar. Xilinx Versal introduces a 2D array of VLIW vector processors called the *AI Engine* [210]. The AI engine cores support both fixed and floating-point operations, support complex math, and can operate in a MIMD fashion [210]. In contrast to modern multicore general-purpose CPUs, communication between cores can occur without the use of a cache coherency protocol via direct access to the memory of adjacent cores [210]. Communication to distant cores can occur via AXI buses and DMAs [210]. The implementation of the AI Engine addresses many of the challenges experienced when using general-purpose CPUs, particularly with the cache subsystem, and presents an architecture which fits well with the compute model taken by Laminar.

Bibliography

[1] Ericsson, "Ericsson Mobility Report November 2021," en, p. 40, Nov. 2021. [Online]. Available: https://www.ericsson.com/4ad7e9/assets/local/reports-papers/mobility-report/documents/2021/ericsson-mobility-report-november-2021.pdf.

[2] Gartner, *Gartner Says Worldwide Smartphone Sales Declined 5% in Fourth Quarter of 2020*, en, Press Release, Feb. 2021. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2021-02-22-4q20-smartphone-market-share-release (visited on 01/11/2022).

[3] ——, *Gartner Says Worldwide Smartphone Sales to Grow 11% in 2021*, en, Press Release, Feb. 2021. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2021-02-03-gartner-says-worldwide-smartphone-sales-to-grow-11-percent-in-2021 (visited on 01/11/2022).

[4] CTIA, "2021 Annual Survey - Highlights," en, Tech. Rep., 2021, p. 11. [Online]. Available: https://www.ctia.org/news/2021-annual-survey-highlights (visited on 01/11/2022).

[5] Advanced Micro Devices, *EPYC 7001 Series Processors*, en. [Online]. Available: https://www.amd.com/en/products/epyc-7000-series (visited on 04/27/2022).

[6] ——, *2nd Gen AMD EPYC Processors Set New Standard for the Modern Datacenter with Record-Breaking Performance and Significant TCO Savings*, en, Aug. 2019. [Online]. Available: https://www.amd.com/en/press-releases/2019-08-07-2nd-gen-amd-epyc-processors-set-new-standard-for-the-modern-datacenter (visited on 04/18/2022).

[7] Intel, *Intel AVX-512 Instructions*, en, Jun. 2017. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html (visited on 04/27/2022).

[8] Advanced Micro Devices, *AMD Ryzen Threadripper 2990WX Processor*, en. [Online]. Available: https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-2990wx (visited on 02/16/2022).

[9] ——, “Software Optimization Guide for AMD Family 17h Processors,” en, Tech. Rep. 55723, Jun. 2017, p. 45. [Online]. Available: `https://developer.amd.com/wordpress/media/2013/12/55723_3_00.ZIP`.

[10] C. Yarp, “A 60 GHz Development & Test Platform for Wireless Systems Research,” Issue: UCB/EECS-2018-17, M.S. thesis, EECS Department, University of California, Berkeley, May 2018. [Online]. Available: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-17.html`.

[11] E. Lee and D. Messerschmitt, “Synchronous data flow,” *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987. DOI: `10.1109/PROC.1987.13876`.

[12] ——, “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing,” *IEEE Transactions on Computers*, vol. C-36, no. 1, pp. 24–35, Jan. 1987. DOI: `10.1109/TC.1987.5009446`.

[13] E. Lee and S. Ha, “Scheduling strategies for multiprocessor real-time DSP,” in *1989 IEEE Global Telecommunications Conference and Exhibition 'Communications Technology for the 1990s and Beyond'*, 1989, 1279–1283 vol.2. DOI: `10.1109/GLOCOM.1989.64160`.

[14] S. Sriram, “Minimizing Communication and Synchronization Overhead in Multiprocessors for Digital Signal Processing,” Issue: UCB/ERL M95/90, PhD Thesis, EECS Department, University of California, Berkeley, Nov. 1995. [Online]. Available: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/1995/2897.html`.

[15] P. Hoang and J. Rabaey, “A compiler for multiprocessor DSP implementation,” in *[Proceedings] ICASSP-92: 1992 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, Mar. 1992, 581–584 vol.5. DOI: `10.1109/ICASSP.1992.226553`.

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Waltham, MA: Morgan Kaufmann, 2011, ISBN: 978-0-12-383872-8.

[17] Advanced Micro Devices, “Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors,” en, Tech. Rep. 56305, Mar. 2020, p. 45. [Online]. Available: `https://developer.amd.com/resources/developer-guides-manuals/%20https://developer.amd.com/wp-content/resources/56305.zip`.

[18] M. Geilen and T. Basten, “Requirements on the Execution of Kahn Process Networks,” in *Programming Languages and Systems*, P. Degano, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 319–334, ISBN: 978-3-540-36575-4.

[19] G. Kahn, “The Semantics of a Simple Language for Parallel Programming,” in *Proceedings of IFIP Congress 74*, North-Holland, 1974, pp. 471–475.

[20] C. A. R. Hoare, "Communicating Sequential Processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978, Place: New York, NY, USA Publisher: Association for Computing Machinery, ISSN: 0001-0782. DOI: `10.1145/359576.359585`. [Online]. Available: `https://doi.org/10.1145/359576.359585`.

[21] Google, *Effective Go - The Go Programming Language*, en. [Online]. Available: `https://go.dev/doc/effective_go` (visited on 04/28/2022).

[22] GNU Radio, *GNU Radio - The Free & Open Source Radio Ecosystem.* [Online]. Available: `https://www.gnuradio.org/` (visited on 04/28/2022).

[23] ——, *Hardware.* [Online]. Available: `https://wiki.gnuradio.org/index.php?title=Hardware` (visited on 04/28/2022).

[24] Ettus Research, *USRP X Series*, en. [Online]. Available: `https://www.ettus.com/products/usrp-x-series/` (visited on 04/28/2022).

[25] ——, *RFNoC.* [Online]. Available: `https://kb.ettus.com/RFNoC_(UHD_3.0)` (visited on 04/28/2022).

[26] DPDK Project, *DPDK.* [Online]. Available: `https://core.dpdk.org/` (visited on 04/28/2022).

[27] K. Venkatesan and B. Perlman, "Accelerating Telco NFV Deployments with DPDK and SmartNICs," Dec. 2018. [Online]. Available: `https://www.dpdk.org/wp-content/uploads/sites/35/2018/12/Kalimani-and-Barak-Accelerating-NFV-with-DPDK-and-SmartNICs.pdf`.

[28] DPDK Project, *16. Wireless Baseband Device Library — Data Plane Development Kit 22.03.0 documentation.* [Online]. Available: `http://doc.dpdk.org/guides/prog_guide/bbdev.html?highlight=baseband` (visited on 04/28/2022).

[29] Telecom Infra Project, *OpenRAN*, en-US. [Online]. Available: `https://telecominfraproject.com/openran/` (visited on 04/29/2022).

[30] O-RAN Alliance, *Operator Defined Open and Intelligent Radio Access Networks*, en-US. [Online]. Available: `https://www.o-ran.org` (visited on 04/29/2022).

[31] srsRAN, *srsRAN - Your own mobile network.* [Online]. Available: `https://www.srslte.com/` (visited on 04/28/2022).

[32] O. S. Alliance, *5G RAN – OpenAirInterface*, en-US. [Online]. Available: `https://openairinterface.org/oai-5g-ran-project/` (visited on 04/29/2022).

[33] Intel, *Intel FlexRAN Reference Architecture for Wireless Access*, en. [Online]. Available: `https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/tools/flexran.html` (visited on 05/08/2022).

[34] NVIDIA, *NVIDIA Aerial SDK*, en-US. [Online]. Available: `https://developer.nvidia.com/aerial-sdk` (visited on 04/29/2022).

[35] 3GPP, *Open RAN*, Jan. 2021. [Online]. Available: `https://www.3gpp.org/news-events/2150-open_ran` (visited on 04/29/2022).

[36] 3rd Generation Partnership Project, "Study on new radio access technology: Radio access architecture and interfaces (Release 14)," Sophia Antipolis, Tech. Rep. 3GPP TR 38.801, Mar. 2017, p. 91. [Online]. Available: `https://www.3gpp.org/ftp//Specs/archive/38_series/38.801/38801-e00.zip` (visited on 04/29/2022).

[37] Synopsys, *VCS Functional Verification Solution*, en. [Online]. Available: `https://www.synopsys.com/verification/simulation/vcs.html` (visited on 04/28/2022).

[38] Veripool, *Introduction to Verilator*. [Online]. Available: `https://www.veripool.org/projects/verilator/wiki/Intro`.

[39] W. Snyder, "Verilator 4.0: Open Simulation Goes Multithreaded," 2018. [Online]. Available: `https://www.veripool.org/papers/Verilator_v4_Multithreaded_OrConf2018.pdf`.

[40] MathWorks, *MATLAB*, en. [Online]. Available: `https://www.mathworks.com/products/matlab.html` (visited on 04/30/2022).

[41] ——, *Simulink - Simulation and Model-Based Design*, en. [Online]. Available: `https://www.mathworks.com/products/simulink.html` (visited on 03/28/2022).

[42] ——, *DSP System Toolbox*, en. [Online]. Available: `https://www.mathworks.com/products/dsp-system.html` (visited on 03/28/2022).

[43] ——, *Communications Toolbox*, en. [Online]. Available: `https://www.mathworks.com/products/communications.html` (visited on 03/28/2022).

[44] ——, *MATLAB Coder*, en. [Online]. Available: `https://www.mathworks.com/products/matlab-coder.html` (visited on 04/30/2022).

[45] ——, *Simulink Coder*, en. [Online]. Available: `https://www.mathworks.com/products/simulink-coder.html` (visited on 03/28/2022).

[46] ——, *How to Combine Modeling Styles with Schedulable Components in Simulink Video*, en. [Online]. Available: `https://www.mathworks.com/videos/run-time-software-modeling-part-3-how-to-combine-modeling-styles-with-schedulable-components-in-simulink-1552370473474.html` (visited on 04/28/2022).

[47] ——, *Dataflow Domain - MATLAB & Simulink*. [Online]. Available: `https://www.mathworks.com/help/dsp/ug/dataflow-domains.html` (visited on 03/28/2022).

[48] ——, *Multicore Processor Targets - MATLAB & Simulink*. [Online]. Available: `https://www.mathworks.com/help/simulink/multicore-processor-targets.html?s_tid=CRUX_lftnav` (visited on 03/28/2022).

[49] R. C. Whaley and J. Dongarra, "Automatically Tuned Linear Algebra Software," University of Tennessee, Tech. Rep. UT-CS-97-366, Dec. 1997. [Online]. Available: `http://www.netlib.org/lapack/lawns/lawn131.ps`.

[50] FFTW, *FFTW Home Page*. [Online]. Available: `https://www.fftw.org/#documentation` (visited on 05/01/2022).

[51] F. Franchetti, T. M. Low, D. T. Popovici, *et al.*, "SPIRAL: Extreme Performance Portability," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018. DOI: `10.1109/JPROC.2018.2873289`.

[52] Intel, *Accelerate Fast Math with Intel oneAPI Math Kernel Library*, en. [Online]. Available: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html` (visited on 05/01/2022).

[53] Advanced Micro Devices, *AMD Optimizing CPU Libraries (AOCL)*, en-US. [Online]. Available: `https://developer.amd.com/amd-aocl/` (visited on 05/01/2022).

[54] ——, *AMD BLIS*, en-US. [Online]. Available: `https://developer.amd.com/amd-aocl/blas-library/` (visited on 05/01/2022).

[55] ——, *AMD FFTW*, en-US. [Online]. Available: `https://developer.amd.com/amd-aocl/fftw/` (visited on 05/01/2022).

[56] NVIDIA, *CUDA-X GPU-Accelerated Libraries*, en-US. [Online]. Available: `https://developer.nvidia.com/gpu-accelerated-libraries` (visited on 05/01/2022).

[57] ——, *cuBLAS*, en-US. [Online]. Available: `https://developer.nvidia.com/cublas` (visited on 05/01/2022).

[58] ——, *cuFFT*, en-US. [Online]. Available: `https://developer.nvidia.com/cufft` (visited on 05/01/2022).

[59] J. Ragan-Kelley, "Decoupling Algorithms from the Organization of Computation for High Performance Image Processing: The design and implementation of the Halide language and compiler," Ph.D. MIT, May 2014.

[60] TensorFlow Developers, *TensorFlow*, Apr. 2022. DOI: `10.5281/zenodo.6476456`. [Online]. Available: `https://doi.org/10.5281/zenodo.6476456`.

[61] Y. Jia, E. Shelhamer, J. Donahue, *et al.*, "Caffe: Convolutional Architecture for Fast Feature Embedding," *arXiv preprint*, 2014. DOI: `10.48550/ARXIV.1408.5093`. [Online]. Available: `https://arxiv.org/pdf/1408.5093.pdf`.

[62] A. Paszke, S. Gross, F. Massa, *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d. Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[63] The Clang Team, *Clang C Language Family Frontend for LLVM*. [Online]. Available: `https://clang.llvm.org/` (visited on 05/02/2022).

[64] Free Software Foundation, Inc., *GCC, the GNU Compiler Collection - GNU Project*. [Online]. Available: `https://gcc.gnu.org/` (visited on 05/02/2022).

[65] Advanced Micro Devices, *AMD Optimizing C/C++ and Fortran Compilers (AOCC)*, en-US. [Online]. Available: `https://developer.amd.com/amd-aocc/` (visited on 05/02/2022).

[66] LLVM Project, *LLVM's Analysis and Transform Passes — LLVM 15.0.0git documentation*. [Online]. Available: `https://llvm.org/docs/Passes.html` (visited on 05/02/2022).

[67] Free Software Foundation, Inc., *Optimize Options (Using the GNU Compiler Collection (GCC))*. [Online]. Available: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html` (visited on 05/02/2022).

[68] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," en, Knoxville, Tennessee, Standard, Jun. 2021, p. 1139. [Online]. Available: `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`.

[69] OpenMP Architecture Review Board, "OpenMP Application Programming Interface," Standard Version 5.2, Nov. 2021, p. 669. [Online]. Available: `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf`.

[70] A. Cassagne, "Optimization and parallelization methods for software-defined radio," fr, Ph.D. École Doctorale Mathématiques et informatique, Bordeaux, Fr, Dec. 2020.

[71] Y. Jia and E. Shelhamer, *Caffe — Blobs, Layers, and Nets*. [Online]. Available: `https://caffe.berkeleyvision.org/tutorial/net_layer_blob.html` (visited on 05/01/2022).

[72] TensorFlow Developers, *Introduction to graphs and tf.function — TensorFlow Core*, en. [Online]. Available: `https://www.tensorflow.org/guide/intro_to_graphs` (visited on 05/01/2022).

[73] C. Yarp, *Cyclopsbb-pub*. DOI: `10.5281/zenodo.6525784`. [Online]. Available: `https://github.com/ucb-cyarp/cyclopsbb-pub` (visited on 04/03/2022).

[74] J. G. Proakis, *Digital Communications*, ser. McGraw-Hill series in electrical and computer engineering. Boston : McGraw-Hill, c2001., 2001, ISBN: 0-07-232111-3.

[75] J. G. Proakis, M. Salehi, and G. Bauch, *Contemporary Communication Systems using MATLAB*. Stamford, CT : Cengage Learning, c2013., 2013, ISBN: 978-0-495-08251-4.

[76] R. G. Gallager, *Principles of digital communication*. Cambridge ; New York: Cambridge University Press, 2008, OCLC: ocn166382261, ISBN: 978-0-521-87907-1.

[77] J. G. Proakis, "Chapter 5: Optimum Receivers for Additive Whate Gaussian Noise," in *Digital communications*, ser. McGraw-Hill series in electrical and computer engineering, 4th ed, Boston: McGraw-Hill, 2000, pp. 231–332, ISBN: 978-0-07-232111-1.

[78] S. K. Mitra, *Digital signal processing: a computer-based approach*, 4th ed. New York, NY: McGraw-Hill, 2011, ISBN: 978-0-07-338049-0.

[79] MathWorks, *QPSK Modulator Baseband.* [Online]. Available: `https://www.mathworks.com/help/comm/ref/qpskmodulatorbaseband.html` (visited on 01/18/2022).

[80] ——, *Rectangular QAM Modulator Baseband.* [Online]. Available: `https://www.mathworks.com/help/comm/ref/rectangularqammodulatorbaseband.html` (visited on 01/18/2022).

[81] ——, *Bit error rate (BER) for uncoded AWGN channels - MATLAB berawgn.* [Online]. Available: `https://www.mathworks.com/help/comm/ref/berawgn.html`.

[82] ——, *AWGN Channel.* [Online]. Available: `https://www.mathworks.com/help/comm/ug/awgn-channel.html` (visited on 05/02/2022).

[83] J. G. Proakis, "Chapter 14: Digital Communications Through Fading Multipath Channels," in *Digital communications*, ser. McGraw-Hill series in electrical and computer engineering, 4th ed, Boston: McGraw-Hill, 2000, pp. 800–895, ISBN: 978-0-07-232111-1.

[84] J. P. A. Pérez, S. C. Pueyo, and B. C. López, "AGC Fundamentals," in *Automatic Gain Control: Techniques and Architectures for RF Receivers*, New York, NY: Springer New York, 2011, pp. 13–28, ISBN: 978-1-4614-0167-4. DOI: `10.1007/978-1-4614-0167-4_2`. [Online]. Available: `http://dx.doi.org/10.1007/978-1-4614-0167-4_2`.

[85] IEEE, "ISO/IEC/IEEE 8802-11 Amendment 3: Enhancements for very high throughput in the 60 GHz Band (adoption of IEEE Std 802.11ad-2012)," vol. 25021, 2012, ISSN: 0738156655. DOI: `10.1109/IEEESTD.2015.7106438`.

[86] Agilent Technologies, "Wireless LAN at 60 GHz - IEEE 802.11ad Explained: Application Note," Tech. Rep., 2013, pp. 1–28. [Online]. Available: `http://cp.literature.agilent.com/litweb/pdf/5990-9697EN.pdf`.

[87] H. Meyr, M. Moeneclaey, and S. A. Fechtel, *Digital Communication Receivers: Synchronization, Channel Estimation, and Signal Processing.* John Wiley & Sons, 1997, ISBN: 0-471-50275-8. [Online]. Available: `http://dx.doi.org/10.1002/0471200573`.

[88] W. C. Lindsey and M. K. Simon, *Telecommunication Systems Engineering*, ser. Prentice-Hall Information and System Sciences Series. Englewood Cliffs, N.J: Prentice-Hall, 1973, ISBN: 0-13-902429-8.

[89] MathWorks, *NCO.* [Online]. Available: `https://www.mathworks.com/help/dsp/ref/nco.html` (visited on 02/27/2022).

[90] M. H. Hayes, *Statistical Digital Signal Processing and Modeling.* New York: Wiley, 1996, ISBN: 0-471-59431-8.

[91] N. Ahmed, "Chapter 12: Adaptive Filtering," in *Handbook of digital signal processing: engineering applications*, D. F. Elliott, Ed., San Diego: Academic Press, 1987, pp. 857–897, ISBN: 978-0-12-237075-5.

[92] M. Ekman, *Learning deep learning: theory and practice of neural networks, computer vision, nlp, and transformers using tensorflow*, First edition. Boston: Addison-Wesley, 2021, ISBN: 978-0-13-747035-8.

[93] G. Clark, S. Mitra, and S. Parker, "Block implementation of adaptive digital filters," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 3, pp. 744–752, 1981. DOI: `10.1109/TASSP.1981.1163603`.

[94] L. Hsieh and S. Wood, "Performance analysis of time domain block LMS algorithm," in *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 3, 1993, 535–538 vol.3. DOI: `10.1109/ICASSP.1993.319553`.

[95] N. H. E. Weste and D. M. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th. Boston: Addison Wesley, 2011, ISBN: 978-0-321-54774-3.

[96] G. E. Moore, "Cramming More Components Onto Integrated Circuits," en, *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, Jan. 1998, ISSN: 0018-9219, 1558-2256. DOI: `10.1109/JPROC.1998.658762`. [Online]. Available: `http://ieeexplore.ieee.org/document/658762/` (visited on 03/16/2022).

[97] J. K. Liu, "FinFET History, Fundamentals and Future," en, *2012 Symposium on VLSI Technology Short Course*, p. 55, Jun. 2012. [Online]. Available: `https://people.eecs.berkeley.edu/~tking/presentations/KingLiu_2012VLSI-Tshortcourse`.

[98] S. Naffziger, N. Beck, T. Burd, *et al.*, "Pioneering Chiplet Technology and Design for the AMD EPYC and Ryzen Processor Families: Industrial Product," en, in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, Valencia, Spain: IEEE, Jun. 2021, pp. 57–70, ISBN: 978-1-66543-333-4. DOI: `10.1109/ISCA52012.2021.00014`. [Online]. Available: `https://ieeexplore.ieee.org/document/9499852/` (visited on 03/07/2022).

[99] Intel, *Intel Xeon Platinum Processors*, en. [Online]. Available: `https://www.intel.com/content/www/us/en/products/details/processors/xeon/scalable/platinum.html` (visited on 02/16/2022).

[100] ——, *Intel Core X-Series Processor Family*, en. [Online]. Available: `https://www.intel.com/content/www/us/en/products/details/processors/core/x.html` (visited on 02/16/2022).

[101] Advanced Micro Devices, *AMD Introduces World's Fastest High-End Desktop Processors With 3rd Gen Ryzen Threadripper Family: Delivering Unmatched Performance With No Compromises*, en, Nov. 2019. [Online]. Available: `https://www.amd.com/en/press-releases/2019-11-07-amd-introduces-world-s-fastest-high-end-desktop-processors-3rd-gen-ryzen` (visited on 04/18/2022).

[102] Nvidia, *Inside Volta: The World's Most Advanced Data Center GPU*, en-US, May 2017. [Online]. Available: `https://developer.nvidia.com/blog/inside-volta/` (visited on 03/04/2022).

[103] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4," en, Tech. Rep. 325462-067US, May 2018, p. 4844. [Online]. Available: `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

[104] Advanced Micro Devices, "AMD64 Architecture Programmer's Manual, Volumes 1-5," en, Tech. Rep. 40332, Mar. 2021, p. 3269. [Online]. Available: `https://www.amd.com/system/files/TechDocs/40332.pdf`.

[105] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th ed, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Waltham, MA: Morgan Kaufmann, 2012, ISBN: 978-0-12-374750-1.

[106] D. Suggs, D. Bouvier, M. Clark, K. Lepak, and M. Subramony, "AMD "ZEN 2"," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, 2019, pp. 1–24. DOI: `10.1109/HOTCHIPS.2019.8875673`.

[107] Advanced Micro Devices, *3rd Gen AMD Ryzen Threadripper 3970X Desktop Processor*. [Online]. Available: `https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3970x` (visited on 03/07/2022).

[108] J. Demmel, *UC Berkeley CS267 Home Page: Spring 2014*, 2014. [Online]. Available: `https://people.eecs.berkeley.edu/~demmel/cs267_Spr14/` (visited on 03/09/2022).

[109] T. Burd, N. Beck, S. White, *et al.*, ""Zeppelin": An SoC for Multichip Architectures," en, *IEEE Journal of Solid-State Circuits*, vol. 54, no. 1, pp. 133–143, Jan. 2019, ISSN: 0018-9200, 1558-173X. DOI: `10.1109/JSSC.2018.2873584`. [Online]. Available: `https://ieeexplore.ieee.org/document/8510845/` (visited on 03/04/2022).

[110] J. Bornholt, *Memory Consistency Models: A Tutorial*, en, Feb. 2016. [Online]. Available: `https://www.cs.utexas.edu/~bornholt/post/memory-models.html` (visited on 03/09/2022).

[111] R. Hallock, *Previewing Dynamic Local Mode for the AMD Ryzen Threadripper WX Series Processors*, en, Section: Gaming, Oct. 2018. [Online]. Available: `https://community.amd.com/t5/gaming/previewing-dynamic-local-mode-for-the-amd-ryzen-threadripper-wx/ba-p/416216` (visited on 03/10/2022).

[112] S. Naffziger, K. Lepak, M. Paraschou, and M. Subramony, "2.2 AMD Chiplet Architecture for High-Performance Server and Desktop Products," en, in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, San Francisco, CA, USA: IEEE, Feb. 2020, pp. 44–45, ISBN: 978-1-72813-205-1. DOI: `10.1109/ISSCC19947.2020.9063103`. [Online]. Available: `https://ieeexplore.ieee.org/document/9063103/` (visited on 03/08/2022).

[113] P. Alcorn, *AMD Threadripper 3970X and 3960X Review: High-End Domination*, en, Dec. 2019. [Online]. Available: `https://www.tomshardware.com/reviews/amd-threadripper-3970x-review` (visited on 03/08/2022).

[114] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260–269, Apr. 1967, ISSN: 0018-9448, 1557-9654. DOI: `10.1109/TIT.1967.1054010`. [Online]. Available: `http://ieeexplore.ieee.org/document/1054010/` (visited on 02/09/2021).

[115] ——, "Convolutional Codes and Their Performance in Communication Systems," *IEEE Transactions on Communication Technology*, vol. 19, no. 5, pp. 751–772, Oct. 1971, ISSN: 0018-9332. DOI: `10.1109/TCOM.1971.1090700`. [Online]. Available: `http://ieeexplore.ieee.org/document/1090700/` (visited on 02/09/2021).

[116] J. G. Proakis, "Chapter 8: Block and Convolutional Channel Codes," in *Digital communications*, ser. McGraw-Hill series in electrical and computer engineering, 4th ed, Boston: McGraw-Hill, 2000, pp. 416–547, ISBN: 978-0-07-232111-1.

[117] G. Forney, "The Viterbi Algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973, ISSN: 0018-9219. DOI: `10.1109/PROC.1973.9030`. [Online]. Available: `http://ieeexplore.ieee.org/document/1450960/` (visited on 02/10/2021).

[118] C. Rader, "Memory Management in a Viterbi Decoder," en, *IEEE Transactions on Communications*, vol. 29, no. 9, pp. 1399–1401, Sep. 1981, ISSN: 0096-2244. DOI: `10.1109/TCOM.1981.1095146`. [Online]. Available: `http://ieeexplore.ieee.org/document/1095146/` (visited on 03/19/2021).

[119] F. de Mesmay, S. Chellappa, F. Franchetti, and M. Püschel, "Computer Generation of Efficient Software Viterbi Decoders," in *High Performance Embedded Architectures and Compilers*, D. Hutchison, T. Kanade, J. Kittler, *et al.*, Eds., vol. 5952, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 353–368, ISBN: 978-3-642-11514-1. [Online]. Available: `http://link.springer.com/10.1007/978-3-642-11515-8_26` (visited on 02/27/2021).

[120] H. Hendrix, "Viterbi Decoding Techniques for the TMS320C55x DSP Generation," en, Texas Instruments, Application Report SPRA776A, Apr. 2009, p. 27. [Online]. Available: `https://www.ti.com/lit/an/spra776a/spra776a.pdf`.

[121] C. Yarp, *ConvolutionalEncDec*. DOI: `10.5281/zenodo.6526429`. [Online]. Available: `https://github.com/ucb-cyarp/ConvolutionalEncDec` (visited on 04/03/2022).

[122] Open Digital Radio, *Opendigitalradio/ka9q-fec*. [Online]. Available: `https://github.com/Opendigitalradio/ka9q-fec` (visited on 03/15/2022).

[123] Quiet Modem Project, *Quiet/libfec*. [Online]. Available: `https://github.com/quiet/libfec` (visited on 03/15/2022).

[124] P. Karn, *Forward Error Correcting Codes*, Aug. 2007. [Online]. Available: `http://www.ka9q.net/code/fec/` (visited on 03/15/2022).

[125] F. de Mesmay, *Spiral Project: Viterbi Decoder Software Generator.* [Online]. Available: `http://www.spiral.net/software/viterbi.html` (visited on 03/15/2022).

[126] G. Cherubini, E. Eleftheriou, and S. Olcer, "Filtered multitone modulation for very high-speed digital subscriber lines," en, *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 5, pp. 1016–1028, Jun. 2002, ISSN: 0733-8716. DOI: `10.1109/JSAC.2002.1007382`. [Online]. Available: `http://ieeexplore.ieee.org/document/1007382/` (visited on 03/17/2022).

[127] F. J. Harris, *Multirate Signal Processing for Communication Systems, Second Edition.* Gistrup: River Publishers, 2021, ISBN: 978-87-7022-210-5.

[128] V. Jacobson, "Congestion Avoidance and Control," in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM '88, event-place: Stanford, California, USA, New York, NY, USA: Association for Computing Machinery, 1988, pp. 314–329, ISBN: 0-89791-279-9. DOI: `10.1145/52324.52356`. [Online]. Available: `https://doi.org/10.1145/52324.52356`.

[129] N. Dukkipati and N. McKeown, "Why flow-completion time is the right metric for congestion control," en, *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 1, pp. 59–62, Jan. 2006, ISSN: 0146-4833. DOI: `10.1145/1111322.1111336`. [Online]. Available: `https://dl.acm.org/doi/10.1145/1111322.1111336` (visited on 05/03/2022).

[130] C. Yarp, *Laminar: An Optimizing DSP Compiler for Data Flow Graphs.* DOI: `10.5281/zenodo.6525757`. [Online]. Available: `https://github.com/ucb-cyarp/vitis` (visited on 04/03/2022).

[131] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall, "GraphML Progress Report Structural Layer Proposal," in *Graph Drawing*, P. Mutzel, M. Jünger, and S. Leipert, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 501–512, ISBN: 978-3-540-45848-7. DOI: `10.1007/3-540-45848-4_59`. [Online]. Available: `https://doi.org/10.1007/3-540-45848-4_59`.

[132] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX.," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, Aug. 2008, pp. 11–15.

[133] yWorks, *yEd Graph Editor*, en. [Online]. Available: `https://www.yworks.com/products/yed` (visited on 03/29/2022).

[134] Apache Software Foundation, *Xerces-C++ XML Parser.* [Online]. Available: `https://xerces.apache.org/xerces-c/` (visited on 03/29/2022).

[135] "Register-Transfer Level Design," en, in *VHDL for Logic Synthesis*, Chichester, UK: John Wiley & Sons, Ltd, Apr. 2011, pp. 7–17, ISBN: 978-1-119-99585-2. [Online]. Available: `https://onlinelibrary.wiley.com/doi/10.1002/9781119995852.ch2` (visited on 05/04/2022).

[136] Altera, *Design Should Not Contain Combinational Loops (Design Assistant Rule)*, 2013. [Online]. Available: `https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/verify/da/comp_file_rules_loop.htm` (visited on 05/04/2022).

[137] MathWorks, *HDL Coder*, en. [Online]. Available: `https://www.mathworks.com/products/hdl-coder.html` (visited on 03/28/2022).

[138] C. Wolf, *Yosys Open SYnthesis Suite*. [Online]. Available: `https://yosyshq.net/yosys/`.

[139] R. Love, *Linux System Programming*, en, Second edition. Sebastopol: O'Reilly, 2013, ISBN: 978-1-4493-3953-1.

[140] D. P. Bovet and M. Cesati, "3.1. Processes, Lightweight Processes, and Threads," en, in *Understanding the Linux Kernel*, 3rd Ed., 2005, ISBN: 978-0-596-00565-8. [Online]. Available: `https://learning.oreilly.com/library/view/understanding-the-linux/0596005652/ch03s01.html` (visited on 03/30/2022).

[141] G. Kroah-Hartman, "Isolcpus," en, in *Linux Kernel in a Nutshell*, O'Reilly, Dec. 2006, ISBN: 978-0-596-10079-7. [Online]. Available: `https://learning.oreilly.com/library/view/linux-kernel-in/0596100795/re46.html` (visited on 03/30/2022).

[142] Red Hat, *3.13. Isolating CPUs Using tuned-profiles-realtime Red Hat Enterprise Linux for Real Time 7*, en. [Online]. Available: `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/isolating_cpus_using_tuned-profiles-realtime` (visited on 03/30/2022).

[143] LLVM Project, *LLVM Language Reference Manual - Abstract*. [Online]. Available: `https://llvm.org/docs/LangRef.html#abstract` (visited on 05/08/2022).

[144] Free Software Foundation, Inc., *SSA (GNU Compiler Collection (GCC) Internals)*. [Online]. Available: `https://gcc.gnu.org/onlinedocs/gccint/SSA.html` (visited on 05/08/2022).

[145] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, & tools*, 2nd ed. Boston: Pearson/Addison Wesley, 2007, ISBN: 978-0-321-48681-3.

[146] C. Yarp, *BerkeleySharedMemoryFIFO*. DOI: `10.5281/zenodo.6525781`. [Online]. Available: `https://github.com/ucb-cyarp/BerkeleySharedMemoryFIFO` (visited on 04/03/2022).

[147] ——, *cyclopsRxTestHarness*. DOI: `10.5281/zenodo.6525808`. [Online]. Available: `https://github.com/ucb-cyarp/cyclopsRxTestHarness` (visited on 04/03/2022).

[148] S. Hauck, "Asynchronous design methodologies: An overview," *Proceedings of the IEEE*, vol. 83, no. 1, pp. 69–93, 1995. DOI: `10.1109/5.362752`.

[149] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Commun. ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978, Place: New York, NY, USA Publisher: Association for Computing Machinery, ISSN: 0001-0782. DOI: `10.1145/359576.359579`. [Online]. Available: `https://doi.org/10.1145/359576.359579`.

[150] K. W. Tracy, "Programming Languages," in *Software: A Technical History*, New York, NY, USA: Association for Computing Machinery, 2021, ISBN: 978-1-4503-8724-8. [Online]. Available: `https://doi.org/10.1145/3477339.3477344`.

[151] D. S. Hochbaum, Ed., *Approximation Algorithms for NP-Hard Problems*. Boston: PWS Pub. Co., 1997, ISBN: 0-534-94968-1.

[152] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 9, no. 6, pp. 841–848, 1961. [Online]. Available: `http://www.jstor.org/stable/167050`.

[153] J. A. Fisher, P. Faraboschi, and C. Young, "8.2.3.2 Compaction Techniques," in *Embedded Computing - A VLIW Approach to Architecture, Compilers and Tools*, San Francisco: Elsevier, 2005, pp. 362–365, ISBN: 978-1-55860-766-8. [Online]. Available: `https://app.knovel.com/hotlink/khtml/id:kt0091SWP1/embedded-computing-vliw/compaction-techniques`.

[154] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis amp; transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, Mar. 2004, pp. 75–86. DOI: `10.1109/CGO.2004.1281665`.

[155] C. Lattner, M. Amini, U. Bondhugula, *et al.*, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14. DOI: `10.1109/CGO51591.2021.9370308`.

[156] J. R. Reinders, *Intel C/C++ compilers complete adoption of LLVM*, en, Aug. 2021. [Online]. Available: `https://www.intel.com/content/www/us/en/developer/articles/technical/adoption-of-llvm-complete-icx.html` (visited on 04/28/2022).

[157] NVIDIA/PGI, *Free Fortran, C, C++ Compilers & Tools for CPUs and GPUs*, en. [Online]. Available: `https://www.pgroup.com/index.htm` (visited on 04/28/2022).

[158] Free Software Foundation, Inc., *GCC Releases - GNU Project*. [Online]. Available: `https://gcc.gnu.org/releases.html` (visited on 04/13/2022).

[159] Canonical Ltd., *Details of package clang in bionic*. [Online]. Available: `https://packages.ubuntu.com/bionic/clang` (visited on 05/03/2022).

[160] ——, *Details of package clang in focal*. [Online]. Available: `https://packages.ubuntu.com/focal/clang` (visited on 05/03/2022).

[161] ——, *Details of package clang-10 in bionic*. [Online]. Available: `https://packages.ubuntu.com/bionic/clang-10` (visited on 05/03/2022).

[162] J. Demmel, "Dense Linear Algebra: History and Structure, Parallel Matrix Multiplication," en, CS267 Lectures, p. 18, Feb. 2014. [Online]. Available: `https://people.eecs.berkeley.edu/~demmel/cs267_Spr14/Lectures/lecture12_densela_1_jwd14_v2_4pp.pdf`.

[163] K. Erciyes, *Guide to Graph Algorithms*, en, ser. Texts in Computer Science. Cham: Springer International Publishing, 2018, ISBN: 978-3-319-73235-0. [Online]. Available: `http://link.springer.com/10.1007/978-3-319-73235-0` (visited on 04/16/2022).

[164] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972, https://doi.org/10.1137/0201010. DOI: `10.1137/0201010`. [Online]. Available: `https://doi.org/10.1137/0201010`.

[165] Intel, *Optimize Memory Access Patterns using Loop Interchange and Cache Blocking Techniques*, en. [Online]. Available: `https://www.intel.com/content/www/us/en/develop/documentation/advisor-cookbook/top/optimize-memory-access-patterns.html` (visited on 05/04/2022).

[166] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Fourth edition, ser. Johns Hopkins studies in the mathematical sciences. Baltimore: The Johns Hopkins University Press, 2013, ISBN: 978-1-4214-0794-4.

[167] V. V. Prasolov, *Problems and Theorems in Linear Algebra*, S. Ivanov, Ed., trans. by D. A. Leites, ser. Translations of Mathematical Monographs. Providence, R.I: American Mathematical Society, 1994, vol. 134, ISBN: 978-0-8218-0236-6.

[168] Xilinx, "Power Analysis and Optimization," en, Tutorial UG997 (v2021.2), Jan. 2022, p. 78.

[169] cppreference.com, *Std::atomic::is_lock_free*. [Online]. Available: `https://en.cppreference.com/w/cpp/atomic/atomic/is_lock_free` (visited on 04/18/2022).

[170] M. Posch, "Atomic Operations - Working with the Hardware," en, in *Mastering C++ Multithreading*, Packt, Jul. 2017, ISBN: 978-1-78712-170-6. [Online]. Available: `https://learning.oreilly.com/library/view/mastering-c-multithreading/9781787121706/71d0f9fb-60d2-45d5-9415-13443410f23b.xhtml` (visited on 04/18/2022).

[171] cppreference.com, *Memory_order*. [Online]. Available: `https://en.cppreference.com/w/c/atomic/memory_order` (visited on 04/18/2022).

[172] Microsoft, *Atomic*, en-us. [Online]. Available: `https://docs.microsoft.com/en-us/cpp/standard-library/atomic` (visited on 04/18/2022).

[173] ——, *Lockless Programming Considerations for Xbox 360 and Microsoft Windows - Win32 apps*, en-us. [Online]. Available: `https://docs.microsoft.com/en-us/windows/win32/dxtecharts/lockless-programming` (visited on 03/09/2022).

[174] The Clang Team, *Guaranteed inlined copy.* [Online]. Available: `https://clang.llvm.org/docs/LanguageExtensions.html#guaranteed-inlined-copy` (visited on 04/19/2022).

[175] K. Asanovic, "Lecture 11: Memory," *CS252 Graduate Computer Architecture Lectures*, Fall 2015, p. 37, Oct. 2015. [Online]. Available: `https://inst.eecs.berkeley.edu/~cs252/fa15/lectures/L11-CS252-Memory.pdf`.

[176] Linux Kernel Organization, *Reducing OS jitter due to per-cpu kthreads.* [Online]. Available: `https://www.kernel.org/doc/Documentation/kernel-per-CPU-kthreads.txt` (visited on 04/20/2022).

[177] I. Molnar and M. Krasnyansky, *SMP IRQ affinity.* [Online]. Available: `https://www.kernel.org/doc/html/latest/core-api/irq/irq-affinity.html` (visited on 04/23/2022).

[178] G. Wassen and S. Lankes, "Bare-Metal Execution of Hard Real-Time Tasks Within a General-Purpose Operating System," in *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, F. J. Cazorla, Ed., ser. OpenAccess Series in Informatics (OASIcs), ISSN: 2190-6807, vol. 47, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 75–84, ISBN: 978-3-939897-95-8. DOI: `10.4230/OASIcs.WCET.2015.75`. [Online]. Available: `http://drops.dagstuhl.de/opus/volltexte/2015/5258`.

[179] C. Yarp, *Sir: Simple Interrupt Reporter.* DOI: `10.5281/zenodo.6526433`. [Online]. Available: `https://github.com/ucb-cyarp/sir` (visited on 04/03/2022).

[180] ——, *Benchmarking.* DOI: `10.5281/zenodo.6526289`. [Online]. Available: `https://github.com/ucb-cyarp/benchmarking` (visited on 04/03/2022).

[181] ——, *laminarCommCharacterize.* DOI: `10.5281/zenodo.6526365`. [Online]. Available: `https://github.com/ucb-cyarp/laminarCommCharacterize` (visited on 04/03/2022).

[182] Advanced Micro Devices, "Workload Tuning Guide for AMD EPYC 7002 Series Processor Based Servers," en, Application Note 56745, Nov. 2019, p. 25. [Online]. Available: `https://www.amd.com/system/files/documents/2019-amd-epyc-7002-tg-bios-workload-56745_0.80.pdf`.

[183] C. Yarp, *SchedulingGraphAnalysis.* DOI: `10.5281/zenodo.6526297`. [Online]. Available: `https://github.com/ucb-cyarp/SchedulingGraphAnalysis` (visited on 04/03/2022).

[184] ——, *cyclopsDemo.* DOI: `10.5281/zenodo.6526172`. [Online]. Available: `https://github.com/ucb-cyarp/cyclopsDemo` (visited on 04/03/2022).

[185] ——, *cyclopsASCIILink*. DOI: 10.5281/zenodo.6526155. [Online]. Available: https://github.com/ucb-cyarp/cyclopsASCIILink (visited on 04/03/2022).

[186] ——, *bladeRFToFIFO*. DOI: 10.5281/zenodo.6526260. [Online]. Available: https://github.com/ucb-cyarp/bladeRFToFIFO (visited on 04/03/2022).

[187] ——, *uhdToPipes*. DOI: 10.5281/zenodo.6526260. [Online]. Available: https://github.com/ucb-cyarp/uhdToPipes (visited on 04/03/2022).

[188] B. Stoker, *Dracula*. New York, NY: Grosset & Dunlap, 1897. [Online]. Available: https://www.gutenberg.org/files/345/345-h/345-h.htm (visited on 04/23/2022).

[189] C. Yarp, *Laminar Telemetry Dashboard*. DOI: 10.5281/zenodo.6526250. [Online]. Available: https://github.com/ucb-cyarp/vitisTelemetryDash (visited on 04/03/2022).

[190] ——, *jobQueue*. DOI: 10.5281/zenodo.6526501. [Online]. Available: https://github.com/ucb-cyarp/jobQueue (visited on 04/23/2022).

[191] ——, *platformScripts*. DOI: 10.5281/zenodo.6526495. [Online]. Available: https://github.com/ucb-cyarp/platformScripts (visited on 04/23/2022).

[192] ——, *PlotLaminarPerformance*. DOI: 10.5281/zenodo.6526308. [Online]. Available: https://github.com/ucb-cyarp/PlotLaminarPerformance (visited on 04/03/2022).

[193] Nuand, *bladeRF 2.0 micro xA9*. [Online]. Available: https://www.nuand.com/product/bladerf-xa9/ (visited on 04/24/2022).

[194] ——, *libbladeRF/src/board/bladerf2/common.c/MAX_sample_throughput*, original-date: 2013-02-19T04:12:41Z, Jan. 2020. [Online]. Available: https://github.com/Nuand/bladeRF/blob/29455d29d8b9e58fb18487f886d14b38974e1dfd/host/libraries/libbladeRF/src/board/bladerf2/common.c#L297 (visited on 04/24/2022).

[195] ——, *DC offset and IQ Imbalance Correction*, en. [Online]. Available: https://github.com/Nuand/bladeRF/wiki/DC-offset-and-IQ-Imbalance-Correction#DC_autocalibration_for_a_single_frequency_and_gain (visited on 04/24/2022).

[196] Analog Devices, *AD9361, AD9364 and AD9363 [Analog Devices Wiki]*. [Online]. Available: https://wiki.analog.com/resources/eval/user-guides/ad-fmcomms2-ebz/ad9361 (visited on 04/24/2022).

[197] ——, *MATLAB Filter Design Wizard for AD9361 [Analog Devices Wiki]*. [Online]. Available: https://wiki.analog.com/resources/eval/user-guides/ad-fmcomms2-ebz/software/filters (visited on 04/24/2022).

[198] C. Yarp, *cyclopsDemo_dualinstance*. DOI: 10.5281/zenodo.6526207. [Online]. Available: https://github.com/ucb-cyarp/cyclopsDemo_DualInstance (visited on 04/03/2022).

[199] Advanced Micro Devices, *AMD Ryzen Threadripper 3990X Processor*, en. [Online]. Available: `https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3990x` (visited on 04/24/2022).

[200] ——, *AMD EPYC 75F3*, en. [Online]. Available: `https://www.amd.com/en/products/cpu/amd-epyc-75f3` (visited on 04/24/2022).

[201] A. M. Devices, *AMD EPYC 7763*, en. [Online]. Available: `https://www.amd.com/en/products/cpu/amd-epyc-7763` (visited on 04/24/2022).

[202] A. P. Chandrakasan and R. W. Brodersen, *Low Power Digital CMOS Design*. Norwell: Kluwer Academic Publishers, 1995, ISBN: 0-7923-9576-X.

[203] M. Evers, L. Barnes, and M. Clark, "Next Generation "Zen 3" Core," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–32. DOI: `10.1109/HCS52781.2021.9567108`. [Online]. Available: `https://hc33.hotchips.org/assets/program/conference/day1/HC2021.C1.2%20AMD%20Mark%20Evers.pdf`.

[204] A. Biswas, "Sapphire Rapids," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–22. DOI: `10.1109/HCS52781.2021.9566865`. [Online]. Available: `https://hc33.hotchips.org/assets/program/conference/day1/HC2021.C1.4%20Intel%20Arijit.pdf`.

[205] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019. DOI: `10.1109/IEEESTD.2019.8766229`.

[206] D. R. Gaur, R. Krishnamurti, and R. Kohli, "The capacitated max k-cut problem," en, *Mathematical Programming*, vol. 115, no. 1, pp. 65–72, Sep. 2008, ISSN: 0025-5610, 1436-4646. DOI: `10.1007/s10107-007-0139-z`. [Online]. Available: `http://link.springer.com/10.1007/s10107-007-0139-z` (visited on 04/27/2022).

[207] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998, _eprint: https://doi.org/10.1137/S1064827595287997. DOI: `10.1137/S1064827595287997`. [Online]. Available: `https://doi.org/10.1137/S1064827595287997`.

[208] K. Ebcioglu, R. D. Groves, K.-C. Kim, G. M. Silberman, and I. Ziv, "VLIW Compilation Techniques in a Superscalar Environment," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94, New York, NY, USA: ACM, 1994, pp. 36–48, ISBN: 0-89791-662-X. DOI: `10.1145/178243.178247`. [Online]. Available: `http://doi.acm.org/10.1145/178243.178247`.

[209] E. Stümpel, M. Thies, and U. Kastens, "VLIW compilation techniques for superscalar architectures," in *Compiler Construction*, K. Koskimies, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 234–248, ISBN: 978-3-540-69724-4.

[210] Xilinx, "Xilinx AI Engines and Their Applications," White Paper WP506 (v1.1), Jul. 2020, p. 13. [Online]. Available: `https://www.xilinx.com/content/dam/xilinx/support/documents/white_papers/wp506-ai-engine.pdf`.

[211] S. Weinstein and P. Ebert, "Data Transmission by Frequency-Division Multiplexing Using the Discrete Fourier Transform," *IEEE Transactions on Communication Technology*, vol. 19, no. 5, pp. 628–634, 1971. DOI: `10.1109/TCOM.1971.1090705`.

[212] Y. Li and G. L. Stüber, Eds., *Orthogonal Frequency Division Multiplexing for Wireless Communications*, en, ser. Signals and Communication Technology. Boston: Kluwer Academic Publishers, 2006, ISBN: 978-0-387-29095-9. [Online]. Available: `http://link.springer.com/10.1007/0-387-30235-2` (visited on 01/21/2022).

[213] MathWorks, *Phase/Frequency Offset*. [Online]. Available: `https://www.mathworks.com/help/comm/ref/phasefrequencyoffset.html` (visited on 02/26/2022).

[214] P. Rigge, J. Wright, and C. Yarp, *A Compile-Time and Run-Time Configurable Viterbi Decoder in Chisel Hardware Construction Language: CS250 Project Report*, Dec. 2014.

[215] B. Nikolic, *"Viterbi and Turbo Decoders." EE290C Sp14. UC Berkeley*, 2014.

[216] J. G. Proakis, M. Salehi, and G. Bauch, "10.3 Channel Coding," in *Contemporary communication systems using MATLAB*, 3rd ed, Stamford, CT: Cengage Learning, 2013, pp. 440–472, ISBN: 978-0-495-08251-4.

[217] C. Yarp, *cyclopsASCIILink-testfiles*. DOI: `10.5281/zenodo.6526156`. [Online]. Available: `https://github.com/ucb-cyarp/cyclopsASCIILink-testfiles` (visited on 04/03/2022).

[218] ——, *cpuTopology: CPU Topology Scripts from Benchmarking*. DOI: `10.5281/zenodo.6526510`. [Online]. Available: `https://github.com/ucb-cyarp/cpuTopology` (visited on 04/16/2022).

Appendix A

Baseband Signal Processing

A.1 Alternative to Single Carrier: Orthogonal Frequency Division Multiplexing (OFDM)

The radio signal processing discussed in chapter 2 is commonly known as single carrier. It involves transmitting one modulated signal at a specified carrier frequency. An alternative approach is to split the occupied bandwidth into multiple small segments, referred to a sub-carriers. Data is mapped to points on the complex plane of each sub-carrier, much in the same way that digital modulation works with single carrier radios.

One method to process these subcarriers would be to produce multiple single carrier radios, each tuned to a different subcarrier. Because the lowpass filtering in single carrier radios realistically has some degree of roll-off, a guard band would need to be inserted between subcarriers to reduce crosstalk. OFDM takes a different approach and leverages orthogonality properties of the Discrete Fourier Transform (DFT). Each bin of the DFT can be viewed as a correlation of a discrete signal with a discrete complex sinusoid over a fixed number of samples. The spacing of bin center frequencies is such that a complex sinusoid at a given bin's center frequency is only visible in that bin of the FFT and is zero in other bins[1]. OFDM works by assigning modulated symbols to each bin and then computing the inverse DFT [212]. This time domain signal is then sent over the air and is run through a DFT in the receiving radio to retrieve the modulated symbols in each subcarrier [212].

One of the advantages of OFDM is the simplification of the equalization process. With a sufficiently large number of subcarriers, the frequency response of the channel for a given subcarrier is modeled as flat [212]. This can be corrected by a single complex multiply for each subcarrier. Channel estimation can be accomplished by sending known signals (pilots) on designated subcarriers and analyzing their response. OFDM is also able to leverage the Fast Fourier Transform (FFT) implementation of the DFT for implementation efficiency.

While elegant in theory and in practice, there are a few things to keep in mind about OFDM:

[1]See [211], [212] for details of the orthogonality property of the DFT used by OFDM.

- While OFDM is spectrally efficient without the need for guard bands, some efficiency is lost in the time domain in the form of the Cyclic Prefix (CP). This is the repetition of time domain samples from the beginning of OFDM transmission. It serves both as a method to allow time synchronization as well as playing a key role in equalization. The CP should be as long as the longest propagation path in the environment (longest tap in the channel impulse response) [212].
- Crosstalk between bins can occur when a carrier frequency offset (CFO) is present. Frequency shifting of the input signal results in non-orthogonality between the bins in the receive DFT. As a result, CFO estimation and correction is exceptionally important for OFDM performance.
- OFDM signals tend to have high peak-to-average power ratios compared to other modulation schemes [212]. This puts additional linearity requirements on the amplifiers in the system, particularly on the power amplifier on the Tx side.

For more information on OFDM radios, their structure, advantages, and disadvantages, see [211], [212].

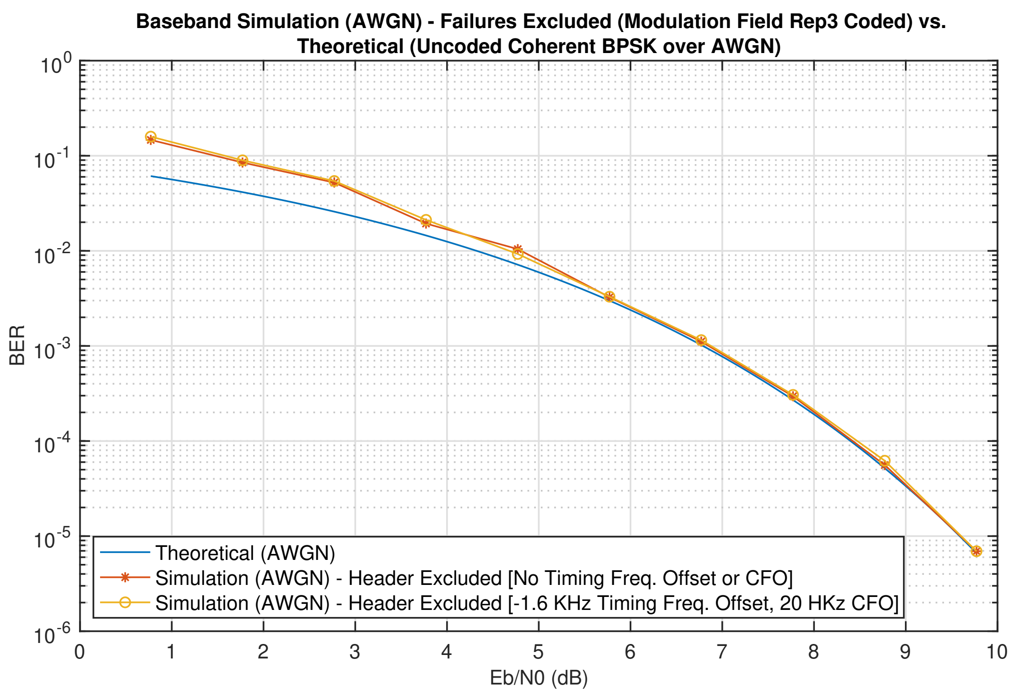
A.2 Cyclops Baseband Performance - All Modulation Schemes

The following plots detail the performance of the Cyclops Radio (Rev 1.40) when simulated in Matlab/Simulink. 100 Trials were conducted per point. The sample rate of the system was simulated to be at 80 MHz with the carrier frequency at 80 MHz[2]. The selection of carrier frequency only effects the relative effect of the CFO. The BER vs. EbN0 curves for all supported modulation schemes are shown in Figure A.1. Note that these plots exclude packets which fail detection or decoded the incorrect modulation scheme, effectively resulting in a whole packet loss. Complete packet decoding and repetition decoding failures in the modulation field are shown in Figure A.2. Only the plots for BPSK and QPSK are included as no packet decode failures occurred for the 16QAM and 256QAM runs.

A.3 Convolutional Encoding and Viterbi Decoding

Convolutional encoding is one strategy used for Forward Error Corrections (FEC). Like other FEC algorithms, it works by sending out some redundant information which can be used at the receiver to correct errors that occur at reception. In this context, errors refer to bit flips (when a 0 is turned into a 1 or a 1 is turned into a 0) and not to the loss of bits. The *rate* of the code indicates the number of information bits per coded bits sent. For

[2]This is due to using the Simulink Phase/Frequency Offset Block [213] which is referenced to the sample frequency at the input. To accelerate simulation times, the carrier effects were simulated at the sample rate.



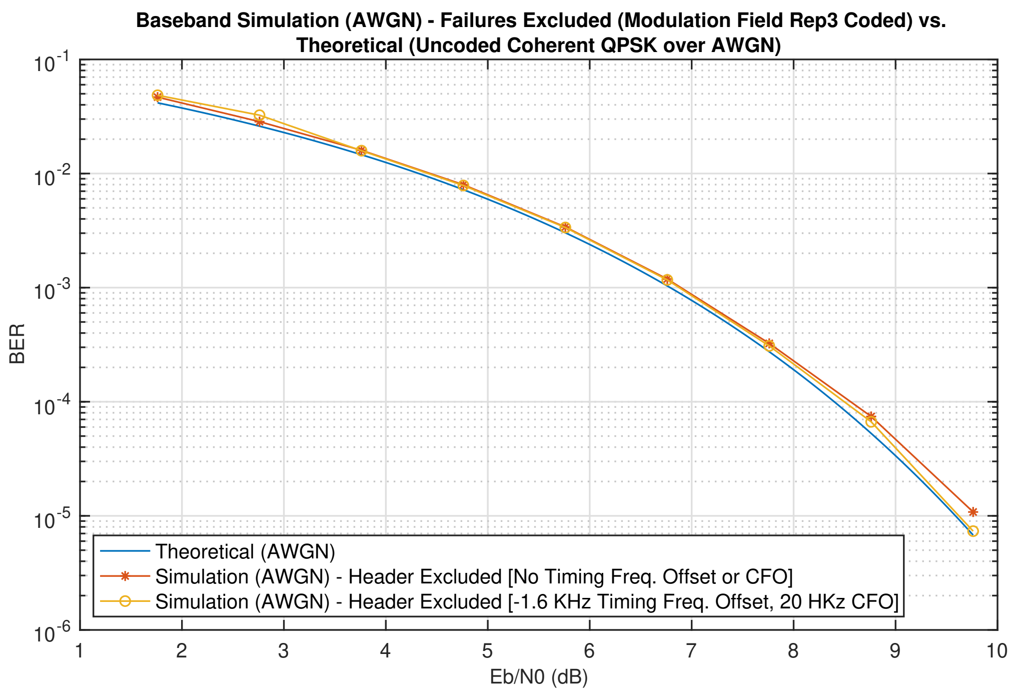
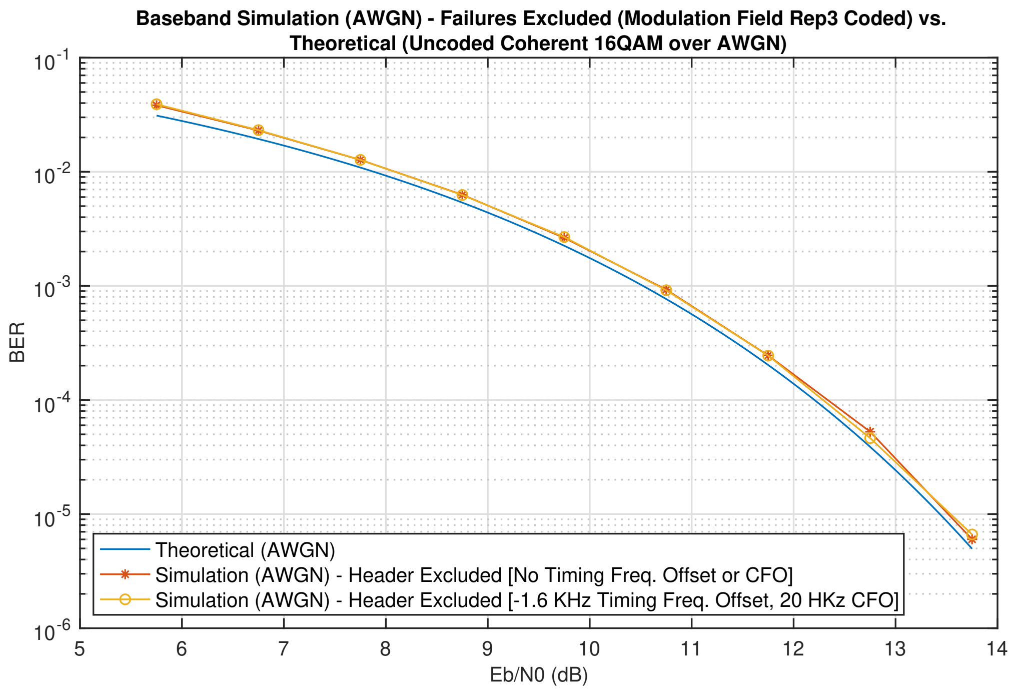


Figure A.1: Cyclops Baseband BER Performance - Rev 1.40



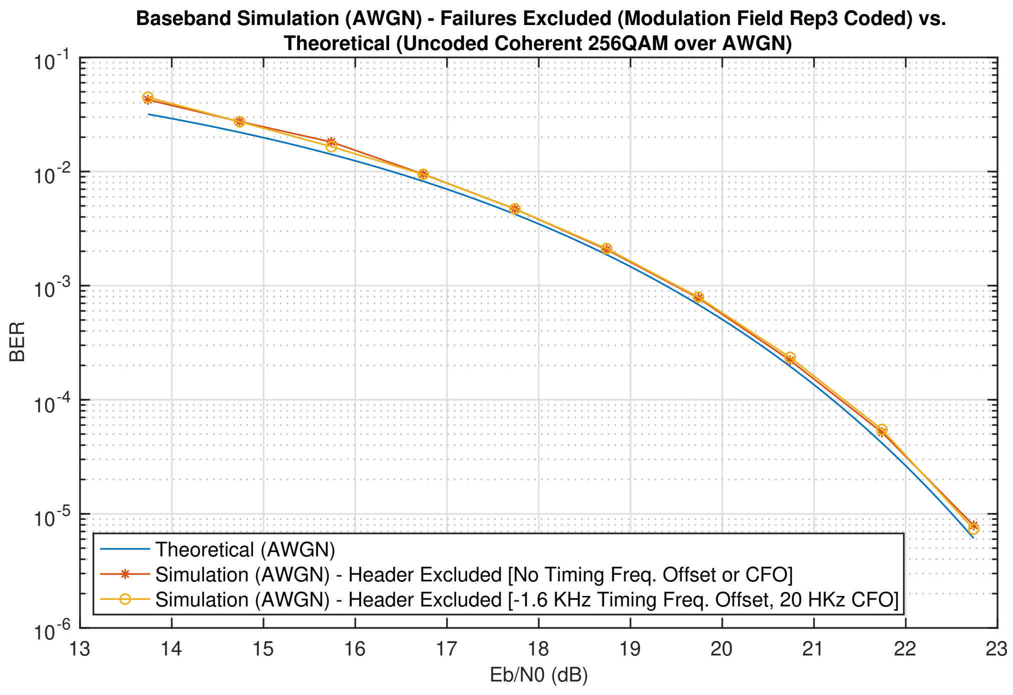
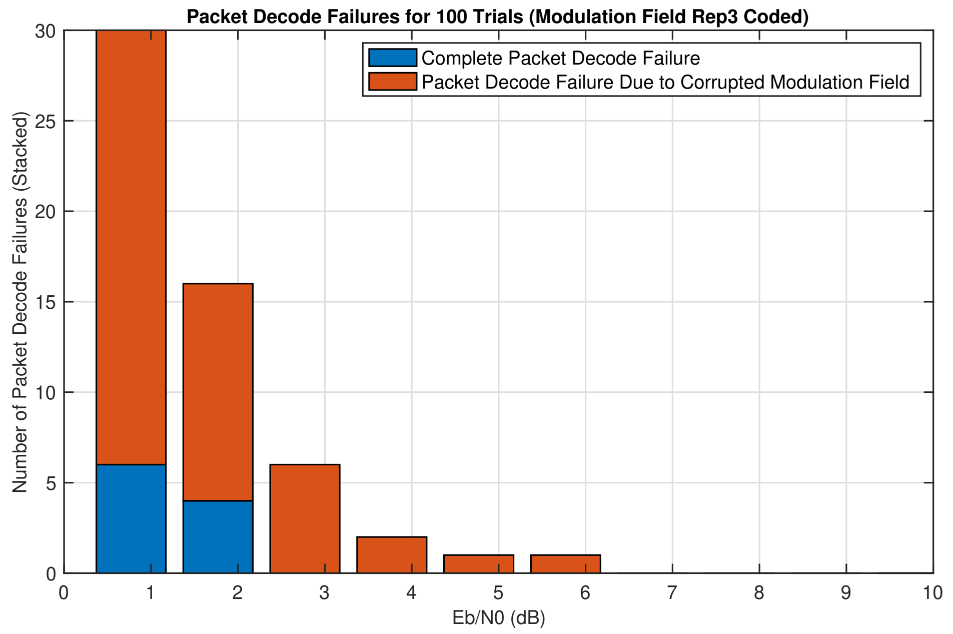
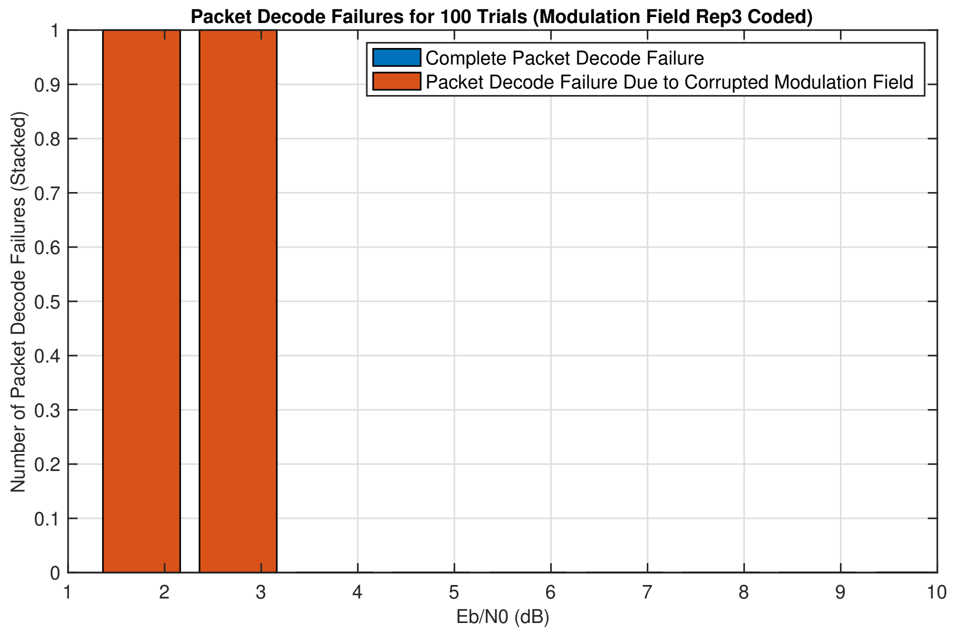


Figure A.1: Cyclops Baseband BER Performance - Rev 1.40

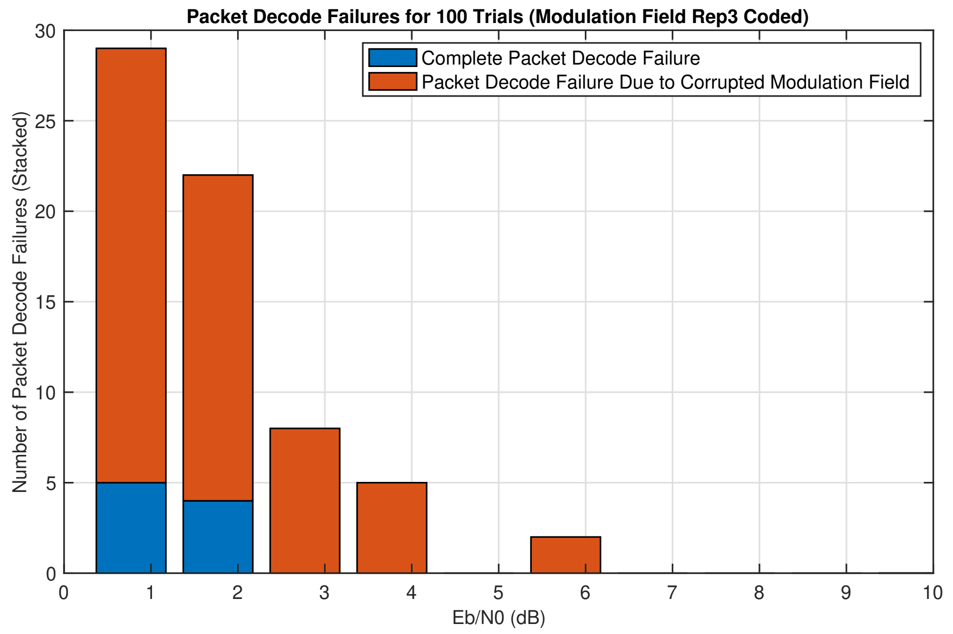


(a) BPSK, No Timing Frequency Error or CFO

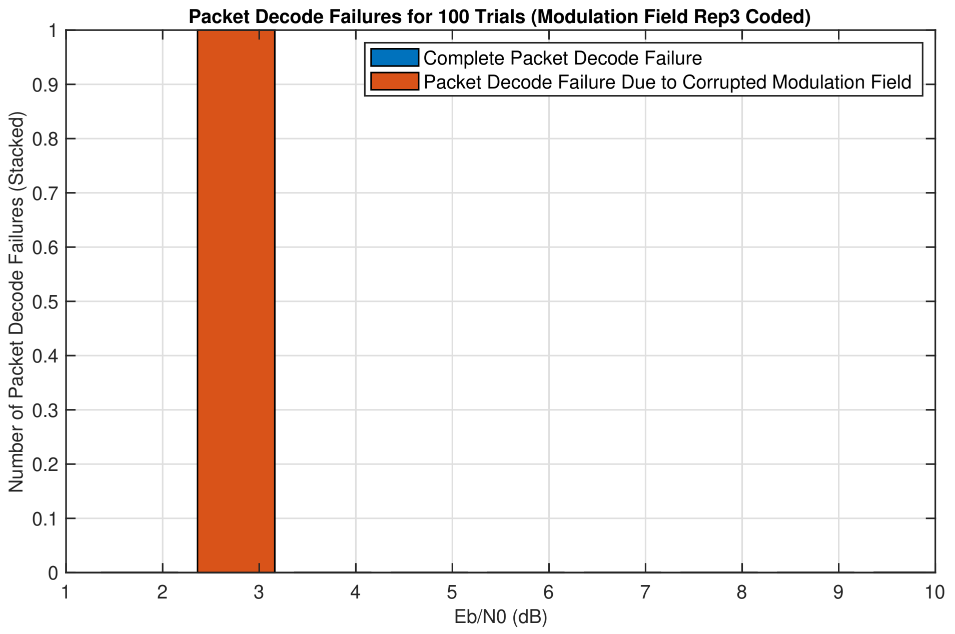


(b) QPSK, No Timing Frequency Error or CFO

Figure A.2: Cyclops Baseband Packet Detect Failures - Rev 1.40



(c) BPSK - Timing Frequency Error -1.6 KHz, CFO 20 KHz

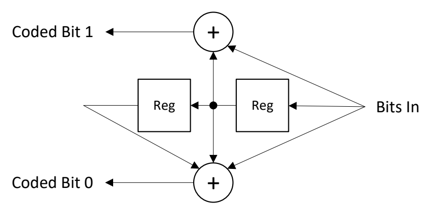


(d) QPSK - Timing Frequency Error -1.6 KHz, CFO 20 KHz

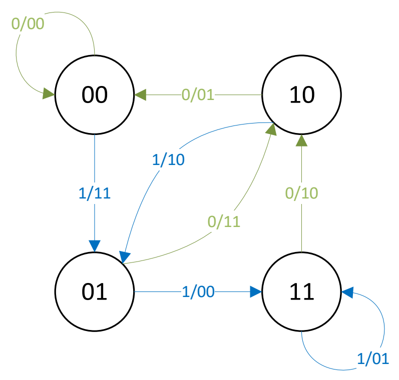
Figure A.2: Cyclops Baseband Packet Detect Failures - Rev 1.40

instance, a rate 1/2 code sends 2 coded bits for every bit of information to be communicated. Generally, the lower the rate (the more redundant data sent), the higher the probability of correctly decoding at the receiver.

Convolutional encoders, as their name suggests, have a very similar structure to FIR filters. Bits of data to be sent are fed into a shift register and outputs are computed from values in the shift register (represented by polynomials). Specifically, the different coded bits are the result of XOR-ing different taps from the shift register (including the zero-delay element). The set of taps to XOR for each bit is described by a generator polynomial which is often represented in a one-hot encoded binary or octal number with each bit representing a tap to XOR. An example rate 1/2 convolutional encoder is shown in Figure A.3a. Before encoding starts, the state of the shift register is initialized to some fixed state with the all-zero state being common. As bits are fed into the encoder, the state in the shift register changes. Since the behavior of the encoder depends on the state, it can be helpful to view it as a Finite State Machine (FSM). An example of the FSM view of an encoder is shown in Figure A.3b. Each node represents the state held in the shift register and is shown in the diagram in binary form with the right most digit representing the last information bit to be sent. The label of each edge shows the incoming information bit as well as the resulting coded bits to be sent. The format is *information bit/coded bits*.



(a) Shift Register Implementation

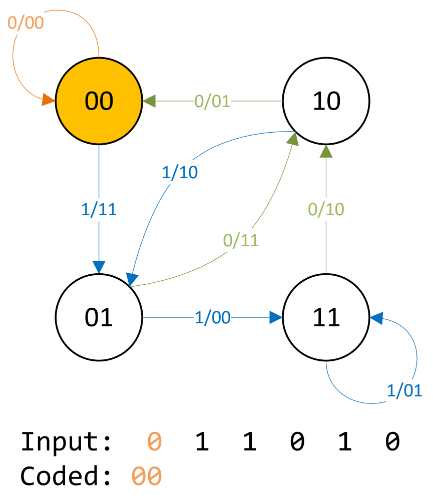


(b) Finite State Machine (FSM) View

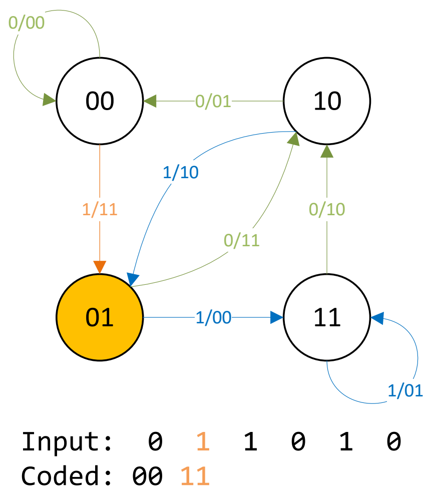
Figure A.3: Example Convolutional Encoder - From [214], Adapted from [215]

An example of convolutionally encoding a sequence of bits in the encoder described in Figure A.3a with an initial state of 00 is shown in Figure A.4. The active encoder state as well as the arc being activated is highlighted in yellow. Note that the active state of the encoder can change with each encoded bit.

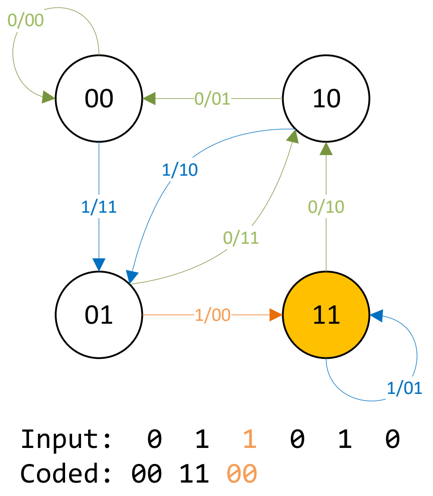
The final result of encoding bits 0 1 1 0 1 0, read from left to right, is 00 11 00 10 10 11. Note that because this is a rate 1/2 code, the number of coded bits is double the number of



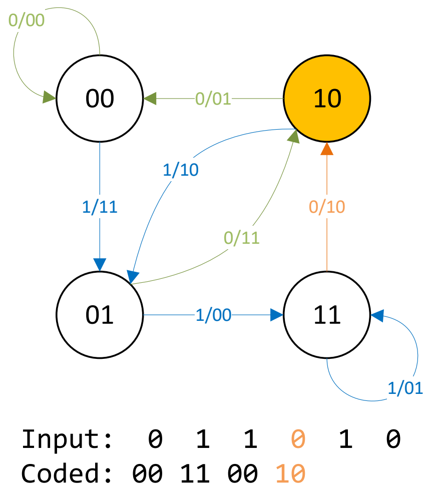
(a) Bit 1



(b) Bit 2



(c) Bit 3



(d) Bit 4

Figure A.4: Example Convolutionally Encoding Bits

information bits. These coded bits are what would be sent out of the transmitter into the air.

At the receiver side, the objective is to turn the possibly corrupted coded bits back into the original information bits. The Viterbi algorithm, originally published in [114] and later expanded to be more accessible in [115], is a popular solution to the problem. Viterbi's algorithm is a particularly appealing because it provides a dynamic programming [116] solution to perform the *maximum likelihood* estimate of the convolutional decoding[3]. The maximum likelihood property was not known in the original paper but was later shown by G.D. Forney, as noted by [115], and is documented in his influential tutorial paper [117]. As described by Forney, the Viterbi algorithm is "a solution to the problem of maximum a posteriori probability (MAP) estimation of the state sequence of a finite state discrete-time Markov process observed in memoryless noise" [117].

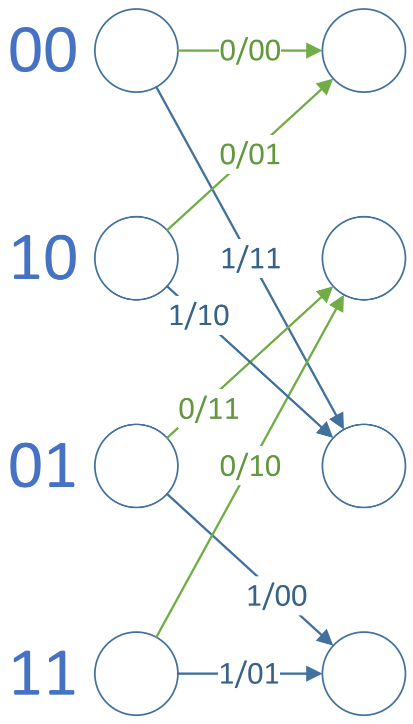
The maximum likelihood decoding can be described in the following way: given a convolutional encoder which can generate a coded sequence: $x_1, x_2, \cdots, x_n$ and a received coded sequence $y_1, y_2, \cdots, y_n$, find the coded sequence $x_1, x_2, \cdots, x_n$ which has the minimum Hamming Distance[4] from the received sequence $y_1, y_2, \cdots, y_n$ [115]. It is important to note that $x_1, x_2, \cdots, x_n$ is restricted to only be a legal coded sequence which can be generated by the convolutional coder. Assuming bit errors are *independent* and all input sequences are equally probable, the coded sequence with minimum Hamming Distance from the received sequence was the one most likely transmitted [115]. Given the most likely coded sequence, we can derive the associated uncoded sequence. The maximum likelihood property was discussed for Binary Symmetric Channels (BSC) and Additive White Gaussian Noise (AWGN) channels in [115].

The algorithm is most easily described by re-organizing the FSM view of the convolutional decoder into a *trellis* which shows each iteration through the FSM as a separate column of nodes. The trellis view is often expanded to show multiple stages of the trellis. A single iteration of the trellis for the convolutional encoder described in Figure A.3 is shown in Figure A.5a. The Viterbi coder works by traversing the trellis for each received coded segment. For each edge in the trellis, it computes the hamming distance (number of bits different) between the received coded segment and the coded output of that edge. That hamming distance is added to a metric stored in the node at the source of the edge which represents the Hamming Distance up to that point. The destination node selects the path with the minimum *path metric* (representing the Hamming Distance of that path) with that edge becoming the *survivor*. The new metric is recorded as well as the surviving path. The process is then repeated on the next coded sequence. An example of this process is shown in Figure A.6a for a corrupted version of the coded sequence generated in the earlier encoding example. The surviving paths are in bold with the discarded paths in gray. One reception is complete, the node with the minimum metric is selected and the coded sequence

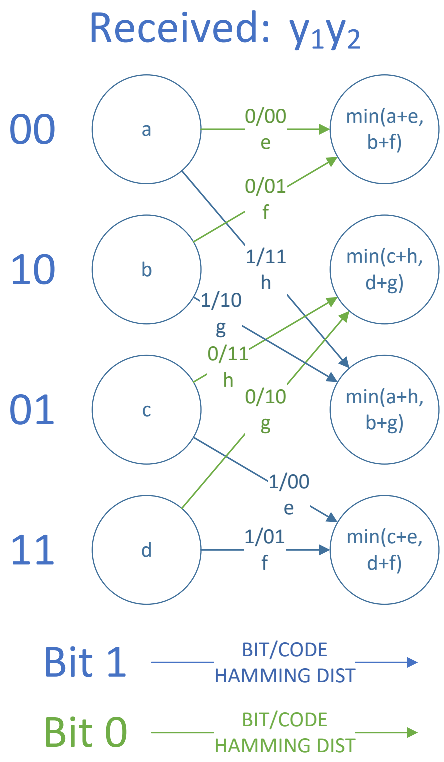
[3]Under the constraint that the traceback includes the entire message. In practice, the traceback length is limited to simplify the implementation.

[4]The Hamming Distance between two bit sequences is the number of bits which differ between them.

is re-assembled by tracing back the surviving paths leading to this node. An example of the traceback phase is shown in Figure A.6b. Note that despite two bit errors, the correct sequence was decoded.



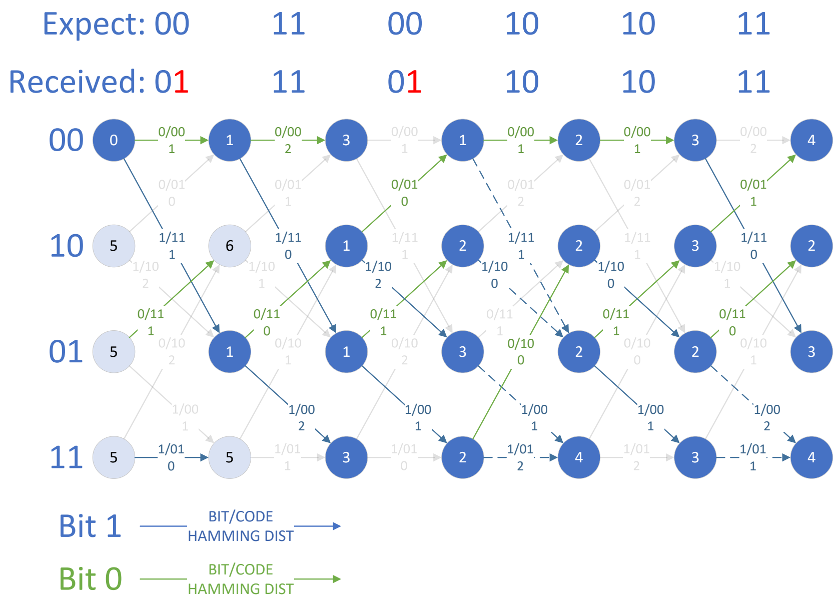
(a) 1 Iteration of Trellis



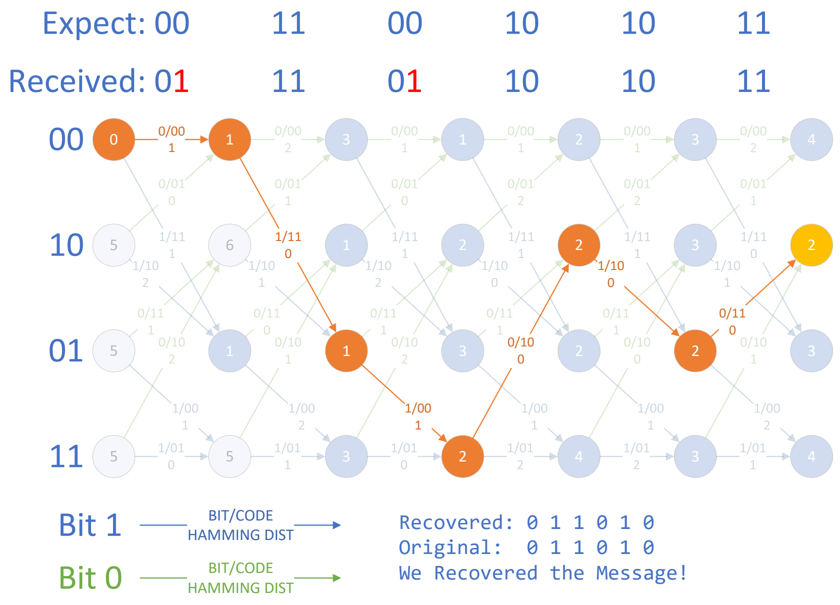
(b) Viterbi Trellis Operations

Figure A.5: Example Viterbi Decoder Trellis for k=1, Generators 0b111, 0b110 - From [214], Adapted from [215]

It is important to note that, even though multiple iterations of the trellis are shown in Figure A.6, hardware and software implementations typically only implement a small number of columns (often one) which are then used for successive coded sequences. The fully unrolled trellis diagram is mostly helpful to illustrate the multiple phases of the Viterbi algorithm. There are also many variations to the Viterbi algorithm such as path metric re-normalization and periodic traceback which are not discussed here. For more details including analysis of the strengths of convolutional codes, known good generator polynomials, and methods for providing different rate codes, see texts such as [116], [216].


Expect: 00 11 00 10 10 11
Received: 01 11 01 10 10 11
00
10
01
11
Bit 1 BIT/CODE HAMMING DIST
Bit 0 BIT/CODE HAMMING DIST

(a) Forward Pass


Expect: 00 11 00 10 10 11
Received: 01 11 01 10 10 11
00
10
01
11
Bit 1 BIT/CODE HAMMING DIST
Bit 0 BIT/CODE HAMMING DIST

```
Recovered: 0 1 1 0 1 0
Original:  0 1 1 0 1 0
We Recovered the Message!
```

(b) Traceback

Figure A.6: Example Viterbi Decoder for k=1, Generators 0b111, 0b110 - From [214], Adapted from [215]

Appendix B

Demonstration System

B.1 Final Test System Configuration

- CPU: AMD Ryzen Threadripper 3970X
- Motherboard: Asus ROG Zenith II Extreme
- Memory: 4 DIMMs - Corsair CMK32GX4M4B3600C18 8 GiB DDR4 3600 MHz
  - XMP/DOCP Profile Enabled in BIOS for DRAM Voltage and Timing Tuning
  - Limited to 3200 MHz, Maximum Supported by CPU
  - Quad-Channel Supported by CPU
- Operating System: Ubuntu Server 18.04 LTS
- Primary Compiler: AOCC (AMD Optimizing C/C++ Compiler) 3.0.0

B.2 Grub Configuration

The following grub config file located at `/etc/default/grub` in Ubuntu 18.04. After the file is updated `sudo update-grub` must be executed and the system rebooted for changes to take effect. Note that nohz_full does not enable full dynamic ticks unless the kernel is compiled to support it. The default kernel provided with Ubuntu does not have this option enabled. It was determined in experiments with Reduced Handshake FIFOs that nohz_full can cause interrupt clusters.

```
# If you change this file, run 'update-grub' afterwards to update
# /boot/grub/grub.cfg.
# For full documentation of the options in this file, see:
#   info -f grub -n 'Simple configuration'

GRUB_DEFAULT=0
```

```
GRUB_TIMEOUT_STYLE=hidden
GRUB_TIMEOUT=2
GRUB_DISTRIBUTOR=`lsb_release -i -s 2> /dev/null || echo Debian`

#Current Command Line of Choice
GRUB_CMDLINE_LINUX_DEFAULT="acpi_irq_nobalance noirqbalance irqaffinity=0
    nosoftlockup nmi_watchdog=0 rcu_nocbs=1-31 nohz_full=1-31 isolcpus=1-31
    " #nohz_full implies rcu_nocbs so its inclusion here is redundant but
    not harmful

#...
```

Listing B.1: Grub Configuration (Summarized)

Appendix C

Software

C.1 Source Code

The source code for this project is available as part of git repositories stored on GitHub and through the Zenodo archiving service. Most repositories are licensed as a BSD-3-Clause except uhdToPipes which is licensed as GPL-3.0 and sir which is dual licensed as BSD/GPL. A list of public repositories is given in Table C.1.

Note that some repositories, such as cyclopsDemo, cyclopsDemo_DualInstance, and cyclopsRxTestHarness use git submodules. If cloning from GitHub, be sure to run `git submodule update --init` inside the cloned directory. If using the Zenodo archive, submodules are not automatically packaged into the compressed archive. The user will need to download the appropriate dependency and place it in the `submodules` directory, replacing the blank placeholder folder.

Repository Name	GitHub URL	Git Tag	Zenodo DOI	Ref
Laminar (vitis)	`https://github.com/ucb-cyarp/vitis`	v1.0.1	`10.5281/zenodo.6525757`	[130]
cyclopsbb-pub	`https://github.com/ucb-cyarp/cyclopsbb-pub`	v1.0.1	`10.5281/zenodo.6525784`	[73]
cyclopsRxTestHarness	`https://github.com/ucb-cyarp/cyclopsRxTestHarness`	v1.0.2	`10.5281/zenodo.6525808`	[147]
cyclopsDemo (Single Instance Variant)	`https://github.com/ucb-cyarp/cyclopsDemo`	v1.0.1_single_inst	`10.5281/zenodo.6526172`	[184]

Table C.1: Project Source Code Repositories (continued on next page ...)

Repository Name	GitHub URL	Git Tag	Zenodo DOI	Ref
cyclopsDemo (Dual Instance Variant)	`https://github.com/ucb-cyarp/cyclopsDemo`	v1.0.1 _dual_inst	`10.5281/zenodo.6526163`	[184]
cyclopsDemo_DualInstance	`https://github.com/ucb-cyarp/cyclopsDemo_DualInstance`	v1.0.1	`10.5281/zenodo.6526207`	[198]
cyclopsASCIILink (Single Instance Variant)	`https://github.com/ucb-cyarp/cyclopsASCIILink`	v1.0.3 _single_inst	`10.5281/zenodo.6526155`	[185]
cyclopsASCIILink (Dual Instance Variant)	`https://github.com/ucb-cyarp/cyclopsASCIILink`	v1.0.3 _dual_inst	`10.5281/zenodo.6526154`	[185]
cyclopsASCIILink-testfiles	`https://github.com/ucb-cyarp/cyclopsASCIILink-testfiles`	v1.0.1	`10.5281/zenodo.6526156`	[217]
bladeRFToFIFO	`https://github.com/ucb-cyarp/bladeRFToFIFO`	v1.0.1	`10.5281/zenodo.6526260`	[186]
uhdToPipes	`https://github.com/ucb-cyarp/uhdToPipes`	v1.0.1	`10.5281/zenodo.6526260`	[187]
BerkeleySharedMemoryFIFO	`https://github.com/ucb-cyarp/BerkeleySharedMemoryFIFO`	v1.0.3	`10.5281/zenodo.6525781`	[146]
Laminar Telemetry Dashboard (vitisTelemetryDash)	`https://github.com/ucb-cyarp/vitisTelemetryDash`	v1.0.1	`10.5281/zenodo.6526250`	[189]
PlotLaminarPerformance	`https://github.com/ucb-cyarp/PlotLaminarPerformance`	v1.0.1	`10.5281/zenodo.6526308`	[192]
SchedulingGraphAnalysis	`https://github.com/ucb-cyarp/SchedulingGraphAnalysis`	v1.0.1	`10.5281/zenodo.6526297`	[183]
benchmarking	`https://github.com/ucb-cyarp/benchmarking`	v1.0.1	`10.5281/zenodo.6526289`	[180]

Table C.1: Project Source Code Repositories (continued on next page ...)

Repository Name	GitHub URL	Git Tag	Zenodo DOI	Ref
laminarCommCharacterize	`https://github.com/ucb-cyarp/laminarCommCharacterize`	v1.0.1[1]	`10.5281/zenodo.6526365`	[181]
ConvolutionalEncDec	`https://github.com/ucb-cyarp/ConvolutionalEncDec`	v1.0.2	`10.5281/zenodo.6526429`	[121]
sir	`https://github.com/ucb-cyarp/sir`	v1.0.2	`10.5281/zenodo.6526433`	[179]
cpuTopology	`https://github.com/ucb-cyarp/cpuTopology`	v1.0.1	`10.5281/zenodo.6526510`	[218]
jobQueue	`https://github.com/ucb-cyarp/jobQueue`	v1.0.1	`10.5281/zenodo.6526501`	[190]
platformScripts	`https://github.com/ucb-cyarp/platformScripts`	v1.0.1	`10.5281/zenodo.6526495`	[191]

Table C.1: Project Source Code Repositories

[1] Multiple variants of these tests exist. The git tag and Zenodo version numbers are appended with the variant. Use the GitHub tag browser or the Zenodo Concept DOI to access the different variants.

Appendix D

Laminar Generation Samples

This section contains some Laminar-generated code for the Cyclops RRC filter under different Laminar configurations. The disassembly for these functions is also provided below, with some comments added to aid in reading. It should be noted that variable names in the C code along with comments were modified for readability.

As noted in section 10.3, there are still opportunities to improve the Laminar compiler. One such optimization includes fixing a Laminar bug which caused some unused arrays to be emitted in the Listing D.2 segment. These arrays were removed for clarity from the code listing in this document as they are easily optimized out by the C compiler. Another potential improvement would be to implement `memcpy` as a mode to handle intermediate copies to the output. This could potentially prevent unnecessary memory movement instructions created due to the unrolling of the output loop shown in Listing D.4. Ideally, the real and imaginary components would not be interleaved and would be copied contiguously. An even better result would be the optimizing away of the accumulators in memory as they can fit in vector registers.

It should also be noted that the compiler inlined the compute function into the outer thread function for the implementation without sub-blocking shown in Listing D.1. However, the compiler did not inline the sub-blocked version shown in Listing D.2.

D.1 Generated Source

D.1.1 RRC, No Sub-Blocking

```c
void rx_demo_partition1_compute(Partition1_state_t *stateStruct,
                                  const float InputSamples_re[120],
                  const float InputSamples_im[120],
                  float OutputSamples_re[120],
                  float OutputSamples_im[120])
{
  for (uint8_t blockingIdx = 0; blockingIdx < 120; blockingIdx++)
  {
    float BlockingInput_re = InputSamples_re[blockingIdx];
    float BlockingInput_im = InputSamples_im[blockingIdx];

    //---- Calculate TappedDelay ----
    memcpy(stateStruct->TappedDelay_state_re +
           stateStruct->TappedDelay_headIdx,
         &BlockingInput_re, sizeof(float) * 1);
    memcpy(stateStruct->TappedDelay_state_re +
           stateStruct->TappedDelay_headIdx - 64,
         &BlockingInput_re, sizeof(float) * 1);
    memcpy(stateStruct->TappedDelay_state_im +
           stateStruct->TappedDelay_headIdx,
         &BlockingInput_im, sizeof(float) * 1);
    memcpy(stateStruct->TappedDelay_state_im +
           stateStruct->TappedDelay_headIdx - 64,
         &BlockingInput_im, sizeof(float) * 1);

    //---- Calculate InnerProduct ----
    float InnerProduct_Accum_re = ((float)0);
    float InnerProduct_Accum_im = ((float)0);
    for (unsigned long indDim0 = 0; indDim0 < 49; indDim0++)
    {
      InnerProduct_Accum_re += (((float)(Coefs_re[indDim0])) *
        (((float)((stateStruct->TappedDelay_state_re +
        stateStruct->TappedDelay_headIdx - 48)[indDim0])));

      InnerProduct_Accum_im += (((float)(Coefs_re[indDim0])) *
        (((float)((stateStruct->TappedDelay_state_im +
        stateStruct->TappedDelay_headIdx - 48)[indDim0])));
    }

    //---- State Update for TappedDelay~~~~
    stateStruct->TappedDelay_headIdx =
      ((stateStruct->TappedDelay_headIdx + 1) % 64) + 64;

    (OutputSamples_re[blockingIdx]) = InnerProduct_Accum_re;
    (OutputSamples_im[blockingIdx]) = InnerProduct_Accum_im;
  }
}
```

Listing D.1: Laminar Generated Cyclops Rx RRC Partition, Blocking: 120, Sub-Blocking: Disabled, Comments and Variable Names Changed for Clarity

D.1.2 RRC, Sub-Blocked

```c
void rx_demo_partition1_compute(Partition1_state_t *stateStruct,
                                const float InputSamples_re[120],
                   const float InputSamples_im[120],
```

```
                float OutputSamples_re[120],
                float OutputSamples_im[120])
{
  //Note: A Laminar bug emitted 3 temporary arrays that are unused.
  //These can be easily optimized out by the C compiler.  Removed here.
  //Split into 5 sub-blocks of 24 elements each
  for (uint8_t blockingIdx = 0; blockingIdx < 5; blockingIdx++)
  {
    //---- Calculate TappedDelay (Moving in Whole 24 Element
           Sub-Block) ----
    memcpy(stateStruct->TappedDelay_state_re +
           stateStruct->TappedDelay_headIdx,
         (InputSamples_re + blockingIdx * 24),
         sizeof(float) * 24);
    memcpy(stateStruct->TappedDelay_state_re +
           stateStruct->TappedDelay_headIdx - 72,
         (InputSamples_re + blockingIdx * 24),
         sizeof(float) * 24);
    memcpy(stateStruct->TappedDelay_state_im +
           stateStruct->TappedDelay_headIdx,
         (InputSamples_im + blockingIdx * 24),
         sizeof(float) * 24);
    memcpy(stateStruct->TappedDelay_state_im +
           stateStruct->TappedDelay_headIdx - 72,
         (InputSamples_im + blockingIdx * 24),
         sizeof(float) * 24);

    //---- Calculate InnerProduct (24 Independent Dot Products) -
    //     Specialized Blocking Implementation Iterates over Taps
    //     in the Outer Loop ----
    float InnerProduct_Accum_re[24] = {(float)0, (float)0, (float)0,
         (float)0, (float)0, (float)0, (float)0, (float)0, (float)0,
       (float)0, (float)0, (float)0, (float)0, (float)0, (float)0,
       (float)0, (float)0, (float)0, (float)0, (float)0, (float)0,
       (float)0, (float)0, (float)0};
    float InnerProduct_Accum_im[24] = {(float)0, (float)0, (float)0,
         (float)0, (float)0, (float)0, (float)0, (float)0, (float)0,
       (float)0, (float)0, (float)0, (float)0, (float)0, (float)0,
       (float)0, (float)0, (float)0, (float)0, (float)0, (float)0,
       (float)0, (float)0, (float)0};
    for (unsigned long tapIdx = 0; tapIdx < 49; tapIdx++)
    {
      for (unsigned long subBlockingIdx = 0; subBlockingIdx < 24;
           subBlockingIdx++)
      {
        InnerProduct_Accum_re[subBlockingIdx] +=
          (((float)(Coefs_re[tapIdx]))) *
          (((float)(((stateStruct->TappedDelay_state_re) +
            ((stateStruct->TappedDelay_headIdx - 48) +
            subBlockingIdx))[tapIdx])));

        InnerProduct_Accum_im[subBlockingIdx] +=
          (((float)(Coefs_re[tapIdx]))) *
          (((float)(((stateStruct->TappedDelay_state_im) +
            ((stateStruct->TappedDelay_headIdx - 48) +
            subBlockingIdx))[tapIdx])));
      }
    }

    //---- State Update for TappedDelay (Branching Technique Since
    //      not Power of 2) ----
    if (stateStruct->TappedDelay_headIdx >= 120)
    {
      stateStruct->TappedDelay_headIdx = 72;
    } else {
      stateStruct->TappedDelay_headIdx += 24;
    }

    for (unsigned long tapIdx = 0; tapIdx < 24; tapIdx++)
    {
      (OutputSamples_re + (24 * blockingIdx))[tapIdx] =
        InnerProduct_Accum_re[tapIdx];
      (OutputSamples_im + (24 * blockingIdx))[tapIdx] =
        InnerProduct_Accum_im[tapIdx];
    }
  }
}
```

Listing D.2: Laminar Generated Cyclops Rx RRC Partition, Blocking: 120, Sub-Blocking: 24, Comments and Variable Names Changed for Clarity, Erroneously Emitted Unused Arrays Removed for Clarity

D.2 Disassembly

D.2.1 RRC, No Sub-Blocking

```
; ================ BEGINNING OF PROCEDURE ================

  rx_demo_partition1_compute:
  mov           al, byte [rdi]                          ; Begin of unwind
                                                        ; block (FDE at
                                                        ; 0x20df0c)
  vmovaps       ymm8, aXdaxfexe0rxfex+96                ; 0x201140
  vmovaps       ymm9, aXdaxfexe0rxfex+32                ; 0x201100
  vmovaps       ymm10, aXdaxfexe0rxfex                  ; aXdaxfexe0rxfex
  vmovaps       ymm3, aX80xedxbcxbatx+98                ; 0x2011e0
  vmovaps       ymm4, aXdaxfexe0rxfex+64                ; 0x201120
  vmovaps      ymm5, aX80xedxbcxbatx+2              ; 0x201180
  vmovss       xmm6, dword [aX80xbbx160x013+14]     ; 0x2010a4
  xor          r9d, r9d
  nop          dword [rax]

loc_20fbf0:
  vmovss       xmm0, dword [rsi+r9*4]             ; CODE XREF=rx_demo
                                                  ;  _partition1_compute
                                                  ;  +316
  movzx        eax, al
  vmovss       xmm7, dword [rdx+r9*4]
  vmovss       dword [rdi+rax*4+4], xmm0
  vmovss       dword [rdi+rax*4-0xfc], xmm0
  movzx        eax, byte [rdi]
  vmovss       dword [rdi+rax*4+0x204], xmm7
  vmovss       dword [rdi+rax*4+0x104], xmm7
  vmulps       ymm0, ymm5, rdi+rax*4+0x144
  vfmadd231ps  ymm0, ymm4, rdi+rax*4+0x164
  vfmadd231ps  ymm0, ymm3, rdi+rax*4+0x184
  vfmadd231ps  ymm0, ymm10, rdi+rax*4+0x1a4
  vfmadd231ps  ymm0, ymm9, rdi+rax*4+0x1c4
  vfmadd231ps  ymm0, ymm8, rdi+rax*4+0x1e4
  vextractf128 xmm1, ymm0, 0x1
  vaddps       xmm0, xmm0, xmm1
  vpermilpd    xmm1, xmm0, 0x1
  vaddps       xmm0, xmm0, xmm1
  vmovshdup    xmm1, xmm0
  vaddss       xmm0, xmm0, xmm1
  vmulps       ymm1, ymm5, rdi+rax*4-0xbc
  vfmadd231ps  ymm1, ymm4, rdi+rax*4-0x9c
  vfmadd231ss  xmm0, xmm6, xmm7
  vfmadd231ps  ymm1, ymm3, rdi+rax*4-0x7c
  vfmadd231ps  ymm1, ymm10, rdi+rax*4-0x5c
  vfmadd231ps  ymm1, ymm9, rdi+rax*4-0x3c
  vfmadd231ps  ymm1, ymm8, rdi+rax*4-0x1c
  vextractf128 xmm2, ymm1, 0x1
  vaddps       xmm1, xmm1, xmm2
  vpermilpd    xmm2, xmm1, 0x1
  vaddps       xmm1, xmm1, xmm2
  vmovshdup    xmm2, xmm1
  vaddss       xmm1, xmm1, xmm2
  vfmadd231ss  xmm1, xmm6, dword [rdi+rax*4+4]
    ; Compute Next Circular Buffer Head Position + Handle Wraparound
  inc          eax
  and          al, 0x3f
  or           al, 0x40
  mov          byte [rdi], al
  vmovss       dword [rcx+r9*4], xmm1
  vmovss       dword [r8+r9*4], xmm0
  inc          r9
  cmp           r9, 0x78
  jne           loc_20fbf0

  vzeroupper
  ret
    ; endp
```

Listing D.3: Disassembly of RRC with No Sub-Blocking (Listing D.1) by Hopper Disassembler

D.2.2 RRC, Sub-Blocked

```
; ================ BEGINNING OF PROCEDURE ================

; Variables:
;    var_10: int8_t, -16
;    var_20: -32
;    var_30: -48
;    var_40: -64
;    var_50: -80
;    var_60: -96
;    var_70: -112
;    var_80: -128
;    var_90: -144
;    var_A0: -160
;    var_B0: -176
;    var_C0: -192
;    var_D0: -208

rx_demo_partition1_compute:
  push          rbp                                        ; Begin of unwind
                                                           ; block (FDE at
                                                           ; 0x2116e4), CODE
                                                           ; XREF=rx_demo_
                                                           ; partition1_thread+
                                                           ; 1304
  mov           rbp, rsp
  push          r14
  push          rbx
  and           rsp, 0xffffffffffffffe0
  sub           rsp, 0xc0
  xor           r10d, r10d
  vxorps        xmm0, xmm0, xmm0
  mov           r9d, 0x48
  nop

loc_2134f0:
  lea         rbx, qword [r10*8]                           ; CODE XREF=rx_demo_
                                                           ; partition1_compute
```

```
                                                    ; +920
movzx      eax, byte [rdi]
  ; Each segment is copying 3*4=12 elements.  With 4 instances,
  ; a total of 48 elements are copied. With the input being complex,
  ; this corresponds to 24 samples being copied, equivalent to a
  ; full sub-block.
lea          r14, qword [rbx+rbx*2]
vmovups      ymm1, rsi+r14*4
vmovups      ymm2, rsi+r14*4+0x20
vmovups      ymm3, rsi+r14*4+0x40
vmovups      rdi+rax*4+0x44, ymm3
vmovups      rdi+rax*4+0x24, ymm2
vmovups      rdi+rax*4+4, ymm1
vmovups      ymm1, rsi+r14*4
vmovups      ymm2, rsi+r14*4+0x20
vmovups      ymm3, rsi+r14*4+0x40
vmovups      rdi+rax*4-0xfc, ymm2
vmovups      rdi+rax*4-0x11c, ymm1
vmovups      rdi+rax*4-0xdc, ymm3
mov          rax, 0xffffffffffffff3c
movzx        r11d, byte [rdi]
vmovups      ymm1, rdx+r14*4
vmovups      ymm2, rdx+r14*4+0x20
vmovups      ymm3, rdx+r14*4+0x40
vmovups      rdi+r11*4+0x244, ymm1
vmovups      rdi+r11*4+0x264, ymm2
vmovups      rdi+r11*4+0x284, ymm3
lea          rbx, qword [rdi+r11*4]
vmovups      ymm1, rdx+r14*4
vmovups      ymm2, rdx+r14*4+0x20
vmovups      ymm3, rdx+r14*4+0x40
vmovups      rdi+r11*4+0x124, ymm1
vmovups      rdi+r11*4+0x144, ymm2
  ; This appears to be setting the accumulators to 0 (in memory).
  ; Technically not necessary as the accumulator values are reset
  ; in each loop iteration and can be held in vector registers.
vmovups      rdi+r11*4+0x164, ymm3
vmovaps      rsp+0xd0+var_30, ymm0
vmovaps      rsp+0xd0+var_50, ymm0
vmovaps      rsp+0xd0+var_70, ymm0
vmovaps      rsp+0xd0+var_D0, ymm0
vmovaps      rsp+0xd0+var_B0, ymm0
vmovaps      rsp+0xd0+var_90, ymm0
vmovaps      ymm6, rsp+0xd0+var_70
vmovaps      ymm5, rsp+0xd0+var_D0
vmovaps      ymm4, rsp+0xd0+var_50
vmovaps      ymm3, rsp+0xd0+var_B0
vmovaps      ymm2, rsp+0xd0+var_30
vmovaps      ymm1, rsp+0xd0+var_90
nop          dword [rax]
```

```

    ; Same coefficient used across all dot products.  A single scalar
    ; was broadcast across the vector register.  rax initialized to
    ; -196 and loop increments tax by 4, giving loop 49 iterations
    ; (length of the RRC filter).  6 vector accumulator registers are
    ; used which corresponds to 48 individual accumulators.  For a
    ; complex*real dot product, this corresponds to 24 dot products,
    ; the sub-blocking Length.
loc_213620:
  vbroadcastss ymm7, dword [rax+0x2012c4]          ; CODE XREF=rx_demo_
                                                    ; partition1_compute
                              ; +400
  vfmadd231ps  ymm6, ymm7, rbx+rax+8
  vfmadd231ps  ymm5, ymm7, rbx+rax+0x248
  vfmadd231ps  ymm4, ymm7, rbx+rax+0x28
  vfmadd231ps  ymm3, ymm7, rbx+rax+0x268
  vfmadd231ps  ymm2, ymm7, rbx+rax+0x48
  vfmadd231ps  ymm1, ymm7, rbx+rax+0x288
  add          rax, 0x4                             ; Increment by 4
  jne          loc_213620

  lea        eax, dword [r11+0x18]
    ; Looks like copying InnerProduct results to memory in preparation
    ; for extracting and coping to the output.
  cmp          r11b, 0x77
  vmovaps      rsp+0xd0+var_70, ymm6
  vmovaps      rsp+0xd0+var_D0, ymm5
  vmovaps      rsp+0xd0+var_50, ymm4
  vmovaps      rsp+0xd0+var_B0, ymm3
  vmovaps      rsp+0xd0+var_30, ymm2
  vmovaps      rsp+0xd0+var_90, ymm1
  movzx        eax, al
  cmova        eax, r9d
  inc          r10
    ; Looks like copying results into the output arrays.  Compiler
    ; appears to have unrolled the copy loop but could have broken the
    ;  loop in half and produced contiguous writes.  Fortunately, the
    ;  AMD processor does support write combining in the write buffer
    ; which can help amortize costs.  Future work to support explicit
    ; memcpy calls for FIFO logic.
  mov          byte [rdi], al
  vmovaps      xmm1, xmmword [rsp+0xd0+var_70]
  vmovss       dword [rcx+r14*4], xmm1
  vmovaps      xmm2, xmmword [rsp+0xd0+var_D0]
  vmovss       dword [r8+r14*4], xmm2
  vextractps   dword [rcx+r14*4+4], xmm1, 0x1
  vextractps   dword [r8+r14*4+4], xmm2, 0x1
  vextractps   dword [rcx+r14*4+8], xmm1, 0x2
  vextractps   dword [r8+r14*4+8], xmm2, 0x2
  vextractps   dword [rcx+r14*4+0xc], xmm1, 0x3
  vextractps   dword [r8+r14*4+0xc], xmm2, 0x3
  vmovaps      xmm1, xmmword [rsp+0xd0+var_60]
  vmovss       dword [rcx+r14*4+0x10], xmm1
  vmovaps      xmm2, xmmword [rsp+0xd0+var_C0]
  vmovss       dword [r8+r14*4+0x10], xmm2
  vextractps   dword [rcx+r14*4+0x14], xmm1, 0x1
  vextractps   dword [r8+r14*4+0x14], xmm2, 0x1
  vextractps   dword [rcx+r14*4+0x18], xmm1, 0x2
  vextractps   dword [r8+r14*4+0x18], xmm2, 0x2
  vextractps   dword [rcx+r14*4+0x1c], xmm1, 0x3
  vextractps   dword [r8+r14*4+0x1c], xmm2, 0x3
  vmovaps      xmm1, xmmword [rsp+0xd0+var_50]
  vmovss       dword [rcx+r14*4+0x20], xmm1
  vmovaps      xmm2, xmmword [rsp+0xd0+var_B0]
  vmovss       dword [r8+r14*4+0x20], xmm2
  vextractps   dword [rcx+r14*4+0x24], xmm1, 0x1
  vextractps   dword [r8+r14*4+0x24], xmm2, 0x1
  vextractps   dword [rcx+r14*4+0x28], xmm1, 0x2
  vextractps   dword [r8+r14*4+0x28], xmm2, 0x2
  vextractps   dword [rcx+r14*4+0x2c], xmm1, 0x3
  vextractps   dword [r8+r14*4+0x2c], xmm2, 0x3
  vmovaps      xmm1, xmmword [rsp+0xd0+var_40]
  vmovss       dword [rcx+r14*4+0x30], xmm1
  vmovaps      xmm2, xmmword [rsp+0xd0+var_A0]
  vmovss       dword [r8+r14*4+0x30], xmm2
  vextractps   dword [rcx+r14*4+0x34], xmm1, 0x1
  vextractps   dword [r8+r14*4+0x34], xmm2, 0x1
  vextractps   dword [rcx+r14*4+0x38], xmm1, 0x2
  vextractps   dword [r8+r14*4+0x38], xmm2, 0x2
  vextractps   dword [rcx+r14*4+0x3c], xmm1, 0x3
  vextractps   dword [r8+r14*4+0x3c], xmm2, 0x3
  vmovaps      xmm1, xmmword [rsp+0xd0+var_30]
  vmovss       dword [rcx+r14*4+0x40], xmm1
  vmovaps      xmm2, xmmword [rsp+0xd0+var_90]
  vmovss       dword [r8+r14*4+0x40], xmm2
  vextractps   dword [rcx+r14*4+0x44], xmm1, 0x1
  vextractps   dword [r8+r14*4+0x44], xmm2, 0x1
  vextractps   dword [rcx+r14*4+0x48], xmm1, 0x2
  vextractps   dword [r8+r14*4+0x48], xmm2, 0x2
  vextractps   dword [rcx+r14*4+0x4c], xmm1, 0x3
  vextractps   dword [r8+r14*4+0x4c], xmm2, 0x3
  vmovaps      xmm1, xmmword [rsp+0xd0+var_20]
  vmovss       dword [rcx+r14*4+0x50], xmm1
  vmovaps      xmm2, xmmword [rsp+0xd0+var_80]
  vmovss       dword [r8+r14*4+0x50], xmm2
  vextractps   dword [rcx+r14*4+0x54], xmm1, 0x1
  vextractps   dword [r8+r14*4+0x54], xmm2, 0x1
  vextractps   dword [rcx+r14*4+0x58], xmm1, 0x2
  vextractps   dword [r8+r14*4+0x58], xmm2, 0x2
  vextractps   dword [rcx+r14*4+0x5c], xmm1, 0x3
  vextractps    dword [r8+r14*4+0x5c], xmm2, 0x3
  cmp           r10, 0x5                              ; Repeated 5 times
  jne           loc_2134f0

  lea           rsp, qword [rbp+var_10]
  pop           rbx
  pop           r14
  pop           rbp
  vzeroupper
  ret
    ; endp
```

Listing D.4: Disassembly of RRC with Sub-Blocking (Listing D.2) by Hopper Disassembler