



AFRL-RY-WP-TR-2023-0140

BLACKPARROT, AN OPEN SOURCE RISC-V MULTICORE, AND ITS ECOSYSTEM

Michael Taylor
University of Washington

SEPTEMBER 2023
Final Report

DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.

See additional restrictions described on inside pages

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with The Under Secretary of Defense memorandum dated 24 May 2010 and AFRL/DSO policy clarification email dated 13 January 2020. This report is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC)
(<http://www.dtic.mil>).

AFRL-RY-WP-TR-2023-0140 HAS BEEN REVIEWED AND IS APPROVED FOR
PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//Signature//

CHRISTOPHER A. BOZADA
Program Manager
Aerospace Components and Subsystems Division

//Signature//

GENE M. WILKINS, Lt Col, USAF
Deputy Chief
Aerospace Components & Subsystems Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show “//Signature//” stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE September 2023		2. REPORT TYPE Final		3. DATES COVERED	
				START DATE 5 June 2018	END DATE 6 June 2022
4. TITLE AND SUBTITLE BLACKPARROT, AN OPEN SOURCE RISC-V MULTICORE, AND ITS ECOSYSTEM					
5a. CONTRACT NUMBER FA8650-18-2-7856		5b. GRANT NUMBER N/A		5c. PROGRAM ELEMENT NUMBER 62716E	
5d. PROJECT NUMBER N/A		5e. TASK NUMBER N/A		5f. WORK UNIT NUMBER Y1TL	
6. AUTHOR(S) Michael Taylor					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Washington Office of Sponsored Programs 4333 Brooklyn AVE NE Seattle, WA 98195				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory, Sensors Directorate Wright-Patterson Air Force Base, OH 45433-7320 Air Force Materiel Command, United States Air Forces		Defense Advanced Research Projects Agency (DARPA/MTO) 675 North Randolph Street Arlington, VA 22203		10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RYPD	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S) AFRL-RY-WP-TR-2023-0140	
12. DISTRIBUTION/AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.					
13. SUPPLEMENTARY NOTES This material is based on research sponsored by the Air Force Research Laboratory (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7856. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory (AFRL), the Defense Advanced Research Projects Agency (DARPA), or the U.S. Government. Report contains color.					
14. ABSTRACT This report introduces BlackParrot, which aims to be the default open-source Linux-capable, cache-coherent, 64-bit RISC-V multicore used by the world. It addresses not only the technical details of BlackParrot, but also the hardware and software ecosystem that created for using it. BlackParrot is implemented using SystemVerilog and is available at https:// github.com/black-parrot .					
15. SUBJECT TERMS Reduced instruction set computer, multicore, cache-coherent, Linux-capable					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT		18. NUMBER OF PAGES
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified	SAR		135
19a. NAME OF RESPONSIBLE PERSON Christopher Bozada				19b. PHONE NUMBER (Include area code) N/A	

Table of Contents

Section	Page
List of Figures	vi
List of Tables	viii
1 INTRODUCTION	2
1.1 BlackParrot Project Goals	2
1.2 Success Metrics	2
1.3 Design Manifesto	5
1.4 BlackParrot Architecture	5
1.5 Race-Free Programmable Cache Coherence	5
1.6 Heterogeneously Tiled Multicore SoC Micro-architecture	6
1.7 BlackParrot Microarchitectural Tile Types	7
1.7.1 BlackParrot Core Tile	7
1.7.2 L2 Extension Tile	7
1.7.3 Coherent Accelerator Tile	8
1.7.4 Streaming Accelerator Tile	8
1.8 Other Tile Components	8
2 SUMMARY	9
2.1 Summary of Progress	9
2.2 Achievements	9
2.3 Detailed Core Microarchitecture	10
2.4 Efficiency Through Thin Interfaces	10
2.5 Front End	10
2.6 Back End	11
2.7 Memory End	11
2.8 Agile Development Process	12
2.9 Leveraging Open-Source Libraries	12
2.10 Evaluating Design Complexity and Out-of-Box Experience	12
2.11 Continuous Integration, Code Review	12
2.12 Co-simulation Testing Framework	13
2.13 ASIC Validation	14
3 RESULTS	17
3.1 BedRock Coherence System	17
3.1.1 BedRock	17
3.1.2 Request Processing	18
3.1.3 Protocol Tables	19
3.1.3.1 Cache Controller Protocol Table	19
3.1.3.2 Coherence Directory Protocol Table	19
3.1.4 Ordering Transactions - Way Groups	20
3.1.5 BP-BedRock Directory	20
3.1.6 Coherence Directory Segment	21
3.1.7 Directory Operations and Access	22
3.1.8 Tag Checker and LRU Extraction	22
3.1.9 BP-BedRock FSM CCE	22

Section	Page
3.1.10 Request FSM	23
3.1.10.1 MSHR.....	23
3.1.10.2 GAD	24
3.1.10.3 Request FSM.....	24
3.1.10.4 Atomics	25
3.1.11 Pending Bits	26
3.1.12 Speculative Bits	26
3.1.13 Memory Response FSM	26
3.1.14 BP-BedRock ucode CCE	26
3.1.15 Microcode ISA.....	27
3.1.15.1 Flag Instructions.....	28
3.1.15.2 Directory Instructions.....	29
3.1.15.3 Queue Instructions.....	30
3.1.15.4 Programming the CCE	30
3.1.16 Instruction Fetch and Predecode.....	30
3.1.17 Instruction Decode and Stall.....	30
3.1.18 Source Select and Arbitration.....	30
3.1.19 Register File.....	31
3.1.20 Functional Units.....	31
3.1.20.1 ALU and Branch	31
3.1.20.2 Message.....	32
3.1.21 Request Processing	32
3.1.22 Protocol Verification.....	33
3.1.23 BedRock Directory Overhead.....	34
3.1.24 CCE Request Processing Occupancy	35
3.1.24.1 FSM CCE Occupancy	35
3.1.24.2 ucode CCE Occupancy.....	36
3.1.24.3 Discussion	36
3.1.25 Area, Utilization, and Timing.....	37
3.1.26 Splash-3 Performance	38
3.1.27 Summary of BedRock Progress.....	38
3.2 FPGA Integration.....	39
3.2.1 Introduction.....	39
3.2.2 ZynqParrot	40
3.2.2.1 Choosing the FPGA.....	40
3.2.2.2 Choosing the Connections to use BlackParrot's External Interfaces. BlackParrot Exposes Three Interfaces to External Logic:.....	41
3.2.2.3 Version 1	44
3.2.2.4 Version 2.....	45
3.2.2.4.1 AXI4-Lite Decoder.....	46
3.2.2.4.2 FIFO with AXI4-Lite interfaces on both ends.	46
3.2.2.4.3 BlackParrot Address Space.	47
3.2.2.4.4 BlackParrot UART Read Issue.....	48

Section	Page
3.2.2.5 Version 3.....	48
3.2.2.6 Software Setup.....	49
3.2.2.6.1 Directory Structure.....	50
3.2.2.6.2 Header Files.....	50
3.2.2.6.3 Testbench.....	51
3.2.2.7 Validation and Results.....	52
3.3 OpenSource Integration.....	54
3.3.1 Introduction.....	54
3.3.2 Background.....	55
3.3.2.1 OpenPiton OpenPiton.....	55
3.3.2.2 Bring Your Own Core (BYOC).....	56
3.3.3 TinyParrot.....	58
3.3.3.1 Motivation for the BlackParrot Unicorn.....	59
3.3.3.2 Iterative Filling/Eviction.....	60
3.3.3.3 Sub-bank Filling/Eviction.....	61
3.3.3.4 ASIC Synthesis.....	63
3.3.3.4.1 Challenges.....	63
3.3.3.4.2 Synthesis Results.....	64
3.3.3.5 FPGA Synthesis.....	65
3.3.4 ParrotPiton.....	66
3.3.4.1.1 Way mapping the BlackParrot D\$ with the BPC.....	66
3.3.5 Requests.....	67
3.3.6 Remote Invalidations.....	67
3.3.6.1.1 L2 Atomics.....	68
3.3.6.2 P-Mesh Cache Engine (PCE).....	68
3.3.6.3 Full System Integration.....	71
3.3.6.4 Baremetal Testing.....	71
3.3.6.4.1 Simulation.....	71
3.3.6.4.2 FPGA.....	72
3.3.6.5 Linux Capability.....	73
3.3.7 HammerParrot.....	74
3.3.7.1 HammerBlade System Architecture.....	76
3.3.7.1.1 Compute Tile.....	76
3.3.7.1.2 Memory Tile.....	76
3.3.7.1.3 Accelerator Tile.....	76
3.3.7.1.4 Networking and Addressing.....	77
3.3.7.2 HammerParrot Hardware.....	77
3.3.7.2.1 Bridge FIFO Interface.....	77
3.3.7.2.2 Bridge MMIO Interface.....	77
3.3.7.2.3 System-level Address Maps.....	78
3.3.7.2.4 Top-level Integration.....	80
3.3.7.2.5 HammerBlade-x86 Simulation Infrastructure.....	80
3.3.7.2.6 Porting CUDA-lite to RISC-V.....	81

Section	Page
3.3.7.2.7 Software Validation using Dromajo HammerBlade.....	81
3.3.7.2.8 Dromajo HammerBlade Simulation Infrastructure	82
3.4 Application.....	82
3.4.1.1 Introductihon	83
3.4.1.2 Goals	84
3.4.1.3 Hardware Selection Ethernet MAC Selection.	85
3.4.1.4 FPGA Board and PHY Module Selection.	86
3.4.1.5 BlackParrot Ethernet Controller	86
3.4.1.5.1 ethernet_memory_map	87
3.4.1.5.2 ethernet_sender and ethernet_receiver	88
3.4.1.5.3 interrupt_control_unit.....	88
3.4.1.5.4 eth_mac_1g_rgmii_fifo	88
3.4.1.5.5 Clock Domains	90
3.4.1.5.6 Meeting the Timing Constraints.....	91
3.4.2 RISC-V PLIC.....	91
3.4.2.1 RISC-V PLIC from OpenTitan.....	92
3.4.2.2 Linux Liteeth Driver Memory Map.....	93
3.4.2.2.1 The TX/RX Buffer Region	93
3.4.2.2.2 Memory-Mapped Registers	93
3.4.2.3 Linux Interrupt Controller Driver.....	94
3.4.2.4 SiFive Platform-Level Interrupt Controller (PLIC).....	95
3.4.3 Why not connecting Liteeth Driver to HLIC.....	96
3.4.4 Percpu Interrupt Handler	97
3.4.4.1 The RX/TX Path in Linux The RX Path	97
3.4.5 The TX Path.....	98
3.4.6 lwIP.....	99
3.4.6.1 Introduction	99
3.4.6.2 Components	100
3.4.6.3 Flow	100
3.4.6.4 Dynamic Memory Allocation	102
3.4.6.5 Porting	103
3.4.6.5.1 Implementing sys_now.....	103
3.4.6.5.2 Creating arch/cc.h.....	103
3.4.6.5.3 Creating lwip/lwipopts.h	103
3.4.6.5.4 Implementing Ethernet device drivers.....	103
3.4.6.6 Introduction	104
3.4.6.7 Porting	105
3.4.6.8 Boot Loaders for BlackParrot System on Module.....	106
3.4.7 The SSL Loader	108
3.4.8 Validation and Results	108
3.4.8.1 FPGA Synthesis Reports Zedboard Utilization.....	109
3.4.9 Quality of the RX RGMII Signals.....	110
3.4.10 Linux Kernel 5.15 (with Liteeth driver and the RISC-V PLIC driver)	110

Section	Page
3.4.10.1.1 TCP/SSL Examples	110
3.4.10.1.2 The BlackParrot System on Module.....	110
3.4.10.2 Software Footprint lwIP Library Footprint	110
3.4.10.3 Network Throughput	111
3.4.10.4 lwIP UDP Throughput	111
3.4.10.4.1 lwIP TCP Throughput	112
3.4.10.4.2 wolfSSL Throughput	112
3.4.10.4.3 Linux Throughput.....	112
3.4.10.4.4 Linux NFS Throughput	112
4 DISCUSSION	115
4.1 Challenges.....	115
4.2 Community Impact	115
5 CONCLUSION	116
6 REFERENCES	117
LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS.....	124

List of Figures

Figure	Page
Figure 1: The BlackParrot Success Metrics and Manifesto	3
Figure 2: Recent Energy/Performance Optimized 64-bit RISC-V Open Source Linuxcapable ASIC Application Class Processor cores	4
Figure 3: BlackParrot multicore SoCs are comprised of a mesh of heterogeneous tiles, allowing flexible composition of cores, accelerators, L2 cache slices, I/O and DRAM controllers	7
Figure 4: BlackParrot Core Microarchitecture.....	10
Figure 5: Microprogrammed Cache Coherence Engine.....	11
Figure 6: Hybrid Co-simulation Using Dromajo	13
Figure 7: Shmoo Plot showing BlackParrot 14nm Silicon Operation Across Voltage and Frequency Conditions.....	14
Figure 8: BlackParrot GF12 Placed and Routed	15
Figure 9: Way Group Organization	19
Figure 10: BP-BedRock Coherence Directory Architecture.....	20
Figure 11: BP-BedRock Coherence Directory Segment.....	21
Figure 12: BP-BedRock FSM CCE Block Diagram.....	22
Figure 13: BP-BedRock FSM CCE Request State Machine	25
Figure 14: BP-BedRock Microcode-Programmable CCE Block Diagram.....	27
Figure 15: MOESIF Microcode Processing Flow - Initial and Fast Path	31
Figure 16: MOESIF Microcode Processing Flow - Full Path.....	33
Figure 17: 8-core BP-BedRock Splash-3 Normalized Execution Time	37
Figure 18: Zynq-7000 SoC Block Diagram [93]	42
Figure 19: Zynq-7000 System-Level Address Map [93]	43
Figure 20: Version 1 ZynqParrot Design	45
Figure 21: Version 2 ZynqParrot Design	46
Figure 22: Version 3 ZynqParrot Design	49
Figure 23: AXI R/W to Write Module and FSM	50
Figure 24: Future Extension to ZynqParrot	51
Figure 25: Testing Directory Setup.....	52
Figure 26: OpenPiton Architecture [39]	56
Figure 27: BYOC System Architecture [37].....	57
Figure 28: Options for Interfacing the Memory System with the Core [37].....	58
Figure 29: A Core Tile in the BlackParrot Multicore System	59
Figure 30: Area Breakdown of a Single BlackParrot Core Tile (no L2 slice).....	60
Figure 31: Iterative Filling of a Cacheline	62
Figure 32: Dual-ported Memories using Single-ported Memories	64
Figure 33: Area Breakdown of BlackParrot with Different cache Organizations in the FreePDK 45nm Process Node	65
Figure 34: ParrotPiton Logo	66
Figure 35: Way Mapping between D\$ and BPC.....	67
Figure 36: PCE FSM Functionality	69
Figure 37: ParrotPiton Tile	71
Figure 38: Three-core ParrotPiton Booting Linux.....	74
Figure 39: HammerBlade with off-chip x86 Host Core.....	75
Figure 40: Compute Tile Core Pipeline	76
Figure 41: HammerParrot Manycore Bridge	80
Figure 42: HammerBlade Simulation Infrastructure in VCS.....	81

Figure	Page
Figure 43: Dromajo+HammerBlade Simulation Infrastructure in VCS	81
Figure 44: The OSI Model.....	83
Figure 45: Ethernet Hardware Overview	84
Figure 46: Ethernet Frame	84
Figure 47: Comparison between Different MAC Modules.....	85
Figure 48: GMII Signals	87
Figure 49: RGMII Signals	88
Figure 50: The <code>eth_mac_1g_rgmii_fifo</code> Module.....	89
Figure 51: The <code>rgmii_phy_if</code> Module.....	89
Figure 52: The <code>ethernet_controller</code> Module	90
Figure 53: The <code>ethernet_controller_wrapper</code> Module.....	90
Figure 54: The Memory Map of the BlackParrot Ethernet Controller.....	91
Figure 55: Waveforms of <code>rgmii_phy_if</code>	92
Figure 56: The Clock Domains of the MAC Module	93
Figure 57: The PLIC Architecture [25].....	94
Figure 58: PLIC and RISC-V Harts [25]	95
Figure 59: The Liteeth Memory Map	96
Figure 60: A Device Tree Example for the RISC-V Interrupt Controllers.....	98
Figure 61: The Linux IRQ Domains for HLIC and PLIC.....	99
Figure 62: lwIP Overview.....	100
Figure 63: Mainloop	101
Figure 64: lwIP Components	101
Figure 65: lwIP Memory Pools.....	102
Figure 66: How an SSL Certificate Works	105
Figure 67: The Booting Process of a Linux Distro on the BlackParrot System on Module	107
Figure 68: The FSM within the TFTP Loader	107
Figure 69: ZynqParrot with Ethernet Controller and PLIC	109
Figure 70: lwIP Library Footprint.....	112
Figure 71: wolfSSL Library Footprint.....	113
Figure 72: The TFTP Loader Footprint	114

List of Tables

Table	Page
Table 1: BedRock Cache Controller Protocol Table - MOESIF. X State Indicates a Valid State Provided by Coherence Directory	16
Table 2: BedRock Coherence Directory Protocol Table - MOESIF. Superscript States are States Attached to Command Messages	18
Table 3: BP-BedRock MSHR Flags	23
Table 4: BP-BedRock ucode CCE Coherence ISA.....	29
Table 5: Coherence Directory Overhead Comparison	34
Table 6: BP-BedRock CCE Request Occupancy (MOESIF)	34
Table 7: BP-BedRock ucode CCE Resource Overheads	36
Table 8: FPGA Utilization for Two BlackParrot Configurations	41
Table 9: ZynqParrot Address Mapping.....	48
Table 10: FPGA Utilization for the ZynqParrot System with the Default BlackParrot Configuration	53
Table 11: ZynqParrot Hierarchical Utilization	53
Table 12: SPEC2000 Benchmarks Running on ZynqParrot	54
Table 13: SPEC2006 Benchmarks Running on ZynqParrot	54
Table 14: Bank and Fill Widths Supported.....	62
Table 15: FPGA Utilization Breakdown of BlackParrot with Different Cache Organizations	65
Table 16: RISC-V Atomic Instructions	68
Table 17: FPGA Utilization for different ParrotPiton Configurations.....	72
Table 18 shows the Hierarchical Utilization for Some of the Major Components of the ParrotPiton System	73
Table 19: HammerParrot Address Map	79
Table 20: HammerParrot Bridge FIFO Address Map.....	79
Table 21: ZynqParrot Utilization	108
Table 22: The BlackParrot Ethernet Utilization	109
Table 23: The RISC-V PLIC Module Utilization	109

Acknowledgement. Special thanks to Andreas Olofsson, Serge Leef and the intrepid team at DARPA, and also Christopher Bozada at AFRL. This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7856. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

1 INTRODUCTION

This report introduces BlackParrot, which aims to be the default open-source, Linux-capable, cachecoherent, 64-bit RISC-V multicore used by the world. The BlackParrot team aims to advance the world’s knowledge about the “software engineering of hardware”. Although originally bootstrapped by the University of Washington and Boston University via DARPA funding, BlackParrot strives to be community-driven and infrastructure agnostic; a multicore which is Pareto optimal in terms of power, performance, area and complexity. In order to ensure BlackParrot is easy to use, extend and most importantly trust, development is guided by three core principles: Be Tiny, Be Modular, and Be Friendly. Development efforts have prioritized the use of intentional interfaces and modularity and silicon validation as first order design metrics, so that users can quickly get started and trust that their design will perform as expected when deployed. BlackParrot has been validated in a 12nm FinFET tapeout. BlackParrot is ideal as a standalone Linux processor or as a malleable fabric for an agile accelerator SoC design flow.

1.1 BlackParrot Project Goals

RISC-V [85] is a disruptive technology. Never before has such a large, global community worked together to put forth a complete open source instruction set, machine model, and software stack. There is a strong belief in the community that RISC-V will find their foothold as low-NRE, high performance host cores to the agile-developed specialized accelerators that are being developed to in response to the end of Dennard Scaling.

Strangely lacking, however, is a similarly globally-maintained open source *implementation* of a RISC-V SoC. BlackParrot, described in this work, is designed to fill this gap. BlackParrot is open source and available now under the BSD license. BlackParrot is written in standard SystemVerilog and hence easily integrates into existing design methodologies and is easily understood and modified by industrial designers. BlackParrot has been fabricated in the GlobalFoundries 12nm process, with several iterations of refactoring to attain high area, delay and power efficiency.

Taking lessons from software engineering and the scalability of Linux development, BlackParrot has a modular design that puts the interfaces first. This approach will enable scalable collaboration by allowing contributors to work independently without having to completely understand all of BlackParrot’s components, or how they are evolving. BlackParrot seeks not only to advance the state-of-the-art in open source processor architecture, but also to develop approaches that change the way the world designs hardware.

1.2 Success Metrics

Adoption of BlackParrot will be driven by optimizing across four dimensions, as shown in Figure 1: quality, virality, functionality and efficiency.

Quality: From the beginning, BlackParrot was architected not simply to achieve RISC-V compliance, but to produce a quality codebase that engineers could inherently *trust* as a secure and high-quality design. At the heart of this approach (informally termed *the software engineering of hardware*) is the pervasive use of intentionally designed, narrow, modular

interfaces that make the design easy to reason about without sacrificing performance, power or area. Leveraging years of processor design experience, the team developed a high-level design document that partitioned the multicore into three major modules with easy-to-understand, light-weight transactional interfaces. Each module then had its own design document with specified its own internal modular interfaces, and worked through the important nuances and special cases. From there, schematics and then SystemVerilog RTL leveraged BaseJump Standard Template Library [78], an expansive set of intentionally designed interfaces for common computer architecture and hardware atoms that comes with corresponding silicon-verified SystemVerilog implementations. The RTL then underwent exten-

BlackParrot Success Metrics

The BlackParrot Manifesto

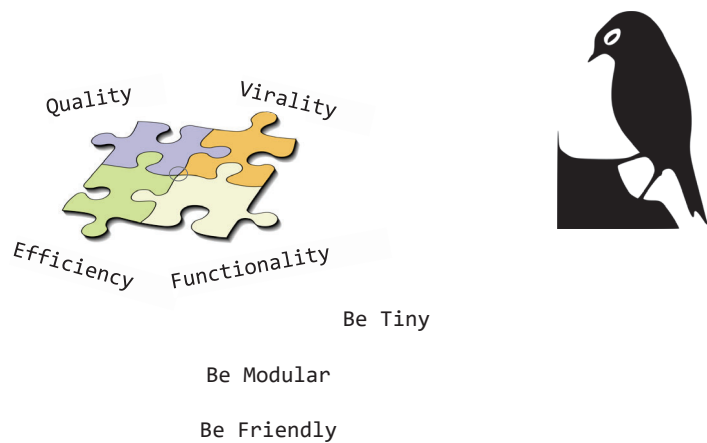


Figure 1: The BlackParrot Success Metrics and Manifesto

The Success Metrics strategically align the project for widespread adoption, and the BlackParrot Manifesto provides tactical guidance for technical decisions.

sive code review, and both unit and random testing. The goal is that an experienced engineer can evaluate the documentation, code, and tests; appreciate BlackParrot’s quality and use it confidently. **Virality:** While BlackParrot is a quality design, it will ultimately be unsuccessful if it is not widely adopted and if the community does not collectively take stewardship of it. For this reason, the project prioritizes the out-of-the-box experience, making it simple to get up and running via a github checkout, and pulling in as few external components as possible. BlackParrot employs a widely known language, SystemVerilog, instead of Chisel or BlueSpec. The team has a focus on friendliness and inclusiveness in our social interactions. Too many online collaboration forums are marred by a tolerance of abusive behavior particularly by respected members of the community, as highlighted by a recent case where Linus Torvalds himself stepped away from Linux for several months to try to contemplate the toxic effects of his curmudgeonly attitude. The team actively tries to prevent “not invented here” syndrome from

taking hold in the BlackParrot effort. The modular nature of the design makes it easy for individual contributors to get up to speed.

Functionality: BlackParrot boots Linux. It implements all of the core features of the RISC-V instruction set that are used by current software stacks. It also contains all the useful features required by modern SoCs, such as interrupt controllers, cache coherence and cache hierarchies, and easy-to-integrate accelerator interfaces.

Efficiency: BlackParrot must have best-of-class PPA for its target domain; the Linux-capable host multicore for accelerator chips. Early Coremark scores show BlackParrot achieves competitive performance with both academic and commercial cores of its class, as shown in Table 2. Extensive design and RTL code review are used to ensure BlackParrot contains efficient implementations of modern microarchitectural features expected of a Linux-class microprocessor. SRAM and logic structures are sized to be performance, power, and area (PPA) efficient. To validate the design, BlackParrot has been fabricated in GlobalFoundries 12nm process node, and is deploying new features through frequent follow-on 12nm, and 28nm tapeouts; as well as validating in Free45PDK, 40nm and open source Sky130 130nm tapeins.

	CoreMark /MHz	Language	Design Schema	Natively Multicore	License
BlackParrot	3.04	SystemVerilog	Pervasive Modular Interfaces; Standard Template Library	✓	BSD-3
Rocket	2.62	Chisel	Generator		BSD-3
Ariane	2.45	SystemVerilog	Monolithic		SHL-2
Shakti	2.20	BlueSpec	Generator		BSD-3
Flute	1.00	BlueSpec	Generator		Apache-2
SiFive U54MC	3.01	Chisel	Generator	✓	Closed Source

Figure 2: Recent Energy/Performance Optimized 64-bit RISC-V Open Source Linuxcapable ASIC Application Class Processor cores

Per-core performance is given in CoreMark / MHz as is standard in the industry. Also offered for comparison is the equivalent closed source, for-money, Linux-capable SiFive U54 core. (Note that since these open source projects are living, breathing projects, this is just a snapshot in time!)

1.3 Design Manifesto

During the course of the project, there were many cases where there are two reasonable technical directions to take the project. To guide the effort, the BlackParrot team developed an informal *manifesto* to help decide these difficult cases. The manifesto has three key rules:

1. *Be tiny.* When choosing among alternatives, choose the option that results in a smaller, more understandable code base and in less die area, simpler critical paths, and fewer bugs. The result is a code base that is as small and understandable as possible, and hardware that is PPA efficient. Take care to *not* implement esoteric and non-performance critical components in RTL, and to avoid a common problem in recent generator-based RTL methodologies: a multitude of tunable knobs in which most combinations have been untested and yield dubious PPA benefit. If a feature is required by the RISC-V spec but is not performance critical, implement it through emulation. The code is “all the RTL you need and nothing that you don’t”.
2. *Be modular.* Employ clean, latency-insensitive interfaces that do not rely on knowledge of the other module’s internals. This allows multiple contributors to work independently of each other, and to minimize bugs that emerge from incomplete understanding of the entire code base; and
3. *Be friendly.* Ask in both in design decisions and in presentation: does this make the project more approachable? Attempt to build a large open source project culture that encourages contributions and avoids the “not invented here” syndrome.

To facilitate widespread adoption, the BlackParrot infrastructure is built with community in mind. Environmental dependencies are kept to a minimum. BlackParrot is developed on a modern Linux version with easily portable Makefile-based simulation and synthesis flows. Support for opensource tools such as Verilator and GTKWave is prioritized as a first-class concern, and BlackParrot is one of the first major test cases for the OpenROAD open-source CAD flow. All of these considerations culminate in a focus on “out-of-box experience”. Prospective users are able to go from GitHub to simulation with a handful of simple commands.

1.4 BlackParrot Architecture

The BlackParrot multicore implements the RISC-V 64-bit “RV64G” architecture, which includes the base integer ISA “I”, multiplication and division “M”, atomics “A”, single and double precision floating-point “F/D”. It supports three privilege levels—machine, supervisor, and user—as well as SV39 virtual memory; these extensions are sufficient to efficiently run full-featured operating systems such as Linux.

1.5 Race-Free Programmable Cache Coherence

BlackParrot implements a distributed directory-based cache coherence protocol, which currently supports VI, MSI and MESI. The underlying implementation, *BedRock*, consists of a collection of *Local Cache Engines (LCE)*, each controlling an L1 cache, that connect over an

interconnection network to the programmable *Cache Coherence Engines (CCEs)*, which collectively maintain the address-sharded directory state.

The BedRock protocol implementations have the unique property of being *race-free*, because they ensure that coherence state transitions occur at the CCEs and not at the local caches or LCEs, which significantly reduces protocol complexity.

BlackParrot contains many novel and useful uncore/multi-core features, including a programmable cache coherence engine and coherent accelerator interfaces. This programmable coherence engine implements a novel distributed directory-based cache coherence protocol called BedRock. BedRock is fast and simple enough that verification tools can prove its correctness. BedRock enables effortless composition of systems with coherent cores, accelerators and I/O devices. BedRock is more completely described later in this report.

1.6 Heterogeneously Tiled Multicore SoC Micro-architecture

BlackParrot is designed as a scalable, heterogeneously tiled multicore micro-architecture, as shown in Figure 3. (The term multicore micro-architecture is used because the user or programmer is not aware of how the multicore's components are arranged, it is hidden beneath the multicore ISA layer.) Decomposing the system into replicated sub-blocks has several benefits: structures are regularized for scalability and ease of timing closure, systems can be flexibly composed into different topologies and protocol complexity can be shifted from the component level to the network level. Rather than connecting these tiles with a shared bus, BlackParrot uses a collection of NoCs. The routers used in BlackParrot are dimension-ordered and wormhole-switched, using credit-based flow-control to limit contention.

In order to promote regularity for hierarchical ASIC flow designs, the system is designed as a 2D mesh with a single router per network contained in each tile. Some networks may have multiple endpoints within a tile – these endpoint connections are combined and connected to the router through a wormhole concentrator. While many other processors use standard bus-based interfaces such as AXI or AHB for all on-chip communication, these protocols are highly complex and require sophisticated IP blocks to achieve reasonable performance. Rather than couple its internal networks with any particular implementation of a standard bus, BlackParrot provides a set of adapters to transduce between a set of well-known protocols, such as AXI or WishBone.

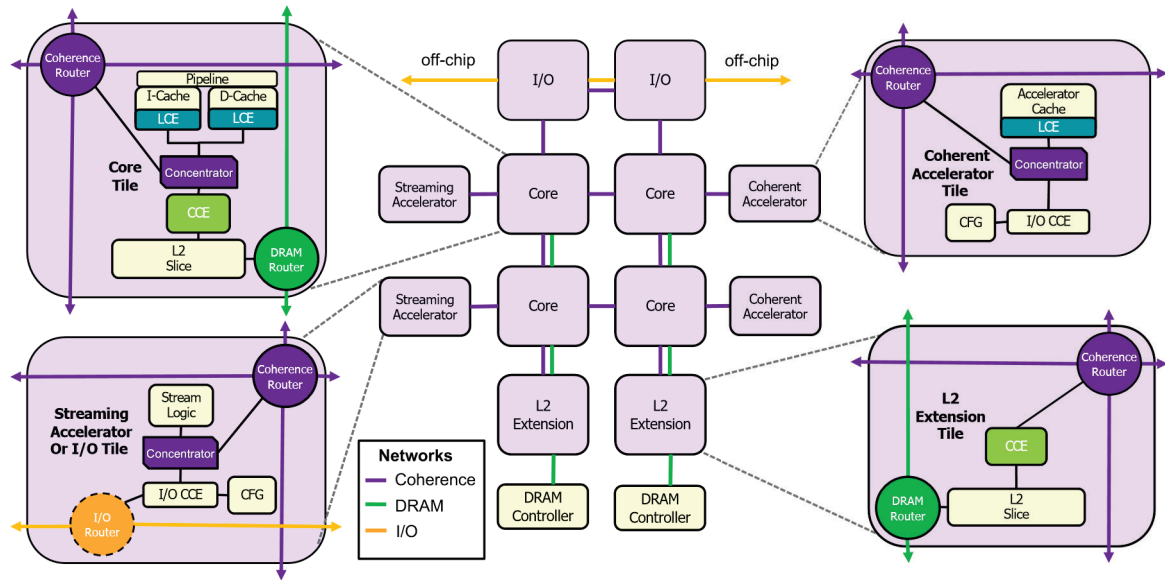


Figure 3: BlackParrot multicore SoCs are comprised of a mesh of heterogeneous tiles, allowing flexible composition of cores, accelerators, L2 cache slices, I/O and DRAM controllers

Four different kinds of tiles are pictured. *Core tiles* implement a processor, a directory shard and an L2 slice. *Coherent Accelerator tiles* implement an accelerator that has access to the cache coherent memory system. *L2 extension tiles* allow the amount of L2 cache to be changed. *Streaming accelerator or I/O tiles* allow flexible interfacing of a common memory system via a shared non L1-cached address space that is routed over the coherence network.

1.7 BlackParrot Microarchitectural Tile Types

BlackParrot microarchitectural tiles fall into one of the four categories enumerated in Figure 3, which is detailed below.

1.7.1 BlackParrot Core Tile

A BlackParrot Core Tile contains a full BlackParrot processor or an accelerator which acts a processor, comprising one or more coherent caches as well as a directory shard and an L2 slice. A typical system has many Core Tiles.

1.7.2 L2 Extension Tile

An L2 Extension Tile provides a simple scale-out method to increase the amount of on-chip L2 in a BlackParrot system. Each L2 extension contains a directory and a non-inclusive non-exclusive L2 slice. By distributing the L2 slices, a system designer can easily change the compute to cache ratio of a BlackParrot system without perturbing critical paths within the cores or the NoCs.

1.7.3 Coherent Accelerator Tile

Attaching a Coherent Accelerator Tile to the BlackParrot network can be done with a few degrees of specialization. From an abstract system view, a coherent accelerator is simply an LCE with a backing coherent cache. Depending on the accelerator's needs and the project's complexity budget, users may (in increasing order of complexity):

1. Attach the accelerator directly to the provided BlackParrot data cache.
2. Reuse the provided BlackParrot LCE and provide a specialized cache.
3. Provide a specialized LCE implementation that interfaces with directly with the network.

1.7.4 Streaming Accelerator Tile

Streaming Tiles are tiles which have no locally cached memory and do not logically control any physical memory. That is, these tiles do not contain a backing coherent cache for its LCE, nor a directory. These tiles may be used for basic I/O devices, network interface links, or even heavyweight streaming-based accelerators such as GPUs.

1.8 Other Tile Components

As a truly tiled micro-architecture, BlackParrot distributes as many system resources as possible. Each BlackParrot core tile contains a memory-mapped configuration block, a slice of the Core Level Interrupt Controller (CLINT) and a global L2 slice. Splitting these system resources promotes regularity in the tiles, removes globally routed configuration and interrupt signals, ultimately easing physical design implementation. With simple adjustments to the network memory map, components can easily be attached to the BlackParrot system and addressed from any other tile.

2 SUMMARY

2.1 Summary of Progress

RISC-V challenges the world order of x86 and ARM. UC Berkeley has bootstrapped a global stewardship to maintain the RISC-V ISA and its software base. However, yet to emerge is a similar global stewardship of a Linux capable RISC-V implementation that is documented, PPA efficient and implemented in standard SystemVerilog. BlackParrot is architected from the ground up to fill this role, in contrast to prior efforts [34] [43] [95] which have centralized stewardship models and evolved organically to their current state.

BlackParrot is tiny, modular and friendly. It is an ideal SoC “base class” to integrate with accelerators and build Linux-capable systems with. As BlackParrot becomes more widely used, community experts can contribute back to BlackParrot, ensuring that it remains representative of state of the art processor designs. BlackParrot is now available on GitHub (<https://github.com/black-parrot/>) under a BSD-3 license.

2.2 Achievements

The BlackParrot team has worked to provide an open-source industrial-strength RISC-V multicore along with accompanying software, ASIC and FPGA infrastructure. Specific achievements of the BlackParrot team include:

- Open-Source release of BlackParrot RTL for community use
- Chip Tape-outs using fully open-source RTL including 3 chips
 - GF12 4-core multicore BlackParrot Alpha
 - GF12 single BlackParrot as a host manycore controller
 - TSMC28 BlackParrots as standalone Linux host and manycore controller
- Design of first open-source race-free cache coherence protocol, BedRock
- A self-contained FPGA co-simulation and prototyping infrastructure
- Full integration with several community projects, including OpenPiton and LiteX
- Creation of fully open-source BlackParrot SoC with Ethernet

2.3 Detailed Core Microarchitecture

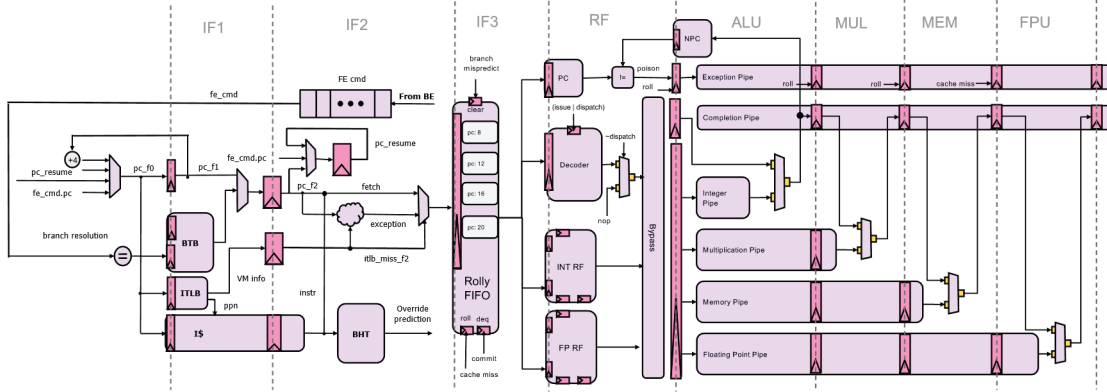


Figure 4: BlackParrot Core Microarchitecture

Both the Front End (IF1 and IF2 stages in the diagram) and Back End adhere to the interface specifications, with simple and efficient pipeline implementations. Because of a modest misprediction penalty, complex branch predictors are unnecessary. In order to remove a physical design intensive global stall signal, the Back End is non-stalling after the issue stage, instead flushing the pipeline and replaying instructions upon infrequent cache misses. Cache misses are handled entirely through the BedRock coherence system.

BlackParrot is designed to be modular, reaping the usual benefits of simpler verification, more agile development and easier onboarding of users and developers. Additionally, by focusing on interfaces rather than concrete implementations, BlackParrot is provisioned to support a wide variety of possible microarchitectures. Figure 3b shows the current BlackParrot core microarchitecture.

2.4 Efficiency Through Thin Interfaces

While software interface abstractions are a useful tool, hardware interfaces have physical overheads which can cripple a design. Interfaces in BlackParrot are designed to have minimal overhead, partitioning regions which are both logically and physically separated. Each interface described here is implemented as a parameterizable SystemVerilog struct passed through a latency insensitive port, usually via a small FIFO. Decoupling the Ends ensures there are no timing paths between these logically separated components and provides confidence that implementation changes in one End will not break another.

2.5 Front End

The Front End presents an in-order but potentially speculative instruction stream to the Back End. The issue queue decouples the Front End fetch from the Back End execution, allowing speculative fetching during long latency Back End operations such as services cache misses.

During instruction fetch, exceptions may arise. Since exceptions in this domain are purely speculative, they are sent to the Back End to be serviced in-line with instructions. Because the RISC-V virtual memory scheme may modify architectural state during instruction fetch (setting the “Access (A) bit”), all TLB misses in the Front End are sent to the Back End to be handled inline with other exceptions. Along with the PC/instruction/exception pair, the Front End also sends metadata associated with the branch prediction that resulted in that particular PC fetch.

2.6 Back End

The Back End executes instructions, handles exceptions, and generally maintains the architectural state of the processor. Messages from the Back End to the Front End are used to correct misspeculation and update shadow state in the Front End. Messages include branch resolution, interrupt redirection, iTLB manipulation and privilege mode changes. Upon branch resolution, the branch metadata associated with the branch is forwarded back to the Front End. This metadata is never inspected by the Back End; the particular branch prediction scheme used by the Front End is completely opaque to the Back End.

2.7 Memory End

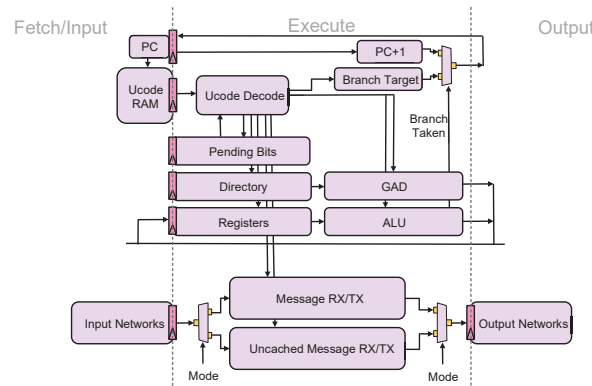


Figure 5: Microprogrammed Cache Coherence Engine

The microarchitecture of the CCE is fairly simple. By minimizing the coherence protocol complexity and pushing the management logic into software, the hardware is easily verifiable.

BlackParrot’s memory end, *BedRock*, is a scalable, distributed, directory-based coherence scheme designed with an emphasis on simplicity and verification. Nodes in the coherence system are Local Cache Engine (LCE) or Cache Coherence Engines (CCE). CCEs are responsible for managing coherence for a slice of physical address space. LCEs are responsible for initiating and responding to coherence requests on behalf of a coherence cache. *BedRock* connects all components of a BlackParrot multicore, including non-coherent or I/O devices.

Three types of CCE are available in *Bedrock*: a novel microprogrammed variant, a traditional fixed-function management engine, and a minimal controller which implements only uncached requests and is used for I/O or simple accelerators. The microprogrammed CCE is the default for a BlackParrot core and provides substantial flexibility and adaptability when implementing

variants of coherence protocols, and comes at a small area cost. Notably, since microcode can be changed out by a simple firmware update, advanced coherence experimentation and security patches can be applied even on existing silicon designs.

Some configurations of BlackParrot (e.g. single core, with software coherence, leveraging external coherence systems) do not require a full coherence system. In these cases, caches simply need a mechanism to bootstrap and service misses. By standardizing the cache miss handling interface and constraining it to be a subset of the Cache to LCE interface, different cache implementations and coherence schemes can be mixed and matched in BlackParrot.

2.8 Agile Development Process

This section describes the key features of the BlackParrot development effort that lead to be a design users can trust.

2.9 Leveraging Open-Source Libraries

In order to rapidly iterate on BlackParrot, it is important to leverage established open-source codebases. By building upon battle-tested hardware libraries such as BaseJump Standard Template Library (STL) for SystemVerilog [78] and Berkeley Verilog HardFloat [55], and integrating using latency insensitive design principles, BlackParrot is able to minimize its universe of possible bugs. Development of BlackParrot is done using commercial tools, such as Synopsys VCS, but supporting at least one open-source alternative, such as Verilator [72], is a first-class infrastructure concern.

2.10 Evaluating Design Complexity and Out-of-Box Experience

One of the most difficult-to-evaluate components of the BlackParrot manifesto is whether BlackParrot has achieved the goals of virality and complexity. To evaluate this, BlackParrot has been assigned as a 3-week class project in an VLSI class attended by 35 fourth and fifth year students, and all of the students (including ones that had not taken computer architecture before!) were successful in proposing and implementing unique, previously unsupported features into the core, such as modifying the pipeline to change the load-use latency to 2 cycles instead of 3, adding variable fill widths to the L1 cache, enabling different bank sizes in the L1 cache, and implementing more complex branch predictors in the frontend. University of Washington has yearly BlackParrot-related projects to continuously self-evaluate these aspects of the design.

2.11 Continuous Integration, Code Review

Cleanly specified interfaces between the front-end, back-end and memory system components allow for stubbed-out versions to quickly be constructed. Test harnesses at the component level were also developed.

The first instruction to flow through the BlackParrot core occurred four weeks after completion of the MAS documents and team formation. After this milestone BlackParrot remained fully functional throughout the development process. New features, such as virtual memory support, were added by forming new teams and adding them into the existing working BlackParrot core.

For some modules there were multiple alternatives implemented. For example, the programmable cache coherence controller was also implemented in a fixed-logic finite state machine form. This was done to fully understand the performance/area trade-off of the design.

Along the way, refinements to the initial module partitions did occur. For example, to remove a potential security flaw in the front-end of the processor, all page table walks are handled nonspeculatively in the back-end. This required refinements to the inter-module messages that occur within the processor. However, by having a clean interface between modules, no significant changes to the MAS documents and overall module structure were necessary to add this support, despite the fact that the early stages of the processor were not designed for it.

2.12 Co-simulation Testing Framework

The majority of testing in BlackParrot is done through a system-level testbench driven by program level execution. In addition to a handful of directed white-box tests, BlackParrot supports the riscv-

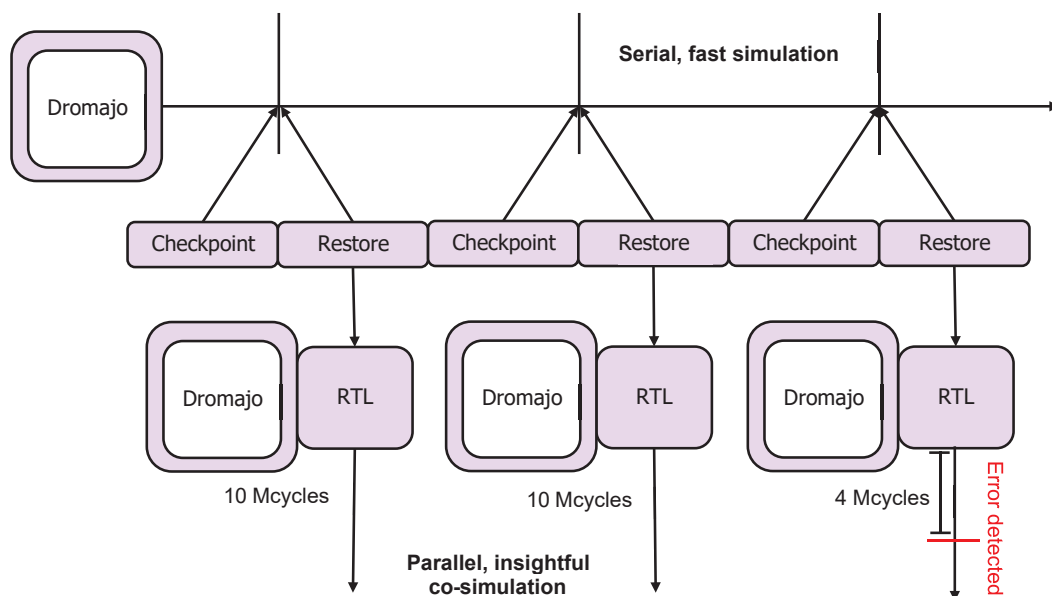


Figure 6: Hybrid Co-simulation Using Dromajo

Parallel co-simulation dramatically speeds up traditional RTL simulation methodologies without sacrificing flexibility tests suite, BEEBS suite, Spec and CoreMark as a baseline functional regression. Our plan is to greatly expand this selection.

RISCV-DV [53] is a UVM-based instruction generator framework recently released with support for RV64GC, MSU privilege modes, page-table generation and trap-handling support. Due to a dearth of open-source directed privileged mode tests (i.e. other than booting Linux), constrained

random testing is an effective way to expose edge cases. In a matter of days, RISC-V-DV was able to expose several hardware bugs in BlackParrot which may otherwise have been found millions of cycles into a Linux boot test.

Under traditional architectural research methodologies, there exist clear trade-offs between speed and accuracy. In order to temper excessive debug cycle times, hardware designers often analogously sacrifice system visibility for speed through techniques such as FPGA emulation [59] or high level modelling of sub-components. Debugging at the RTL level is commonly done by comparing execution traces of a simulation model with a “golden trace”, typically generated by a well-trusted functional model, and examining waveforms in the case of a trace mismatch. Another approach is to implement the RTL model in an FPGA, outputting periodic status and debug information. (A third option largely out of reach to academics and hobbyists is to use sophisticated FPGA emulation frameworks such as Synopsys ZeBu.)

Instead, BlackParrot employs a hybrid approach of parallel co-simulation using an open-source RISC-V ISA simulator, Dromajo [11]. First, a long-running program is simulated using Dromajo.

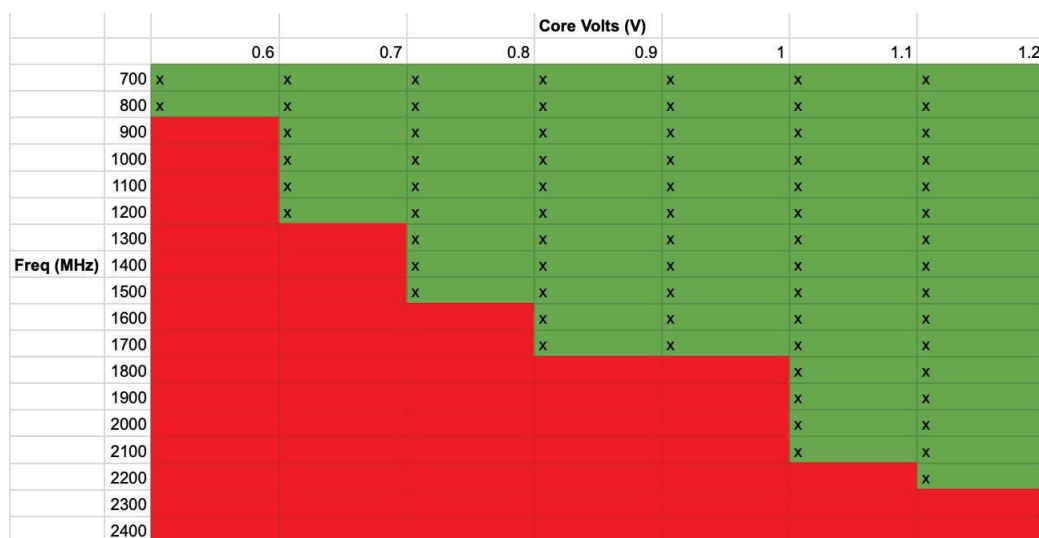


Figure 7: Shmoo Plot showing BlackParrot 14nm Silicon Operation Across Voltage and Frequency Conditions

Every N cycles, Dromajo collects the architectural state of the system and creates both a memory dump as well as a checkpoint ROM. The checkpoint ROM comprises ordinary RISC-V instructions designed to initialize a freshly rebooted processor to a well-defined architectural state. Next, *in parallel*, the BlackParrot RTL model is restored using each checkpoint ROM and co-simulated alongside the Dromajo model. Imprecise events such as interrupts and device I/O are relayed from RTL to Dromajo to keep the two versions aligned.

2.13 ASIC Validation

BlackParrot was taped out twice in GlobalFoundries 12nm and once in TSMC 28nm, as well as taped in multiple times in TSMC28, FreePDK 45nm and partially in Skywater 130nm. Alongside

these tapeouts, features and performance-power-area optimizations were identified and added to the core, continuously adding to its quality. The first BlackParrot tapeout was successfully brought up in a University of Washington lab while other tapeouts are in the bringup process. Providing the capability to rapidly switch between free and commercial flows enables architects to quickly validate their hardware experiments without the financial and intellectual overheads of maintaining a commercial CAD flow, or needing to sign restrictive foundry NDAs.

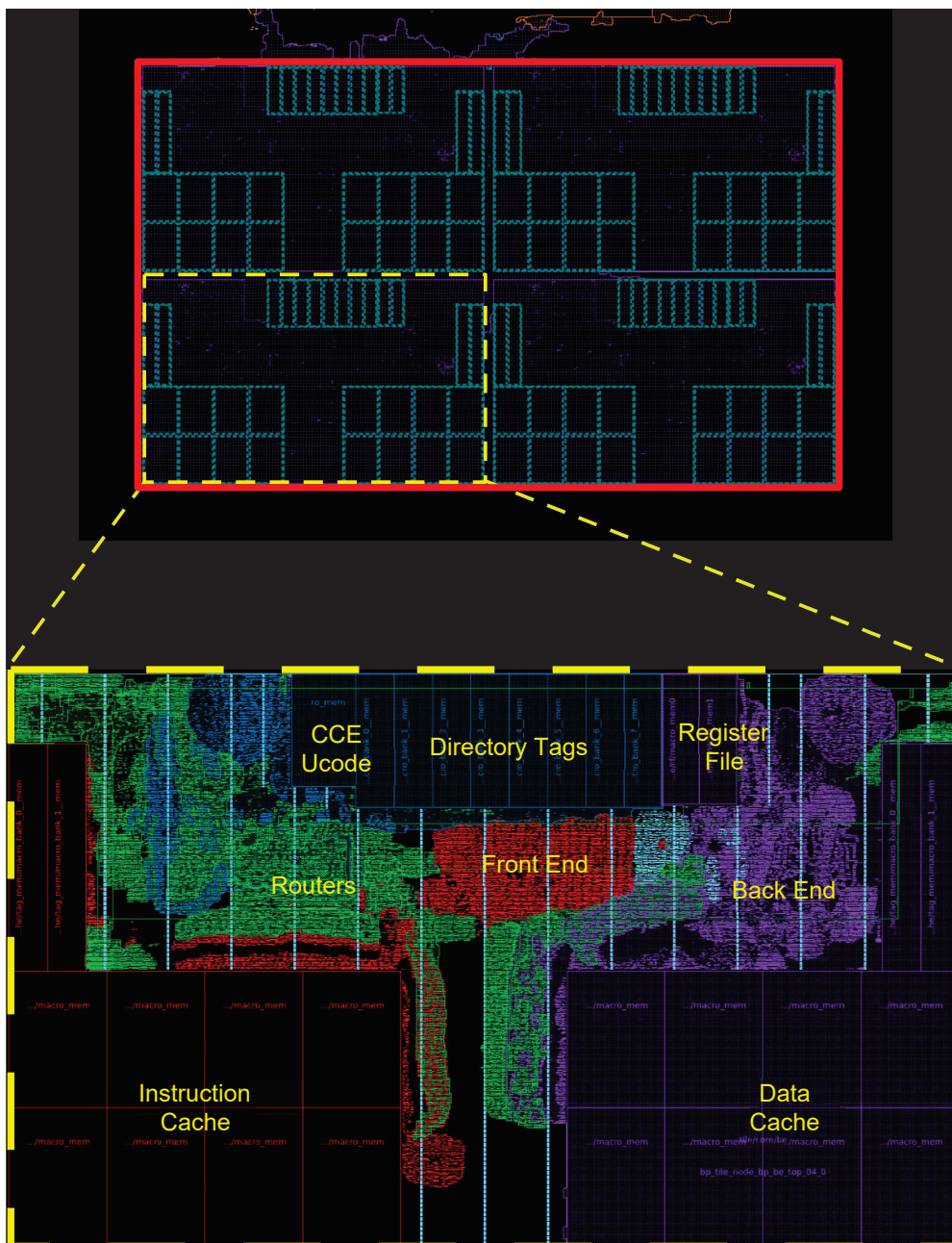


Figure 8: BlackParrot GF12 Placed and Routed

A quad-core BlackParrot system was taped out in July 2019 using GlobalFoundries 12nm. Lessons learned from this tapeout have driven several major changes in the BlackParrot system architecture, resulting in a 50% reduction in tile area as well as a 50% reduction in total wire length. Although RTL-level simulation is effective at analysing and comparing relative performance, an experiment without ASIC validation can mask serious physical limitations of a design.

Table 1: BedRock Cache Controller Protocol Table - MOESIF. X State Indicates a Valid State Provided by Coherence Directory

DATA messages are sent to another controller over the BedRock Fill network.

Cache Acti			Coherence Message							
Load State	Store		Inv	DATA	STW	WB	TR	ST-WB	ST-TR	ST-TR-WB
	ReqRd	ReqWr								
I	ReqRd	ReqWr		CohAck/X						
S	Hit	ReqWr	InvAck/I		CohAck/M					
E	Hit	Hit/M				NullWB/E		NullWB/X DATA/X	DATA, NullW-B/X	
M	Hit	Hit				DirtyWB/M		DirtyWB/X DATA/X	DATA, DirtyWB/X	
O	Hit	ReqWr			CohAck/M	DirtyWB/O	DATA/O	DirtyWB/X DATA/X		
F	Hit	ReqWr			CohAck/M		DATA/F		DATA/X	

3 RESULTS

3.1 BedRock Coherence System

This report section presents BP-BedRock, the open-source cache coherence protocol and system implemented within the BlackParrot 64-bit RISC-V multicore processor. BP-BedRock implements the BedRock directory-based MOESIF cache coherence protocol and includes two different opensource coherence protocol engines, one FSM-based and the other microcode programmable. Both coherence engines support coherent uncacheable access to cacheable memory and L1-based atomic read-modify-write operations.

Fitted within the BlackParrot multicore, BP-BedRock has been silicon validated in a GlobalFoundries 12nm FinFET process and FPGA validated with both coherence engines in 8-core configurations, booting Linux and running off the shelf benchmarks. After describing BP-BedRock and the design of the two coherence engines, performance is studied by analyzing processing occupancy and running the Splash-3 benchmarks on the 8-core FPGA implementations. Careful design and coherence-specific ISA extensions enable the programmable controller to achieve performance within 1% of the fixed-function FSM controller on average (2.3% worst-case) as demonstrated on our FPGA test system. Analysis shows that the programmable coherence engine increases die area by only 4% in an ASIC process and increases logic utilization by only 6.3% on FPGA with one additional block RAM added per core.

3.1.1 BedRock

The BedRock Cache Coherence Protocol [88] defines a family of directory-based invalidate cache coherence protocols using the common MOESIF cache coherence states and the coherence system components required to implement the protocol. This section provides a brief overview of the BedRock protocol, which the BP-BedRock system described in this paper implements. See the protocol specification [88] for additional details on the design decisions of the protocol itself.

A BedRock system comprises one or more cache controllers (Local Cache Engines), one or more coherence directories (Cache Coherence Engines), and the coherence message networks. Each cache controller manages a single cache participating in the coherence protocol. The coherence directory is Dir a standalone, inclusive, duplicate tag directory and acts as the point of serialization for all coherence transactions. Coherence is enforced using the *Single-Writer, Multiple-Reader (SWMR) Invariant* and *Data-Value Invariant* [60]. BedRock works with ordered or unordered networks and assumes the network implementation provides error free message delivery.

Table 2: BedRock Coherence Directory Protocol Table - MOESIF. Superscript States are States Attached to Command Messages

State	Coherence Request					Replacement
	ReqRd	ReqRd-NE	ReqWr from S	ReqWr from S	ReqWr from O/F	
I	DATA to Req/E	DATA to Req/S	DATA to Req/M			
S	DATA to Req/S	DATA to Req/S	Inv all S, DATA to Req/M	Inv other S, STW ^M to Req/M		
E	ST ^F -TR ^S -WB to Owner/F	ST ^F -TR ^S -WB to Owner/F	ST ^I -TR ^M to Owner/M			ST ^I -WB to Req/I
M	ST ^O -TR ^S to Owner/O	ST ^O -TR ^S to Owner/O	ST ^I -TR ^M to Owner/M			ST ^I -WB to Req/I
O	TR ^S to Owner/O	TR ^S to Owner/O	Inv all S, ST ^I -TR ^M to Owner/M	Inv other S and Owner, STW ^M to Req/M	Inv all S, STW ^M to Req/M	ST ^I -WB to Req/I
F	TR ^S to Owner/F	TR ^S to Owner/F	Inv all S, ST ^I -TR ^M to Owner/M	Inv other S and Owner, STW ^M to Req/M	Inv all S, STW ^M to Req/M	

BedRock differs from a canonical directory protocol in a few subtle ways. First, the coherence directory has complete control over all changes to cache block coherence states, including invalidation and eviction of blocks from the cache controllers. The sole exception is a cache may silently upgrade a block from the Exclusive (E) state to the Modified (M) state on a write operation. Second, BedRock utilizes four unidirectional coherence networks, ordered in priority from highest to lowest: Response, Fill, Command, and Request. The Request and Response networks carry messages from cache controller to coherence directory, the Command network carries messages from directory to cache, and the Fill network carries cache to cache data transfers. Third, the cache controllers never hold a block in a transient state; all transient state is hidden from the controllers by the coherence directory's processing flow. Lastly, the protocol design favors reducing complexity over maximizing concurrency.

3.1.2 Request Processing

The canonical processing flow for a BedRock request is divided into two stages. First, the directory is read and processed to determine if a cache block eviction (replacement) or invalidations are required. Then, the coherence engine determines the source of the cache block

and initiates a cache to cache transfer, performs a memory read, or responds to the requesting cache with read/write permissions for the block if it already has a valid copy. The transaction completes when the cache controller receives the block and responds to the directory with a coherence acknowledgment. Advanced directory implementations may be able to overlap some of these actions or perform speculative memory accesses to reduce request processing latency.

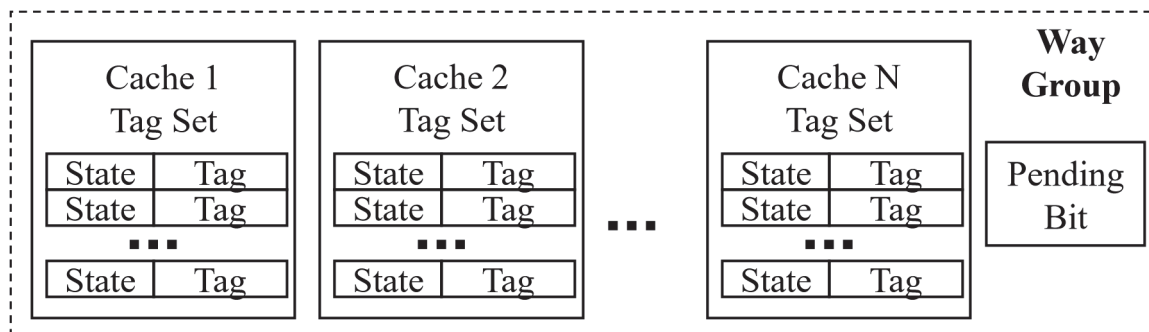


Figure 9: Way Group Organization

3.1.3 Protocol Tables

Tables 1 and 2 present the complete tabular specification [73] of the BedRock MOESIF coherence protocol for the cache controller and coherence directory, respectively. Each table uses an "Action/State" notation to describe the behavior of the controllers. Given the current coherence state for a cache block (row) and an event (column), an entry in the table describes the action taken by the controller in response to the event and the next coherence state of the block at the controller. Blank entries indicate impossible state and event pairs for the controller.

3.1.3.1 Cache Controller Protocol Table

Table 1 describes how the cache controller responds to cache actions (load or store) and coherence commands. Cache actions may either hit in the cache or trigger a new coherence request. Coherence commands are directives issued by the directory to modify the state of a block and send a message in response on the Response or Fill network. An X state indicates a valid coherence state provided by the directory in the command message that is not known *a priori* by the cache controller.

3.1.3.2 Coherence Directory Protocol Table

Table 2 describes how the coherence directory processes coherence requests. The directory may also evict cache blocks from a controller to make room for a newly requested block (Replacement). Each "Action/State" entry describes the command messages sent by the directory to complete the request and the next state of the block at the directory. Some commands generate a response to the directory, and all transactions are finalized when the cache controller sends a coherence acknowledgment to the directory. The coherence state superscripts attached to some messages provide an associated coherence state for the message. For example, an ST^I-TR^M message directs a cache controller to set the state of the target block to Invalid (I) and transfer that block in the Modified (M) state to another controller.

3.1.4 Ordering Transactions - Way Groups

The BedRock protocol allows concurrent coherence transactions to independent cache blocks, but only one transaction at a time per group of related blocks. Two cache blocks are related if they belong to the same *Way Group*. A way group, shown in Figure 9, is the collection of cache blocks that map to the same cache set in *any* cache. A *Tag Set* is simply the pairs of coherence state and address tag for all ways within a single set of a single cache. The *Pending Bit* is set when a new transaction targeting a block in the way group starts and cleared when the active transaction completes. Transactions to way groups with a set pending bit must stall until the previous transaction completes. Requests targeting the same way group may be concurrently issued by many cache controllers and are serialized by the directory.

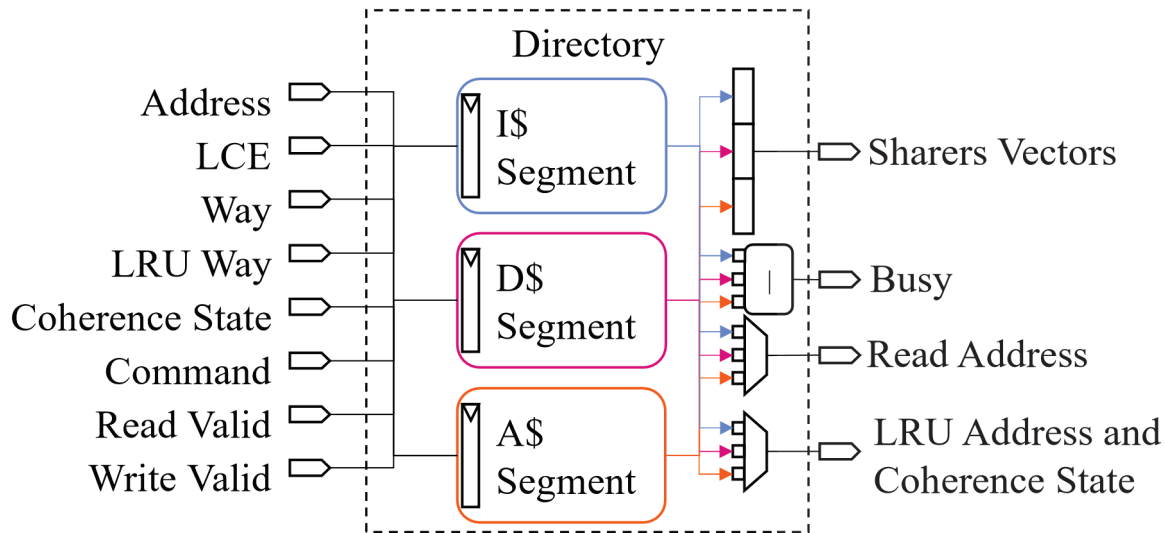


Figure 10: BP-BedRock Coherence Directory Architecture

Every way group in the system is managed exclusively by a single coherence directory. In a system where all caches participating in coherence have the same organization with S sets, there are S way groups (one per cache set). A system with varied cache organizations has a number of way groups equal to the minimum number of cache sets across all caches.

Way groups guarantee that a coherence request will only cause changes to cache blocks within the target way group. Consequently, two transactions to two different way groups are guaranteed to be independent and can be processed concurrently. The use of way groups sacrifices protocol concurrency, but also eliminates transient states at the cache controllers that are typically required to handle that concurrency, thus simplifying the protocol for the cache controllers.¹

3.1.5 BP-BedRock Directory

The BP-BedRock cache coherence engines (CCE) implement complete standalone duplicate tag directories. The organization of the coherence directory is identical in both the hardware-based (FSM) and microcode programmable (ucode) coherence engines. The directory is constructed

¹ Future work will investigate efficient mechanisms to support concurrency within way groups.

from multiple directory segments with one segment per cache type in the system. BP-BedRock's cache types are instruction, data, and optional coherent accelerator caches. Figure 10 shows a block diagram of the coherence directory. Systems without coherent accelerators do not instantiate the accelerator cache segment.

3.1.6 Coherence Directory Segment

Figure 11 shows the organization of a BP-BedRock directory segment. Each segment stores a subset of the tag sets from all caches of a single type. The tag sets of all caches are spread evenly across the CCEs, with all tags sets in a single way group mapped to a single CCE. The tag sets of a single cache are stored in sequential rows in the directory SRAMs, and each row stores related tag sets from one or more caches. If the number of *Tag Sets per Row* is less than the number of caches being tracked by the directory segment, additional blocks of *Tag Sets per CCE* rows are added to track all caches.

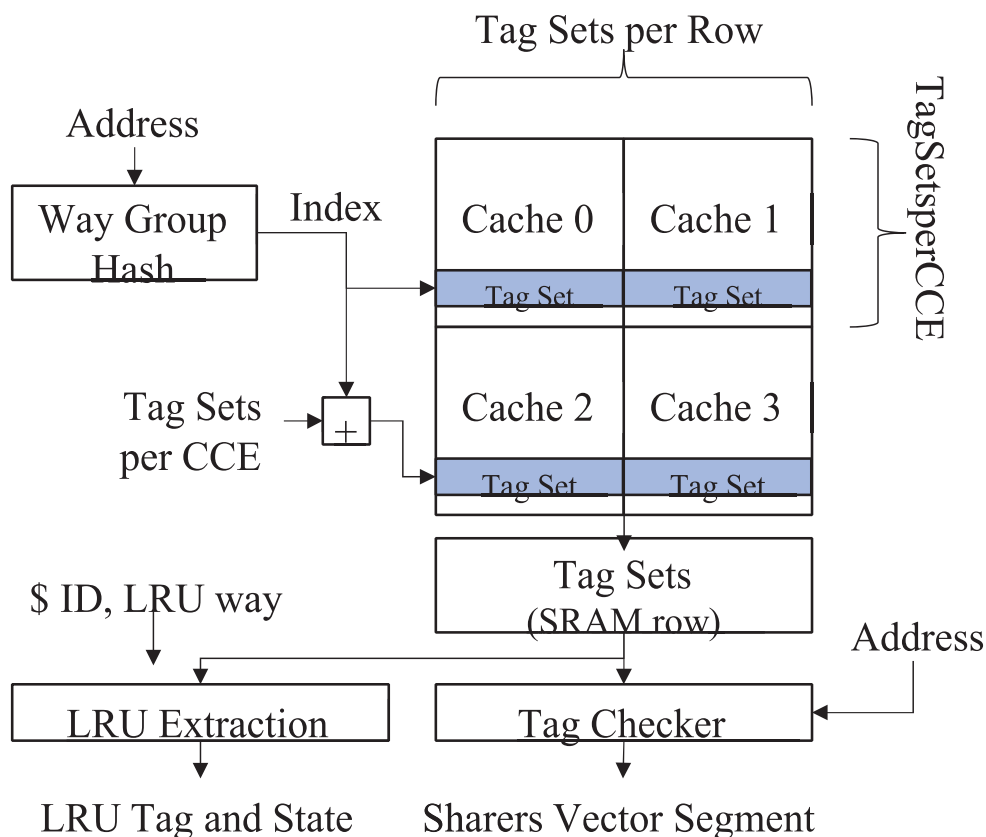


Figure 11: BP-BedRock Coherence Directory Segment

The directory segment SRAM is a single-ported synchronous read-write memory. The width of the SRAMs are sized based on prior physical design work showing that two tag sets per row is PPA-efficient, however any power-of-two value is supported. The location of a cache block is easily computed using the cache's ID and the block address, with the cache ID providing the first row of the cache's tag set rows and the address providing a row offset within the block. The least significant bits of the cache's ID also provides a horizontal offset into the target row.

3.1.7 Directory Operations and Access

A small FSM controls each directory segment and supports tag set entry reads, way group reads of all tag sets in a way group, tag set entry writes, and physical row clears. The directory segment organization provides single cycle writes and multiple cycle reads. Write operations have a latency of one cycle, and a new write may be issued every cycle. Tag set entry reads require two cycles. Way group reads require two or more cycles, depending on the total number of caches being tracked by the segment. In a BP-BedRock multicore, a way group read requires $1 + (\text{Cores}/2)$ cycles for the default directory organization.

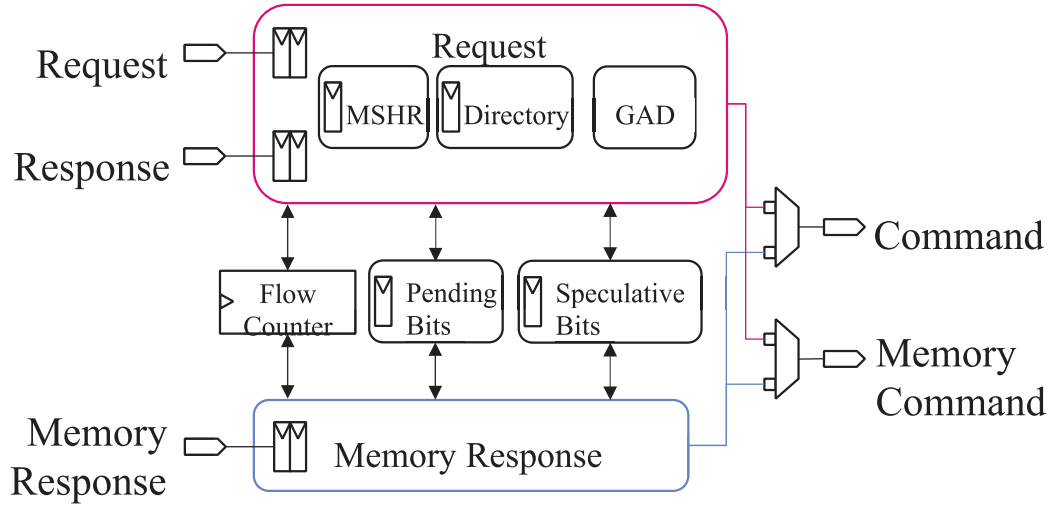


Figure 12: BP-BedRock FSM CCE Block Diagram

3.1.8 Tag Checker and LRU Extraction

BP-BedRock includes two modules that process the directory way group reads. The Tag Checker examines each directory row as it leaves the SRAM and produces the three Sharers Vectors. Each vector has one entry per cache, and the three vectors provide a cache hit bit, coherence state, and the cache way of the current LCE request's cache block. The LRU extraction module processes each directory row and extracts the tag set entry at the LRU way for the specified LCE. If the directory segment requires multiple rows to store all cache's tag sets for a single cache set, the LRU Extraction module outputs the LRU information only for the row containing the requesting LCE's tag set. The LRU way input is provided in the coherence request from the cache controller.

3.1.9 BP-BedRock FSM CCE

BP-BedRock's FSM-based cache coherence engine (FSM CCE) is a direct hardware implementation of an FSM implementing the BedRock MOESIF coherence protocol of Table 2. Figure 12 shows the FSM CCE's block diagram comprising Request and Memory Response state machines, Speculative Bits to track speculative memory reads, Pending Bits to enforce coherence transaction ordering for each way group, and a Flow Counter to provide memory network flow control.

3.1.10 Request FSM

The Request state machine processes coherence requests from the cache controllers. It instantiates a Miss Status Handling Register (MSHR) to hold processing state, the coherence directory, and a GAD module that processes the coherence directory output. The coherence directory implementation is described in Section 3.1.5. The MSHR and GAD modules are explained below, followed by a description of the state machine.

3.1.10.1 MSHR

The MSHR accumulates information related to the current LCE request during request processing. This information includes the request address, type, size, and requesting controller's ID; information about any required cache block replacement; the current state, owner, and cache way of the requested block, and a set of control flag bits. The MSHR information is filled primarily from the request message begin processed and the required coherence directory read.

Table 3: BP-BedRock MSHR Flags

Flag	Description
Write Not Read	Write request
Uncached	Uncached request
Non Exclusive	Non-Exclusive request
Atomic	Atomic operation request
Atomic No Return	Atomic no return request
Cacheable Address	Request to cacheable memory
Pending	Pending bit set on last read
Cached Shared	Block cached in S at other LCE
Cached Exclusive	Block cached in E at other LCE
Cached Modified	Block cached in M at other LCE
Cached Owned	Block cached in O at other LCE
Cached Forward	Block cached in F at other LCE
Replacement	Replacement required
Upgrade	Request can be resolved with a permission upgrade

Table 3 lists the control flow flags stored in the MSHR. These flags come mostly from the current LCE request and the GAD module after a directory read, and are used to make efficient control flow decisions. The top six flags record properties of the current request. The Pending flag is set by the result of a pending bits read operation. The remaining flags are set after reading and processing the directory. The Replacement flag is set if a cache block replacement is required, and the Upgrade flag is set if the request can be resolved with a read/write permission

upgrade to the requesting cache. The *Cached State* flags are set if the requested cache block exists in the corresponding state in any cache other than the requester.

3.1.10.2 GAD

During request processing, the FSM CCE reads the coherence directory to extract the current state of the target block across all caches in the system. The raw directory data is processed by the tag checker and output as the sharers vectors. The GAD, or Generate Auxiliary Directory Information, module consumes the sharers vectors and LRU information and computes a subset of the MSHR control flags; the owner, location, and coherence state of the target block, if an owner exists; and the cache way of the block within the requesting cache, if present. The GAD module takes a single cycle to execute and is invoked in the cycle following the coherence directory read.

3.1.10.3 Request FSM

Figure 13 shows the FSM CCE's request processing state machine. The FSM processes one request at a time and runs without interruption. States requiring multiple cycles to execute (outlined in magenta) are labelled with an execution latency. All other states execute in a single cycle. Execution begins in the Ready state, where requests are classified by request address as either coherent if they target cacheable memory or uncacheable if targeting uncacheable memory.

All requests targeting cacheable memory participate in the coherence protocol and require a coherence directory read. These requests are processed through an initial set of states that check the pending bits for an open transaction to the same way group, issue a speculative memory read for cacheable requests, read and process the directory, and then write the next state for the block into the directory. After initial processing, the FSM executes only those steps required by the request. Diamonds in the state machine diagram represent control flow decisions and have a cost of zero cycles. A key benefit of the FSM-based CCE implementation is that it can concurrently perform protocol processing and make control flow decisions, thereby incurring no overhead to transition between protocol processing steps.

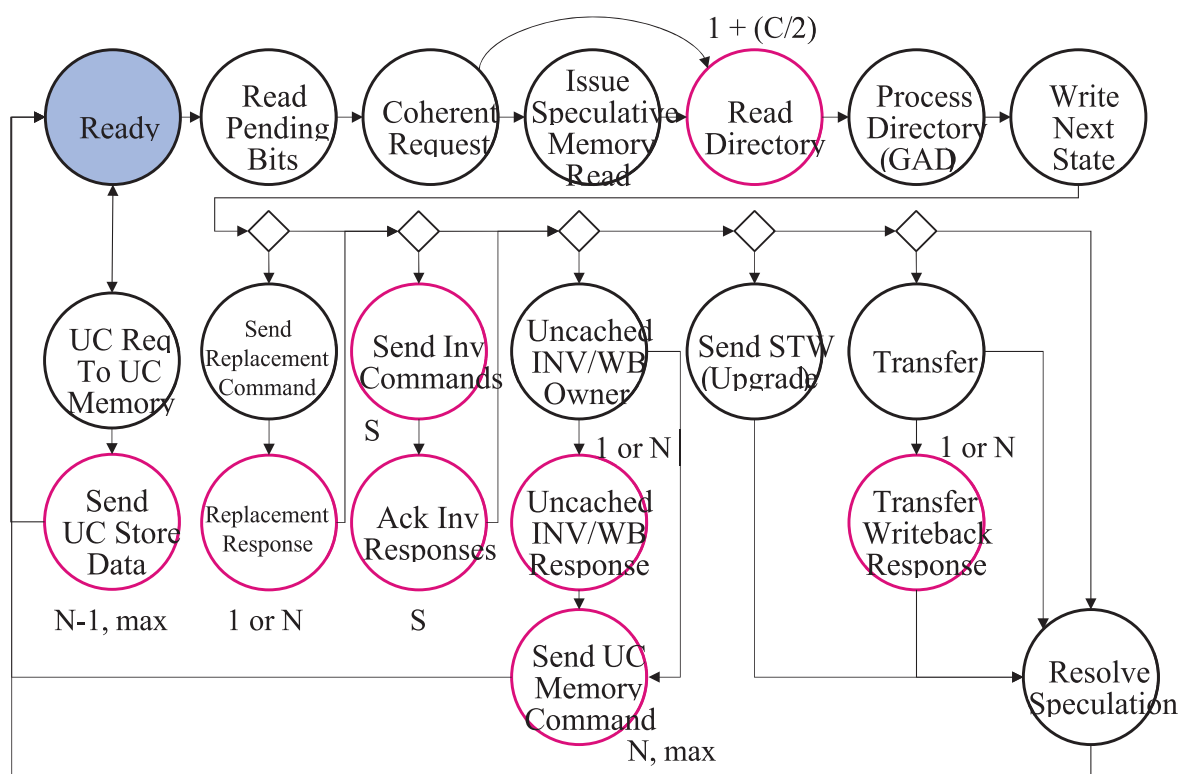


Figure 13: BP-BedRock FSM CCE Request State Machine

Cacheable requests may require a cache block replacement to evict a block from the requesting cache or cache block invalidations to caches that contain valid copies of the requested block. After replacement and invalidations are complete, the request is resolved by either sending an upgrade command when the requester needs read/write permissions and has a valid read-only copy of the block, initiating a cache to cache transfer if the block is owned by another cache, or confirming that the block will be sourced from memory.

Uncacheable requests to cacheable memory invalidate and writeback the target block from all caches that contain a valid copy, before issuing the uncached operation to memory.

Uncacheable requests to uncacheable memory require minimal processing and result in either an uncached load or store being issued to memory. Uncacheable requests require one cycle to send the message header and first data beat plus one cycle per additional data beat. These requests do not participate in the coherence protocol.

3.1.10.4 Atomics

BP-BedRock supports atomics at the L1 and L2 caches, with L1 atomics enabled by default. L1 atomics result in the L1 data cache requesting write permission for the target block before the cache executes the operation after the block fill completes. RISC-V LR/SC sequences are handled similarly with the L1 data cache requesting write permissions at the LR operation and are guaranteed to eventually succeed per the RISC-V specification. L2 atomic requests are

executed at the L2 cache, with the coherence directory invalidating the target block from all L1 caches before the operation is sent to the L2 cache. All atomic operations are ordered by the coherence protocol (requesting write permissions for L1 atomic or LR/SC) or the coherence directory (serialization of requests for L2 atomic)

3.1.11 Pending Bits

The FSM CCE implements the pending bits attached to each way group as a collection of small counters external to the coherence directory. A way group's pending bit is considered set if the counter is non-zero and unset if the counter is zero. Pending bit writes are synchronous, while reads are asynchronous and support write to read forwarding. The CCE increments a way group's pending bit when beginning request processing or issuing a memory command. Pending bits are decremented by consuming memory responses or coherence acknowledgment responses, and when the CCE finishes processing uncached requests to coherent memory.

3.1.12 Speculative Bits

The Speculative Bits record information about speculative memory reads issued during request processing. There is one Speculative Bits entry per way group. Each entry includes a coherence state, a speculative bit, a squash bit, and a forward-modified bit. The CCE sets a way group's speculative bit when issuing a speculative memory request and clears the bit once the source of the cache block is determined after reading the directory. A speculative memory request will be squashed if the squash bit is set, forwarded with the stored coherence state when the forward-modified bit is set, or forwarded without modification if all bits have been cleared. Memory requests are squashed when the block can be supplied by a cache to cache transfer.

3.1.13 Memory Response FSM

The Memory Response FSM is a three-state multi-cycle state machine that processes BP-BedRock Memory Response messages returning from the L2 cache / memory or I/O devices. The state machine examines each memory response as it arrives and either forwards it to the appropriate LCE or sinks the response from the memory network. All memory responses include a speculative bit that is set if the response was generated by a speculative memory command. Speculative responses are either squashed or forwarded to a cache controller, as explained above. Non-speculative responses either carry data from a cached or uncached read, which are forwarded to the requesting controller, or are header-only responses to memory write commands, which are sunk by the FSM.

3.1.14 BP-BedRock ucode CCE

The BP-BedRock microcode-programmable CCE (ucode CCE) is a user-programmable coherence engine with a two-stage fetch-execute pipeline, 64-bit general purpose registers and datapath, specialized coherence protocol processing logic, and a custom RISC ISA specialized for coherence protocol processing. The ucode CCE is specialized for the BedRock coherence protocol and provides programmability to enable system- and application-specific functionality on top of the coherence protocol. Figure 14 shows a block diagram of the BP-BedRock ucode CCE. The programmable CCE reuses many modules from the FSM CCE, including the Coherence Directory, GAD, Pending Bits, and Speculative Bits. BP-BedRock provides

microcode implementations of the BedRock MOESIF, MESI, MSI, and EI protocols. The baseline MOESIF protocol with speculative memory access closely matches the processing flow of the FSM CCE implementation. The remainder of this section describes the microcode ISA and modules unique to the ucode CCE.

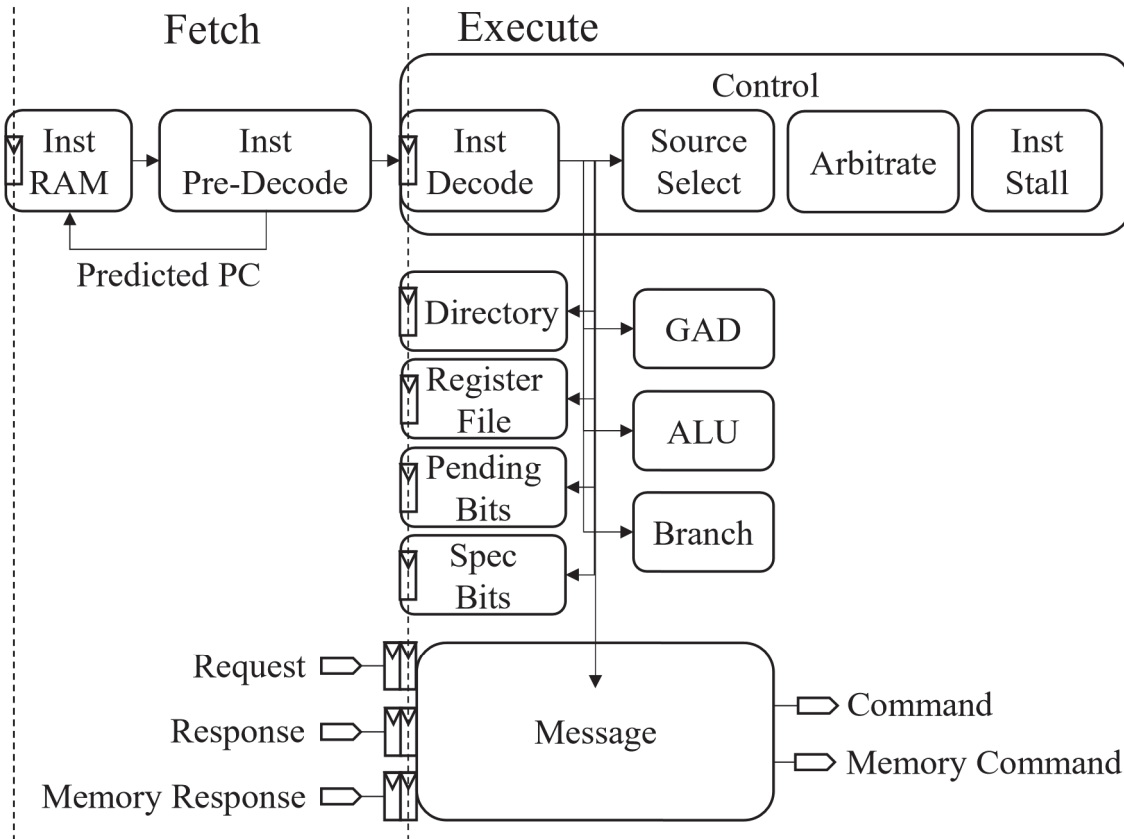


Figure 14: BP-BedRock Microcode-Programmable CCE Block Diagram

3.1.15 Microcode ISA

The ucode CCE executes a custom RISC instruction set architecture (ISA) specialized for coherence protocol processing. The ISA is divided into a *Base ISA* and a *Coherence ISA*. The Base ISA contains standard RISC instructions for arithmetic, branch, and data movement operations. The Coherence ISA contains instructions specialized for coherence processing, such as manipulating control flags, flag-based control flow, reading and writing the coherence directory, and sending or receiving messages. All branch instructions are tagged with a static taken/not-taken prediction bit and the branch mispredict penalty is one cycle. Table 4 lists the BP-BedRock Coherence ISA instructions, which are divided into Flag, Directory, and Queue operations. Directory read and queue operations may take more than one cycle to execute depending on functional unit conflicts and latencies, while all other instructions execute in a single cycle.

A *key contribution* of BP-BedRock is its specialized Coherence ISA that accelerates protocol processing by invoking coherence-specialized functional units. These instructions are critical to closing the performance gap between the ucode and FSM-based CCEs.

3.1.15.1 Flag Instructions

Flag instructions can set or clear flags, perform logic operations on pairs of flags, and make control flow decisions based on the state of a programmer-selected set of flags. The most important of these are the flag-based branch instructions (bf, bfz, bfnz, bfnz). Each flag-based branch examines a set of programmer-selected MSHR flags, encoded in a bitmask within the instruction, and branches the microcode PC to the supplied target PC if the branch condition is met. A single flag-based branch instruction is able to replace a sequence of regular branch instructions, thereby accelerating common protocol processing control flow decisions.

Table 4: BP-BedRock ucode CCE Coherence ISA

Op	Format	Function
sf, sfz	op flag	flag = 1, flag = 0
andf, orf	op flag flag gpr	rd = flag op flag
nandf, notf	op flag flag gpr	rd = flag op flag
notf	notf flag gpr	rd = flag
bf	bf tgt flag [flag...] [pt]	pc = tgt if all flags 1
bfnof	bfnof tgt flag [flag...] [pt]	pc = tgt if all flags 0
bfz	bfz tgt flag [flag...] [pt]	pc = tgt if any flag 1
bfnz	bfnz tgt flag [flag...] [pt]	pc = tgt if any flag 0
rdp	rdp addr=<a>	pf = pending_bits[addr]
rdw	rdw addr=<a> lce=<l> lru_way=<w> [src=<ra>]	produce sharers, lru info, etc.
rde	rde addr=<a> lce=<l> way=<w> [src=<ra>] dst=<rd>	rd = addr, sh_states[lce] = state
wdp	wdp addr=<a> p=<0,1>	pending_bits[addr] +/- 1
clp	clp addr=<a>	pending_bits[addr] = 0
clr	clr addr=<a> lce=<l>	clear directory row
wde	wde addr=<a> lce=<l> way=<w> [src=<ra>] state=<s> [state_imm]	dir[addr, lce] = [tag, state]
wds	wds addr=<a> lce=<l> way=<w> [src=<ra>] state=<s> [state_imm]	dir[addr, lce] = [..., state]
gad	gad	execute GAD unit
wfq	wfq queue [queue...]	Wait for message on queue
pushq	pushq queue cmd addr=<a> lce=<l> way=<w> [src=<ra>] wp=<0,1> spec=<0,1>	push message to queue
popq	popq queue [wp]	dequeue message, write pending bit
poph	poph queue rd	capture message header
specq	specq spec_cmd addr_sel [state]	speculation bits operation
inv	inv	send invalidations

3.1.15.2 Directory Instructions

Directory instructions accelerate directory read, write, and processing operations by invoking the coherence directory and GAD modules. Directory way group reads require only $1+(C/2)$ cycles to execute, compared to tens or hundreds of cycles that would be required by a general-purpose implementation of the same routine using `for` loops. Pending bit and directory entry reads require one and two cycles, respectively. Directory writes execute in a single cycle. The GAD module executes in a single cycle, compared to a cost of tens of instructions to implement

equivalent logic in general-purpose RISC code. Additionally, the flag outputs of the GAD module never need to be recomputed by the microcode program, saving many additional cycles for every flag-based branch instruction.

3.1.15.3 Queue Instructions

Queue instructions enable efficient sending and receiving of coherence protocol and memory messages. The ucode CCE is able to send and receive messages with a cost of one cycle per message header or data beat. The invalidate (inv) instruction further accelerates coherence protocol processing by invoking a small hardware-implemented state machine within the ucode CCE's message unit to efficiently send invalidation commands to all caches with a Shared (S) copy of the specified cache block at a rate of one message per cycle. A general-purpose RISC routine for invalidations would require at least a few instructions per invalidation sent if executed in a tight `for` loop.

3.1.15.4 Programming the CCE

The CCE is programmed at the microcode level. A custom assembler applies a limited set of instruction transformations to map available software pseudo-ops into hardware-implemented microcode instructions. BP-BedRock's MOESIF protocol microcode is only 125 instructions, which includes support for uncacheable access to both cacheable and uncacheable memory and system initialization.

3.1.16 Instruction Fetch and Predecode

The Instruction RAM and Predecode modules comprise the Fetch Stage of the ucode CCE. The instruction RAM unit contains the microcode instruction memory and logic to determine the next microcode program counter (PC). The predecode module examines the just fetched instruction to detect branches and provides a predicted PC to the fetch logic. A stall in the Execute stage halts instruction fetch, and a branch mispredict squashes the just fetched instruction while redirecting fetch to the correct PC.

3.1.17 Instruction Decode and Stall

The Instruction Decode unit expands the current narrow microcode instruction into a wider decoded instruction containing control signals for the Execute stage's functional units.

The Instruction Stall unit detects functional unit hazards and message stalls due to busy or empty networks. It outputs a stall signal that is routed to the Fetch stage and the instruction decoder, causing the current instruction to replay in the following cycle. Functional unit hazards occur when the ucode engine and message unit access the same resource.

3.1.18 Source Select and Arbitration

The source select module routes operands to the ucode CCE's functional units as specified by the current instruction. Source operands may come from the GPRs, MSHR, inbound messages, or directory outputs.

The arbitration unit controls access to the coherence directory, pending bits write port, and speculative bits read port. In a given cycle, each of these three resources may be used by either the microcode instruction or the message unit. The message unit has priority over the ucode engine for each resource, causing the ucode engine to stall when it loses arbitration, which helps guarantee deadlock freedom in the coherence protocol.

3.1.19 Register File

The Register File stores the CCE's Miss Status Handling Register (MSHR), eight 64-bit general purpose registers (GPRs), a coherence state register, and an auto-forward control register. The coherence state register holds a default coherence state that can be used as a source operand for coherence and memory commands. The auto-forward control register is a single bit register that controls whether the ucode CCE's message unit will automatically process memory response messages. It is set by default, but can be disabled via the microcode.

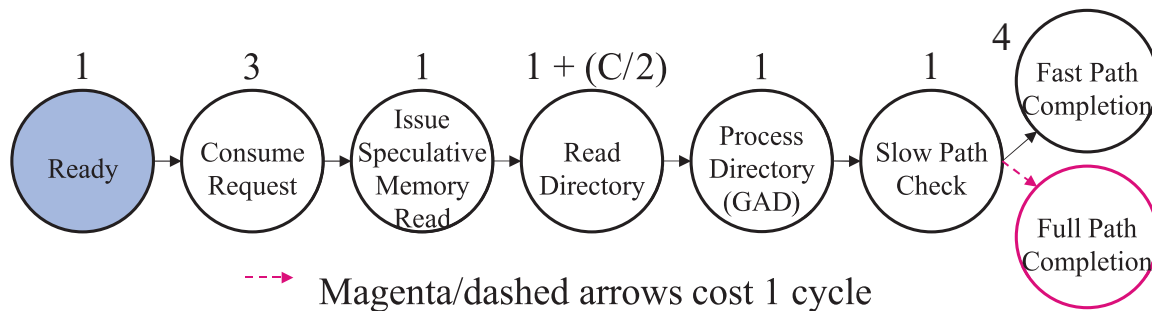


Figure 15: MOESIF Microcode Processing Flow - Initial and Fast Path

3.1.20 Functional Units

The ucode CCE includes functional units tailored for both general purpose and coherence specialized execution. The Coherence Directory, Pending Bits, Speculative Bits, and GAD modules are all reused without modification from the FSM CCE design and are described in Section 3.1.9.

3.1.20.1 ALU and Branch

The Arithmetic Logic Unit (ALU) has a 64-bit datapath and supports add, subtract, shift, and bitwise operations. The hardware ALU is very simple, and many operations available to the programmer are supported at the software level as pseudo-ops using assembler transformations.

The Branch unit resolves branch operations and validates the Fetch stage's speculative fetch predictions. The result of the branch is compared to the branch prediction made in the Fetch stage to determine if a mispredict occurred. Mispredicts redirect the fetch stage to the resolved PC and result in a single cycle bubble in the execute stage.

3.1.20.2 Message

The Message unit is responsible for sending and receiving all coherence and memory messages. It can write the pending bits, read the speculative bits, and write the coherence directory. It also contains a memory credit flow counter that limits the number of outstanding memory commands. The message unit has two state machines that process memory response messages and send or receive messages as directed by the microcode program. The memory response state machine is identical to the one found in the FSM CCE. However, it can be disabled via the microcode program by clearing the auto-forward register in the register file. The other state machine sends and receives messages based on the currently executing instruction. This FSM also implements the specialized invalidation routine logic that can issue one invalidation per cycle.

3.1.21 Request Processing

The ucode CCE's microcode programs implement an execution flow similar to the FSM CCE. Figures 15 and 16 show the MOESIF protocol microcode processing flow for cacheable requests. Each circle represents one or more microcode instructions and is called a subroutine, and each subroutine is labelled with its execution latency in cycles. Transitions between states that are black/solid cost zero cycles and those that are magenta/dashed cost one cycle, which indicate a branch mispredict.

States outlined in magenta require more than one cycle to execute.

Figure 15 shows the initial subroutines executed by the MOESIF protocol for all coherence requests to cacheable memory. The microcode is optimized to handle load requests to blocks in the Invalid state, which is called the *Fast Path*. All other requests are handled by the *Full Path*. A fast path request is fulfilled by a memory access, which is issued speculatively before the directory read occurs. The directory read latency is $1 + (\text{Cores}/2)$ cycles. Following the directory read, a single cycle is required to confirm the request does not require the full path for processing before finalizing the fast path processing.

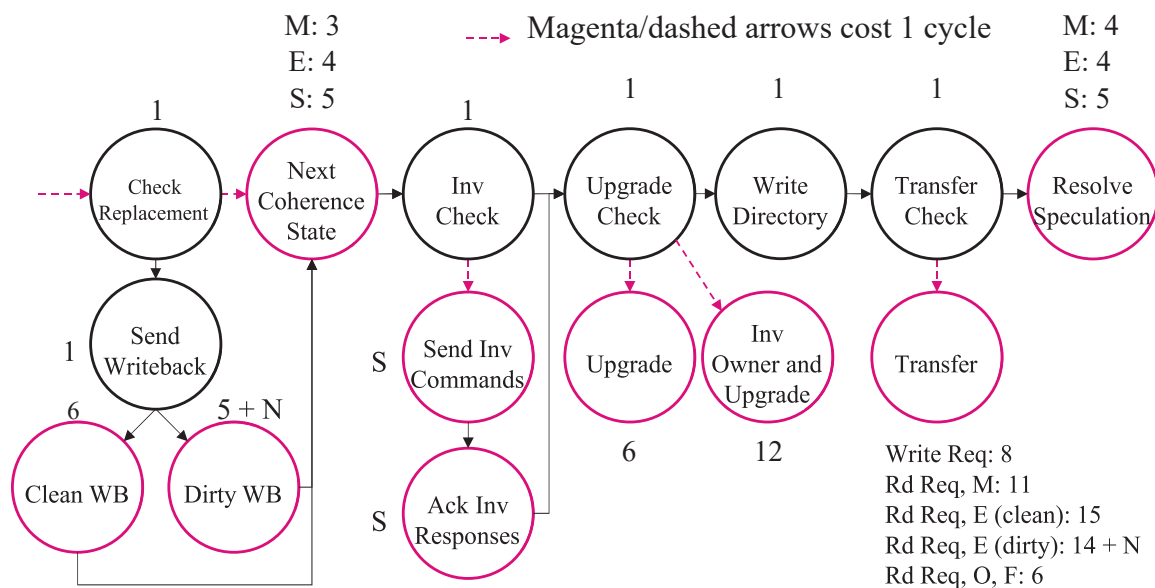


Figure 16: MOESIF Microcode Processing Flow - Full Path

The ucode CCE MOESIF protocol continues processing non-fast path requests using the flow shown in Figure 16. These requests either target a block that is already cached somewhere in the system, require a cache block replacement at the requesting LCE, or are write requests. The total execution latency of all requests can be computed directly from the processing flow diagrams by adding the cost per state visited plus one cycle per branch mispredict transition (magenta/dashed arrow). The *Compute Next Coherence State* and *Resolve Speculation* subroutines require a variable number of cycles to execute, depending on the coherence state that will be supplied to the requesting cache. Invalidations are offloaded to the message unit and require a single cycle per invalidation sent or received, for a total of $2 * S$ cycles to complete all invalidations. Writebacks during cache block replacement or from a transfer require N cycles to forward the N cache block data beats from the LCE response to the memory network. The *Transfer* subroutine occupancy depends both on the type of request (read or write) and the coherence state of the block at the directory. Request processing completes after performing an Upgrade, Transfer, or Resolving Speculation (indicated by states with no out transitions).

3.1.22 Protocol Verification

The BedRock cache coherence protocol is similar to, yet subtly different from, commonly understood directory-based protocols [44,60,62,74,75]. Therefore, it is important that the protocol itself is shown to be correct. BedRock's MESI protocol has been verified correct using CMurphi [64], an improved version of the Murphi [46] model checking framework, for a system with eight coherent caches and a single coherence directory. BedRock's CMurphi description models a single cache block, a single coherence directory, and unordered networks. These assumptions are valid because, by definition, cache coherence is constrained to a single memory location, BedRock's coherence directories operate independently from one another in a multi-directory system, and every cache block is managed by a single directory.

Table 5: Coherence Directory Overhead Comparison

Caches	BP-BedRock	Complete	Coarse (8-bits)
2	6.25%	7.81%	7.81%
4	6.25%	7.81%	9.38%
8	6.25%	9.38%	9.38%
16	6.25%	10.94%	9.38%
32	6.25%	14.06%	9.38%
64	6.25%	20.31%	9.38%

Table 6: BP-BedRock CCE Request Occupancy (MOESIF)

Request	LCE State	Directory State	FSM CCE Occupancy (cycles)	ucode CCE Occupancy (cycles)
Read	I	I	$8 + (C/2)$	$12 + (C/2)$
Excl Read NE				$26 + (C/2)$
Read	I	S	$8 + (C/2)$	$26 + (C/2)$
		E (clean)	$9 + (C/2)$	$36 + (C/2)$
		E (dirty)	$9 + (C/2) + N$	$35 + (C/2) + N$
		M	$9 + (C/2)$	$32 + (C/2)$
		O, F	$9 + (C/2)$	$27 + (C/2)$
Write	I	I	$8 + (C/2)$	$23 + (C/2)$
		S	$8 + (C/2) + (2 * S)$	$24 + (C/2) + (2 * S)$
		E, M	$9 + (C/2)$	$27 + (C/2)$
		O, F	$9 + (C/2) + (2 * S)$	$28 + (C/2) + (2 * S)$
Write	S	S	$9 + (C/2) + (2 * (S - 1))$	$24 + (C/2) + (2 * (S - 1))$
		O, F	$9 + (C/2) + (2 * S)$	$30 + (C/2) + (2 * S)$
Write	O, F	O, F	$9 + (C/2) + (2 * S)$	$24 + (C/2) + (2 * S)$

3.1.23 BedRock Directory Overhead

BP-BedRock utilizes a standalone, duplicate tag coherence directory. However, this is not the only choice of directory organization that could have been made. Therefore, an investigation of the coherence directory storage overhead of BP-BedRock’s directory was performed and compared to the overhead of standalone complete and coarse directories [60,86]. The analysis assumes 8-way associative L1 caches with 64 sets and 64-byte cache blocks, which are the default for BP-BedRock, with private instruction and data caches per core. The system uses 28-bit physical address tags and 3-bit coherence states. Table 5 shows the results of the overhead analysis. BP-BedRock’s duplicate tag directories have a constant storage overhead of 6.25%, which is less than both the complete and coarse directories. This property is greatly beneficial for the physical design of BP-BedRock, where a slice of the coherence directory is instantiated on

each multicore tile and the multicore is constructed by instantiating tiles in a 2D mesh. The constant overhead results in a fixed directory size per tile, regardless of core count, which enables the use of a hierarchical, tile-based backend design flow for ASIC implementations.

3.1.24 CCE Request Processing Occupancy

The BP-BedRock coherence engines both implement the BedRock MOESIF cache coherence protocol. The FSM CCE is a direct hardware implementation of the protocol's request processing flow, while the ucode CCE executes a microcode program implementing the protocol and provides flexibility to add additional processing and functionality to protocol processing. The throughput and latency of cache coherence engines are important to overall system performance, therefore it makes sense to first analyze each coherence engine's request processing occupancy.

Table 6 presents the request processing occupancy for both coherence engines. Processing occupancy, given in cycles, is the number of cycles required in a best-case, no-contention execution to process a coherence request. Three constants are used in the processing occupancy computations: C is the number of cores in the multicore processor, N is the number of data beats required to send a full cache block across the coherence network data channels, and S is the number of caches holding a block in the Shared (S) coherence state, called the sharers. The data presented are the number of cycles that the coherence engine is busy processing a single request, and processing occupancy are derived directly from Figures 13, 15, and 16. The numbers presented assume that a cache block eviction (replacement) is not required. Occupancy provides insight into the maximum achievable throughput of the coherence engine designs. The request occupancy does not include the time required to process memory responses, which are handled by a separate state machine in both designs that operates concurrent to request processing. Network time is also excluded as the time for messages to transit networks is the same for both designs.

3.1.24.1 FSM CCE Occupancy

The FSM CCE has a base request processing occupancy of $7+(C/2)$ cycles, incurred by all requests, as it moves from Ready through Write Next State in Figure 13. During this initial processing, the request is consumed, the directory is read and processed, and the directory entry for the requesting cache is updated with the final next state for the block. Then, depending on the specific request and state of the target block in the system, the FSM executes only those steps required to complete the transaction. The diamonds in Figure 13 indicate control flow decisions, but have a cost of zero cycles. The key performance advantage of the FSM-based design is that control flow decisions are effectively free; in any given state, the next state is computed concurrently with the protocol processing occurring in the state. Thus, after executing the initial processing, the added cost to complete a request is simply the cost of the remaining states visited. The worst-case request, in terms of occupancy, is a write request to a block in the O or F state, which is owned by a single cache but shared by many caches and may be present in every single cache in the system. A cache block replacement adds either two or $1+N$ cycles of processing time for clean and dirty blocks, respectively.

3.1.24.2 ucode CCE Occupancy

The ucode CCE incurs execution overheads relative to the FSM-based CCE primarily due to its inability to execute protocol processing and control flow in the same instruction and the fact that each control flow decision requires a separate instruction. As described in Section 3.1.21, the MOESIF microcode program includes a fast path (Figure 15) to process regular reads for blocks in the Invalid state. This path has an execution overhead of only four cycles compared to the FSM-based coherence engine. The fast path is effectively a single basicblock of microcode, and therefore can be executed at a rate matching that of the FSM-based engine. However, all other requests must branch to the full path, shown in Figure 16, which is capable of performing replacements, invalidations, and cache to cache transfers. The base occupancy for both paths is only one cycle greater than the FSM-based engine at $8 + (C/2)$ cycles. Requests processed by the full path have occupancy overheads between 15 to 25 cycles. Significant overheads are incurred for subroutines that require multiple control flow decisions. In particular, determining the proper next coherence state for the block, resolving the outcome of the speculative memory access, and initiating cache to cache transfers all add significant latency to request processing. A cache block replacement adds either seven or $6 + N$ cycles of processing time for clean and dirty blocks, respectively.

Table 7: BP-BedRock ucode CCE Resource Overheads

Design	Component	Resource	Overhead
ASIC	Multicore		4.08%
	Tile	Die Area	4.28%
	CCE		31.08%
FPGA	Multicore	Logic LUTs	6.32%
		BRAM	1.54%
	Tile	Logic LUTs	7.08%
		BRAM	1.54%
	CCE	Logic LUTs	66.19%
		BRAM	1 per CCE

3.1.24.3 Discussion

Designing a microcode programmable coherence engine with processing occupancy equivalent to an FSM-based design remains an open challenge. The ucode CCE's contribution to this problem is its use of coherence-specific ISA extensions to efficiently offload common coherence protocol operations, such as reading and processing the directory and performing invalidations, to specialized functional units. Despite the control flow overheads experienced by the ucode CCE, processing occupancy overheads are limited to tens of cycles through the use of coherence-specific instructions. The coherence directory hardware accelerates directory operations, replacing a potentially expensive loop-based microcode routine requiring tens to hundreds of cycles with a comparatively inexpensive fixed-latency execution. Similarly, constructing and sending coherence messages are executed by the specialized message unit at a

rate of one cycle per header or data beat. A reasonable software implementation might require tens of cycles per message send or receive using memory-mapped messaging queues. The ucode CCE's inclusion of coherence-specialized functional units points to a promising path forward for programmable coherence engines that are specialized for protocol processing yet flexible enough to enable unique system features.

3.1.25 Area, Utilization, and Timing

BlackParrot, including BP-BedRock, has been silicon validated using GlobalFoundries 12nm FinFET process and FPGA validated in an 8-core configuration for each coherence engine using a Xilinx Ultrascale+ VCU128 development platform [94]. Table 7 provides area and resource utilization overheads for ASIC and FPGA-based designs. The overheads listed are normalized to designs using the FSM-based coherence engine. The more efficient ASIC implementations show the introduction of programmability into the coherence system comes at a small area cost of only 4.08% extra die area for the entire multicore and a 4.28% increase per BlackParrot Tile. Each BlackParrot Tile comprises a BlackParrot core, its 32 KiB L1 D\$ and I\$, a 64 KiB slice of the distributed L2 cache, the on-chip networks and routers to connect tiles, and an instance of the BP-BedRock coherence engine and directory. The L2 cache acts as a memory-side buffer and does not participate in the coherence protocol. All SRAM macros are hardened in the ASIC flow, and the multicore is a 2-D mesh of Tiles with minimal additional surrounding logic. These area overheads are largely due to the addition of a microcode instruction SRAM. In the FPGA implementations, the logic utilization increases by only 6.32% and 7.08% for the entire multicore and per tile, respectively, when using the ucode CCE. Each programmable CCE additionally requires a single 18 Kib block RAM resource, which amounts to a 1.54% increase in 18 Kib block RAM resources². Additionally, both coherence engine implementations meet the same design target frequency in both the ASIC and FPGA implementations.

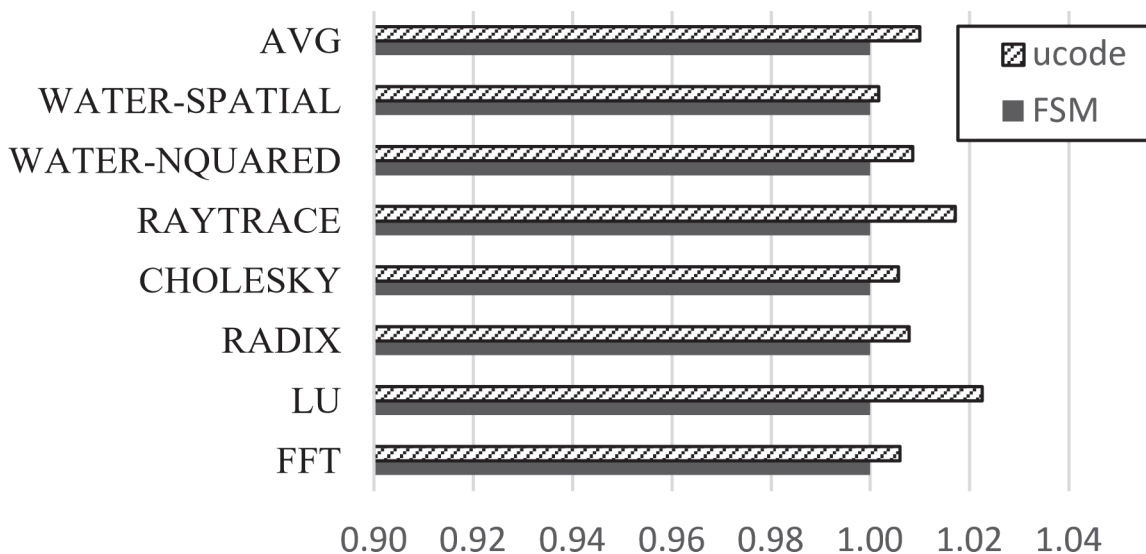


Figure 17: 8-core BP-BedRock Splash-3 Normalized Execution Time

3.1.26 Splash-3 Performance

To compare the impact of coherence engine design on system performance a collection of benchmarks from the Splash-3 [71] suite were run on FPGA-based 8-core BP-BedRock systems with the FSM-based and microcode programmable coherence engines. The benchmarks were compiled for the RISC-V ISA using `gcc` targeting a Linux environment, and invoked to execute on all 8 available processor cores. The FFT, LU, RADIX, and CHOLESKY programs are smaller kernel programs, while the remaining three programs are larger application programs. Readers are referred to [71] and [87] for more details on the synchronization and memory characteristics of these programs. Wall-clock execution time for all benchmarks ranged between tens of seconds and tens of minutes.

The BP-BedRock FPGA designs instantiate an 8-core BlackParrot multicore with 32 KiB L1 instruction and data caches, a 512 MiB shared L2 cache, and 2 GiB of HBM-based main memory. The multicore instance was clocked at 50 MHz. The programs were run in a Linux-based OS environment constructed using BuildRoot [40] and Busybox [41] with Linux kernel v5.15 [83] and OpenSBI v1.0 [61].

Figure 17 shows total execution time measured using the `time` utility for each benchmark, averaged over three runs, and normalized to the FSM-based multicore design. Our experiments show that the microcode programmable coherence engine is within 1% of the hardware-based FSM’s performance on average, and only 2.3% slower at worst. Despite having a 15 to 25 cycle best-case processing occupancy overhead, the ucode CCE multicore experiences a very small performance slowdown. Intuitively, this result makes sense as any program with reasonably good cache utilization and low miss rates will only invoke the coherence system on a cache miss. If misses are infrequent, the overall impact on system performance will be small, which follows directly from the standard average memory access time computation.

This result indicates a promising path forward for further exploration of programmable coherence engines. Careful design of the protocol processing paths can keep a programmable coherence engine competitive with a fixed-function engine, while the flexibility of a programmable system can unlock exciting new system features.

²36 Kib block RAMs are counted as two 18 Kib block RAMs for this analysis.

3.1.27 Summary of BedRock Progress

This report section presents BP-BedRock, the open-source cache coherence protocol and system implemented within the BlackParrot 64-bit RISC-V multicore processor. BP-BedRock implements the BedRock directory-based cache coherence protocol and includes two different open-source coherence protocol engines, one FSM-based and the other microcode programmable. BP-BedRock has been validated in both ASIC- and FPGA-based implementations. Analysis shows that the programmable coherence engine increases die area by only 4% in an ASIC process and increases logic utilization by only 6.3% on FPGA with one additional block RAM added per ucode CCE. FPGA-based experiments with the Splash-3 benchmark suite show that the programmable controller’s performance is only 1% slower than

the FSM-based CCE on average (2.3% worst-case). These results show that it is possible to introduce programmability into the cache coherence system at a reasonable complexity and performance cost, encouraging further exploration into the benefits of programmable coherence systems.

3.2 FPGA Integration

Hardware accelerators are an active field of research in computer architecture as one solution to overcome hurdles such as dark silicon. A host processor interfaces with an accelerator and offloads the time-consuming computations onto it. BlackParrot is one such accelerator host with promising performance comparisons with other similar RISC-V processors, and it is validated in silicon through a tapeout in 12 nm technology. BlackParrot is currently an industrial-strength ASIC Design, but ASIC flows are not readily available to individual users outside research groups that want to use BlackParrot in their designs. Open-source ASIC flows mitigate this to a certain extent, but taping out a chip is a costly exercise. An alternative to ASIC design is to use simulation only to validate the design, but this does not provide silicon validation and is an issue for long-running benchmarks which could run for days. In order to provide users with a silicon option for their designs using BlackParrot, the most viable option is an FPGA system. FPGAs allow rapid design iterations and provide an opportunity to prototype the system in real hardware. This section of the report examines building out BlackParrot's FPGA environment by creating an in-house system using the Zynq-7000 SoC available on specific FPGAs and supporting hardware cosimulation for the BlackParrot design.

3.2.1 Introduction

Since the advent of the RISC-V [84] open-source instruction set architecture (ISA), there has been a steady increase in the number of processors [35,42,51,65,95] built using this ISA across academia and industry. At the same time, hardware accelerators are an active field of research since specialization is a viable solution to overcome the dark silicon problem [77], with accelerators designed for various applications [45,48,57,63,70]. The BlackParrot processor aims to be an accelerator host multicore system.

As shown in [65], the per-core performance of BlackParrot is very competitive compared to other similar RISC-V processors. BlackParrot is silicon-validated in 12 nm technology, and ASIC Design continues to be a significant focus. However, while these chips could be used within the research group, supporting only an ASIC flow does not attract many users because tapeouts are prohibitively expensive. Another option is to support a simulation-only setup for the validation of designs using BlackParrot. While simulation is the first step towards ensuring correct functionality, it is an issue for long-running benchmarks, some of which could run for days. These issues require further exploration into other alternatives for extending the community reach of BlackParrot, and Field Programmable Gate Arrays (FPGAs) are an attractive option due to their reprogrammability, capacity for fast design iterations, and reasonable costs.

This section of the report describes the creation of a setup for BlackParrot with hardware cosimulation on boards containing the Zynq-7000 SoC (ZynqParrot), allowing users to simulate the design and run it on the board using the same testbench written in C++.

3.2.2 ZynqParrot

This section describes the ZynqParrot system. Any FPGA containing the Zynq Processing System (PS) [93] that runs Linux will be suitable to implement this system. The idea is to implement BlackParrot in the Programmable Logic (PL), connect it to the Zynq PS, and run the tests on the PS Linux, which communicates with the PL via AXI ports. This setup would avoid PCIe altogether since the GitHub repository can be directly cloned in the Linux running on the PS, and the communication to the PL is through the simpler AXI protocol. Another advantage in ZynqParrot is that a desktop is no longer required to run tests on the FPGA since the ARM core on the board can function as the desktop. The bitfiles are sent to the file system of the core over Ethernet. Once the FPGA is programmed with the correct bitstream, the core can run the testbench.

The idea of using the PS to control a processor implemented in the PL has been explored earlier in [82], where the PS communicates with a dual-core RISC-V processor in the PL. Although the structure of this system is similar, there are some key differences between this system and ZynqParrot. Firstly, in [82], all communication between the PS and the PL is through shared DRAM addresses.

As will be shown in Section 3.2.2.2, this is not ideal in the BlackParrot setting since i) messages from BlackParrot can cause race conditions with the PS and ii) BlackParrot requires direct connections with the PS. Further, constant polling of the shared memory locations by the PS could lead to contention and a potential performance implication. Secondly, the RISC-V cores in [82] are not capable of running an operating system on them, whereas the goal in ZynqParrot is to enable a Linux boot using BlackParrot on the FPGA. This requires a different approach to the interfaces between the PS and the PL.

3.2.2.1 Choosing the FPGA

The two requirements for choosing the FPGA were the availability of the Zynq PS and a cost between \$100 and \$150 so that the design could reach more users. The PYNQ Z-2 FPGA [29] is a board designed to support the Python Productivity for Zynq (PYNQ) framework and fit these requirements well. The PYNQ [22] framework is an opensource Xilinx project that allows users of the Zynq SoC the opportunity to run their programs in Jupyter Notebook and interact with the hardware in the PL. Although the ZynqParrot infrastructure described in this report used this FPGA, the underlying design would work for any Zynq-based FPGA. Table 8 shows the BlackParrot utilization results (both raw and percentage) for this part (*xc7z020c1g400-1*) for both the default (32 KB L1 caches and a 64 KB L2 cache) and small (8 KB L1 caches and a 64 KB L2 cache) BlackParrot configurations. Both these utilization numbers use the BlackParrot state at [this](https://github.com/black-parrot/blackparrot/tree/f5cbac361c8099b18b9b665c00bc02f691b654d4) commit (<https://github.com/black-parrot/blackparrot/tree/f5cbac361c8099b18b9b665c00bc02f691b654d4>). The main difference between these two results is the amount of BRAM used since the L1 caches are four times smaller in the second design.

Table 8: FPGA Utilization for Two BlackParrot Configurations*BlackParrot is continuously evolving so this is a snapshot in time*

Configuration	Logic LUTs	LUTRAMs	FFs	RAMB36	RAMB18	DSP48
Default	30,796	1,360	12,945	6	145	11
Small	23,899	1,102	10,234	6	85	11
Default	57.89%	7.82%	12.17%	4.29%	51.79%	5%
Small	44.92%	6.33%	9.62%	4.29%	30.36%	5%

3.2.2.2 Choosing the Connections to use BlackParrot's External Interfaces. BlackParrot Exposes Three Interfaces to External Logic:

- **Outgoing port:** BlackParrot sends a command to the external logic and expects a response. This port is used when BlackParrot sends a read or write command to the UART or denotes that the program has finished.
- **Incoming port:** BlackParrot receives a command from the external logic and provides a response after processing it. This port is used to set the configuration registers, load the program to the DRAM through the L2 cache, and send user input to BlackParrot for a UART read command.
- **DRAM port:** BlackParrot sends a read or write request to the DRAM and waits for the response.

These ports can be used either with BlackParrot-specific interfaces or as AXI ports. The outgoing and incoming ports connect to I/O <-> AXI converters and become AXI4-Lite ports, and the DRAM output port connects to the L2 cache packet <-> AXI converter and becomes an AXI4 port.

Zynq. Figure 18 shows the Zynq PS. There are two General Purpose AXI ports (denoted in red at the bottom left of the figure) with PS as the master and PL as the slave, and two with PL as the master and PS as the slave. Further, four High-Performance AXI ports (denoted in green at the bottom right) with 64-bit data connect to the DDR3 controller.

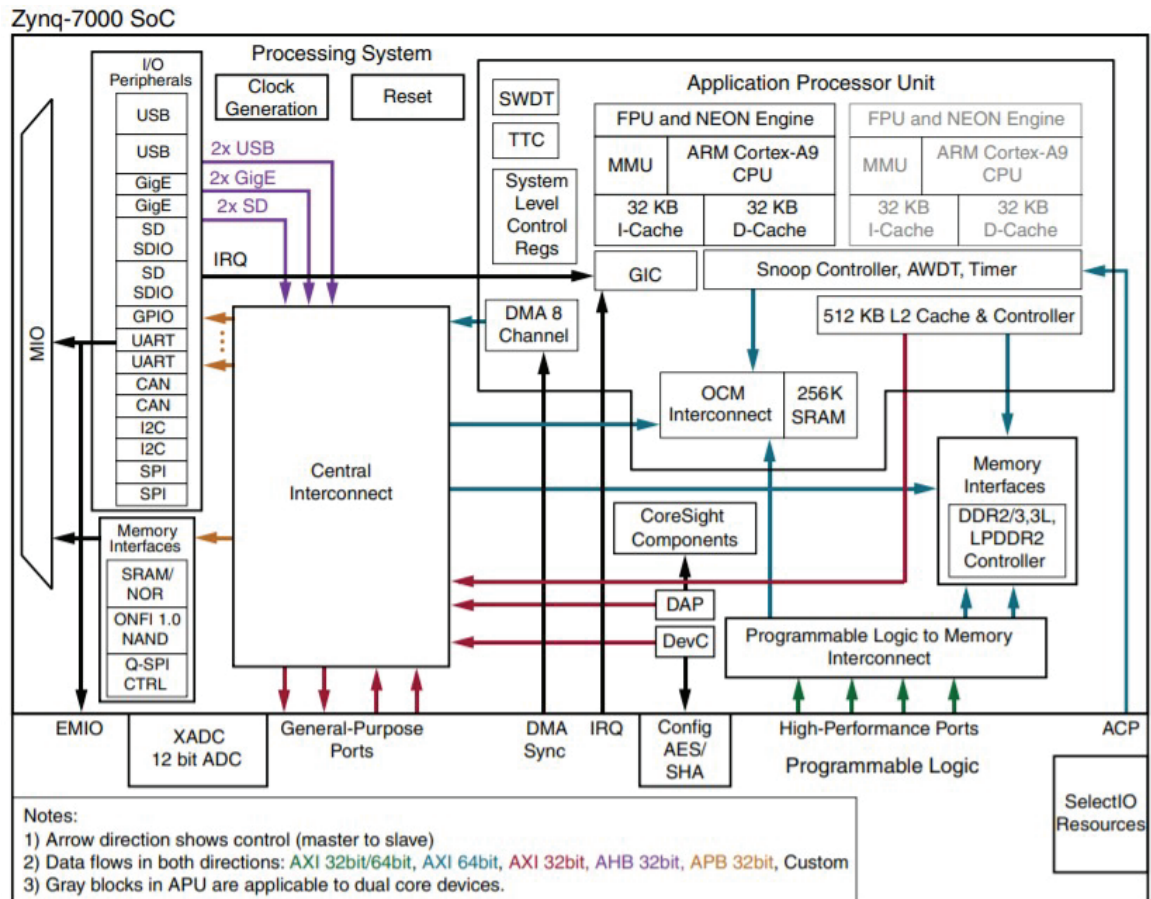


Figure 18: Zynq-7000 SoC Block Diagram [93]

Figure 19 shows the PS address map. Based on the settings used, the Zynq PS ports map to the following physical addresses:

- DDR: 0x00000000 - 0x3FFFFFFF
- M_AXI_GP0 - 0x40000000 - 0x7FFFFFFF
- M_AXI_GP1 - 0x80000000 - 0xBFFFFFFF

The PS allows the user to allocate DRAM space for the PL to access. The allocation returns a virtual and physical address corresponding to the start of the allocated region. The PS can use the virtual address for access and send the physical address to the PL for correct address translation.

AXI4-Lite Protocol. The AMBA AXI protocol supports high-performance, high-frequency system designs for communication between Manager (master) and Subordinate (slave) components.

Address Range	CPUs and ACP	AXI_HP	Other Bus Masters ⁽¹⁾	Notes
0000_0000 to 0003_FFFF ⁽²⁾	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU ⁽³⁾
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFE_FFFF ⁽⁴⁾	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF ⁽²⁾	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

Figure 19: Zynq-7000 System-Level Address Map [93]

It has separate read and write address and data channels. The AXI4-Lite protocol is a subset of the full AXI4 protocol, with all transactions having a burst size of 1 and all data accesses using the full width of the data bus (32-bits or 64-bits). The specification [33] contains in-depth information about the AXI4 protocol signals and timing diagrams.

Identifying connectable interfaces. The BlackParrot AXI interface would be the best choice for the ZynqParrot system since the PS communicates using the AXI3 protocol. The following connections are immediately apparent:

- PS M_AXI_GP1 -> BlackParrot incoming port AXI. The PS port follows AXI3, and BlackParrot follows AXI4-Lite, requiring a converter between them.
- BlackParrot DRAM port AXI4 -> PS S_AXI_HP0. The HP port allows 64-bit data, the minimum data width supported by the BlackParrot L2 cache. BlackParrot follows AXI4, and the PS port follows AXI3, requiring a converter between them.

The converter used in this system is the Xilinx AXI SmartConnect [91]. This IP allows multiple master AXI devices to communicate with multiple slave AXI devices while performing the required protocol conversion. For the first connection mentioned above, only the protocol conversion is necessary since it is a 1:1 connection.

The remaining connection is the outgoing port of BlackParrot. There are two options here:

- Connect the BlackParrot port to the S_AXI_GP0 port and map the connection to the DDR or OCM
- Use a FIFO in the PL that can be written by the BlackParrot port and read by the PS M_AXI port

The first option seems to be the most intuitive since the BlackParrot outgoing port is a master AXI interface, and the PS S_AXI port is a slave interface. However, even if the address mapping is such that the data goes to the DDR or the OCM, there are two issues.

- BlackParrot currently sends a UART write request to a single UART address (0x101000). If the requests go to a single DRAM location, there is a possibility of missed reads by the PS due to race conditions. The program working would depend on the relative latency between BlackParrot sending a request and the PS reading it, which is not acceptable.
- Even if BlackParrot is modified to send UART requests to consecutive locations, it is limited by the DRAM space allocated for this purpose. Further, it has no way of knowing if the PS read the data at older locations before overwriting them.

The second option is not apparent at first but offers an easy way for BlackParrot to guarantee that the PS receives every request. Every request is converted into an AXI write and packed into the 32-bit data, indicating the address and data (for a UART write). If the FIFO is full, BlackParrot stalls until space is available. If the FIFO is empty, there could be additional logic to send garbage data so that the PS AXI request does not stall. This FIFO is designed using the AXI Peripheral creation option provided by Vivado and borrowing the FIFO design from BaseJump STL [81]. It is connected to the M_AXI_GP0 port of the PS and the BlackParrot outgoing port using the AXI SmartConnect.

Additionally, the PS writes a base address register to communicate the physical address of the allocated DRAM to the PL logic. An AXI GPIO [89] module supplies the data written by the PS in its register to the PL.

The connections described above are used in Version 1 of the ZynqParrot design.

3.2.2.3 Version 1

Figure 20 shows the Version 1 ZynqParrot design. All the connections mentioned previously appear in the figure along with their protocol type. Before connecting BlackParrot as the Design Under Test (DUT), a trivial module was added as the DUT to test the interfaces. This DUT contained the following:

- AXI4-Lite BRAM Controller connected to a Block RAM module (to test the direct PS to BlackParrot connection).
- Loopback connection to the DRAM from PS M_AXI_GP1 port to S_AXI_HP0 port (to test the BlackParrot to DRAM connection).
- Simple FSM with AXI4-Lite master interface generating the sentence "Hello from PL" (to test the BlackParrot to FIFO connection).

A Python code using the PYNQ-provided APIs tested the connections on the FPGA and confirmed their working. However, this design was highly Xilinx IP-centric, potentially making debugging more difficult. The ideal setup would allow for simulation of the DUT using an open-source simulator such as Verilator [30] with an emulated PS AXI port interface before testing the design on the board. The PS AXI port emulation would provide confidence to the users that a design working in simulation is likely to work on the board, as long as the address mappings are correct. Further, BlackParrot is tested in Verilator so it would be a better option than Vivado simulation.

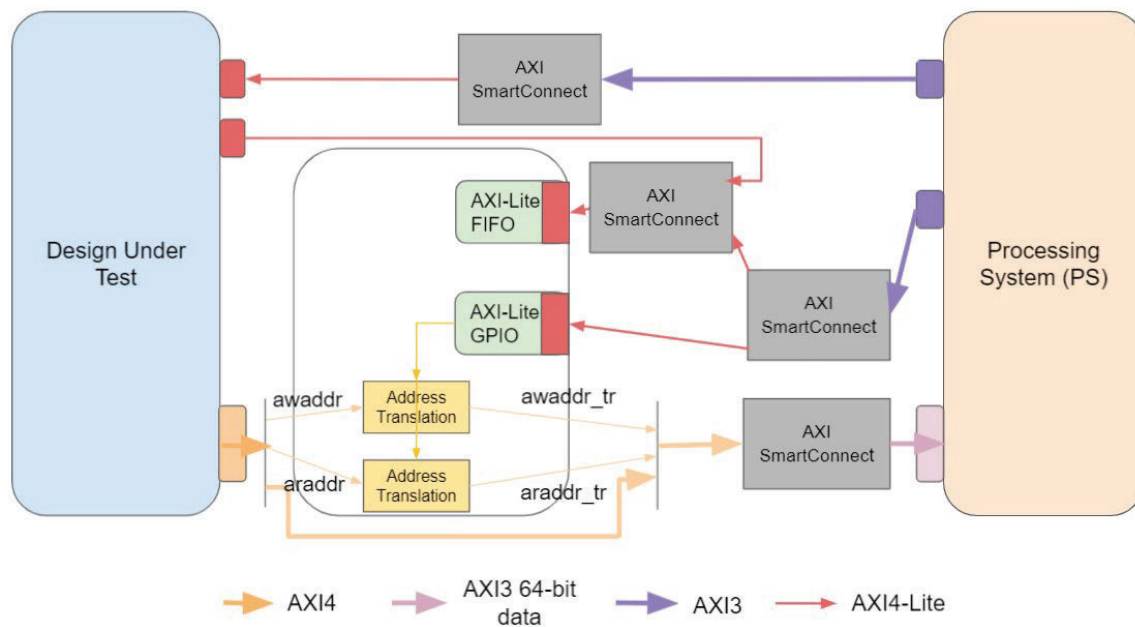


Figure 20: Version 1 ZynqParrot Design

These issues prompted further experiments to identify a better setup, resulting in Version 2 of the ZynqParrot design.

3.2.2.4 Version 2

Figure 21 shows the Version 2 design for ZynqParrot. The PL modules reside in a single Vivado IP Integrator (IPI) block. The changes from the above design are as follows:

- The AXI SmartConnects are limited only for converting the PS AXI3 connections to BlackParrot's AXI4-Lite and AXI4 connections.

- Instead of using Xilinx IPs for GPIO and FIFO modules, a decoder module contains the FIFO with both ends connected to an AXI4-Lite interface, the FIFO occupied count, and offset registers.

The address translations for the DRAM remain the same but are not shown explicitly in the figure. By including the decoder module, the only Xilinx IPs used (apart from the PS) are the three AXI SmartConnects. The fourth SmartConnect, which arbitrated between the PS and the DUT for FIFO access, is replaced by creating a FIFO with AXI4-Lite interfaces on both the input and output sides. This design provides an opportunity to design the PL modules, test them entirely in Verilator simulation before adding the SmartConnect modules and the PS for implementation on the FPGA.

3.2.2.4.1 AXI4-Lite Decoder.

The decoder module allows users to add a parameterizable number of Control and Status Registers (CSRs) to which the PS can write address offsets, the above-mentioned FIFO, and its corresponding flow counter, which is a BaseJump STL module that provides the number of elements currently in the FIFO. It has one AXI4-Lite interface, and the AXI write and read addresses are decoded into one-hot signals, which are used as the select signals for a one-hot multiplexer, all BaseJump STL modules. The PS can read and write the registers, whereas it can only read the FIFO since the DUT writes to it.

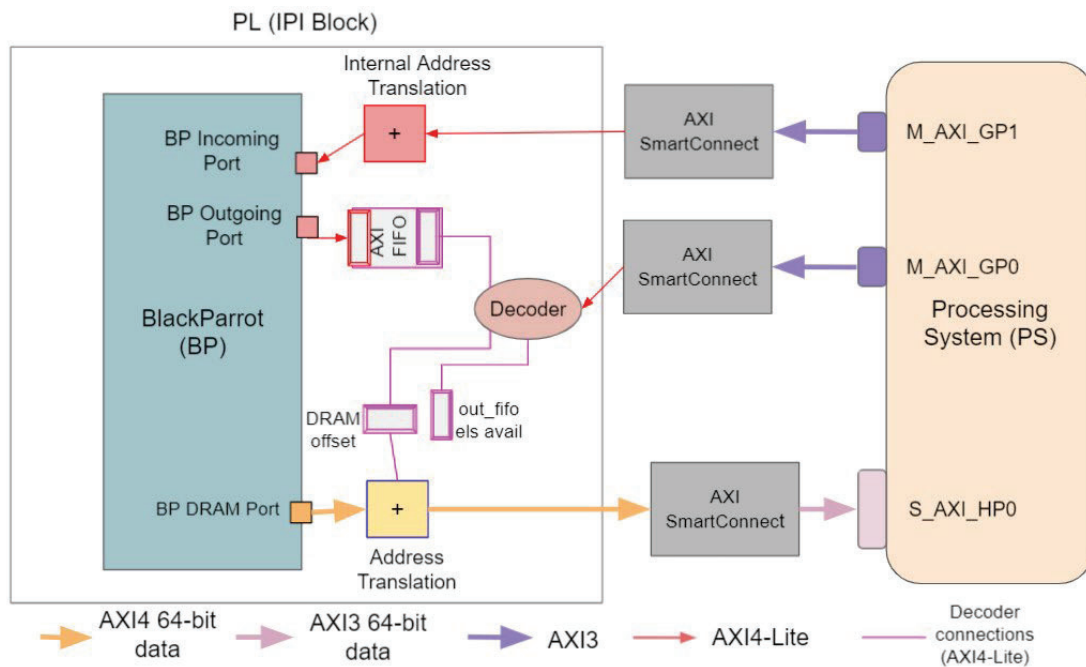


Figure 21: Version 2 ZynqParrot Design

3.2.2.4.2 FIFO with AXI4-Lite interfaces on both ends.

This FIFO allows an AXI4-Lite master to write to it and an AXI4-Lite master to read from it. The difference between this FIFO and the Xilinx custom IP earlier is that there are two different

AXI interfaces in this module. Therefore, arbitration is not required, reducing the number of modules for debugging. On the input side, the AXI master can only write to the FIFO. However, if the master does read, zero is returned as the data without hanging. On the output side, the AXI master can only read from the FIFO.

However, if the master does write, the write is dropped without hanging.

The counter works in conjunction with the FIFO. When an element is enqueued, the counter is incremented. Similarly, when an element is dequeued, the counter is decremented. The PS should read the count at regular intervals and only read the number of elements from the FIFO as specified by the count.

3.2.2.4.3 BlackParrot Address Space.

BlackParrot uses a 40-bit physical address split as follows:

- 0x0000000000 - 0x007FFFFFFF: Local addresses (<7-bit tile ID, 4-bit device ID, 20-bit device address space>. An example device is the UART.
- 0x0080000000 - 0x0FFFFFFFFF: Cached DRAM addresses.
- 0x1000000000 - 0x1FFFFFFFFF: On-chip streaming accelerator address space.
- 0x2000000000 - 0xFFFFFFFF: Off-chip addresses

For the ZynqParrot design, the BlackParrot local addresses and cached DRAM addresses are supplied by the same PS M_AXI port to avoid performing an extra MUX operation (required if two different AXI ports supply the data for the two regions). An equal distribution would ideally allow 512 MB for the DRAM address space and 512 MB for the local address space, since a single PS M_AXI port can access 1 GB of address space.

The PS runs PetaLinux [90], which requires 64 MB of DRAM space, and the PYNQ Z-2 board contains a DDR3 memory of size 512 MB. BlackParrot can therefore use 256 MB of DRAM space, which is sufficient for most of the standard benchmarks.

Similarly, a 256 MB address space for the local addresses ensures access to 16 tiles since the highest address accessible in this scenario is 0x000FFFFFFFF, which includes 4 bits of the tile ID. Due to hardware constraints of the board, it is improbable to require more than 4 bits for the tile ID.

The two address spaces are combined to map to 512 MB of the PS GP1 port. The address maps are shown in Table 9. An internal address translation is required since the Xilinx tool subtracts the 0x80000000 offset after moving through the GP1 port. This is also indicated in the table.

Table 9: ZynqParrot Address Mapping

Region	Address seen by PS	Address inside PL IPI	Address Translation
DRAM	0x80000000 - 0x8FFFFFFF	0x00000000 - 0x0FFFFFFF	Add 0x80000000
Local	0xA0000000 - 0xAFFFFFFF	0x20000000 - 0x2FFFFFFF	Subtract 0x20000000

3.2.2.4.4 BlackParrot UART Read Issue.

The Version 2 design contains all BlackParrot interface logic except for the UART read request logic. The only path for this request in the design is to the FIFO with AXI4-Lite interfaces at both ends. As already mentioned, the input to this FIFO is write-only, but a read will return zero without hanging. This issue is not immediately apparent since most standard tests used in BlackParrot testing do not involve UART reads. However, consider the scenario where BlackParrot is running Linux and requests user input. The user input would always be zero, without any intervention possible by the user.

Version 3 of the ZynqParrot design fixed this issue using a new module and another FIFO.

3.2.2.5 Version 3

A store packer converts read requests from the BlackParrot outgoing port into write requests that are then queued onto the FIFO. Further, another FIFO is added in the reverse direction that the PS can write, and BlackParrot can read. The PS reads the request in the FIFO, deciphers it, performs the required operation (print to the screen, obtain user input or terminate the program), and sends the user input (if any) to the newly added FIFO. The store packer would contain a small FSM that waits for the user input from the PS if the BlackParrot request was a UART read and responds to BlackParrot once it receives the data.

Earlier, the FIFO enqueueing BlackParrot AXI requests had an AXI4-lite input interface. Now, since all requests are converted into write packets that the PS deciphers, the FIFO can revert to having an output AXI interface and a standard input interface of data and valid.

The entire process is described below. The FIFO enqueueing requests from BlackParrot is called the output FIFO, and the FIFO enqueueing requests from the PS is called the input FIFO.

- BlackParrot sends an I/O request (UART read/write, finish).
- The store packer converts this request into a 32-bit packet with a concatenation of <read/write (1-bit), address (23-bits), data (8-bits)>.
- The 32-bit packet is queued onto the output FIFO.
- The PS reads the packet, decodes it in software, performs the necessary operation, and returns data, if any, to the input FIFO.
- In the event of a read, the store packer responds to BlackParrot with the data.

The decoder module (called the *shell* in this version) contains the two FIFOs, a DRAM base offset register and a newly added reset register that controls the BlackParrot reset signal along with the AXI interface reset. Earlier, the only way BlackParrot could be reset was by reloading the bitstream. This register allows the PS to control the BlackParrot reset through the testbench without having to rely on the interface reset.

Figure 22 shows the version 3 ZynqParrot design. The store packer is denoted as BP r/w to write in the figure.

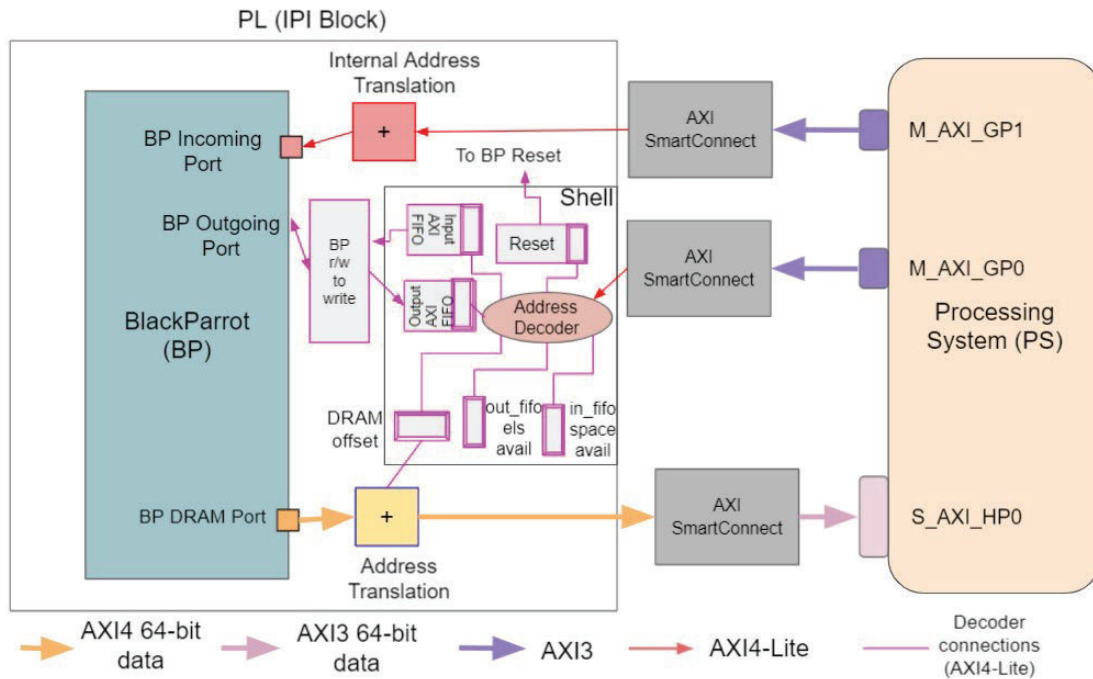


Figure 22: Version 3 ZynqParrot Design

A future version of ZynqParrot would benefit from using the AXI4-Lite outgoing port interface since BlackParrot could easily interface with other accelerators and use a multicore configuration with multiple I/O ports. Figures 23 and 24 show an example design.

3.2.2.6 Software Setup.

The main idea behind cosimulation is to provide users with an option to test their BlackParrot system in simulation by mimicking the PS AXI interface as closely as possible. The strategy is to create a single C++ testbench file and swap the include files based on the environment used. Verilator converts the Verilog modules into a C++ equivalent triggered using the testbench for use in simulation, whereas the same testbench is run on the Linux instance of the ARM core on the FPGA to communicate with the PL.

3.2.2.6.1 Directory Structure.

Figure 25 shows the directory structure in the ZynqParrot repository. As depicted, the same testbench is reused for both the simulation and the hardware by including different header files for each environment.

3.2.2.6.2 Header Files.

The software API provides two header files, one for simulation and one for the FPGA. These two files contain a class with largely the same methods so that the testbench can swap them based on the environment. The common methods between the two files are as follows:

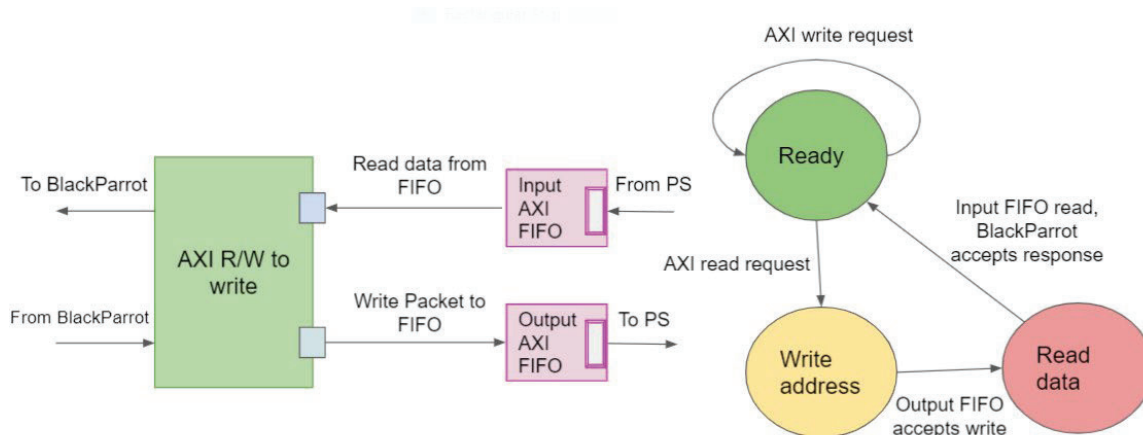


Figure 23: AXI R/W to Write Module and FSM

- *axil_write*: This method performs a write operation over an AXI port. In the simulation environment, the actual AXI4-Lite functionality is emulated in the function, whereas in the FPGA environment, it essentially becomes a write to a pointer that signifies the specific AXI port.
- *axil_read*: This method performs a read operation over an AXI port. In the simulation environment, the actual AXI4-Lite functionality is emulated in the function, whereas in the FPGA environment, it essentially becomes a read from a pointer that signifies the specific AXI port.
- *done*: This method signifies the end of testing.

There are a few additional methods and operations in the FPGA environment:

- *allocate_dram*: The PS needs to allocate the required amount of DRAM for the PL to access. This method performs the allocation and returns the virtual and physical pointer to the first DRAM location. Experiments with different DRAM sizes found that 64 MB works well. A plan is to try and extend this size since some programs need more than 64 MB of DRAM.

- *free_dram*: This method frees the allocated DRAM and should be called at the end of testing.

3.2.2.6.3 Testbench.

The testbench for BlackParrot instantiates an object of the class described above and performs two operations: set the configuration registers inside BlackParrot and write the program to DRAM through the BlackParrot L2 cache. Once these operations are complete, the testbench waits for data to arrive at the output FIFO and decodes the data to perform the correct operation. The two functions specific to the BlackParrot testbench are described below:

- *nbf_load*: This function sets the configuration registers and loads the program into the DRAM by using a .nbf (Network Boot Format) file available in BlackParrot. The nbf file contains a packet with the address and data in each line. This function reads the file line by line, extracts the address and data, and calls the *axil_write* method of the object with these values.

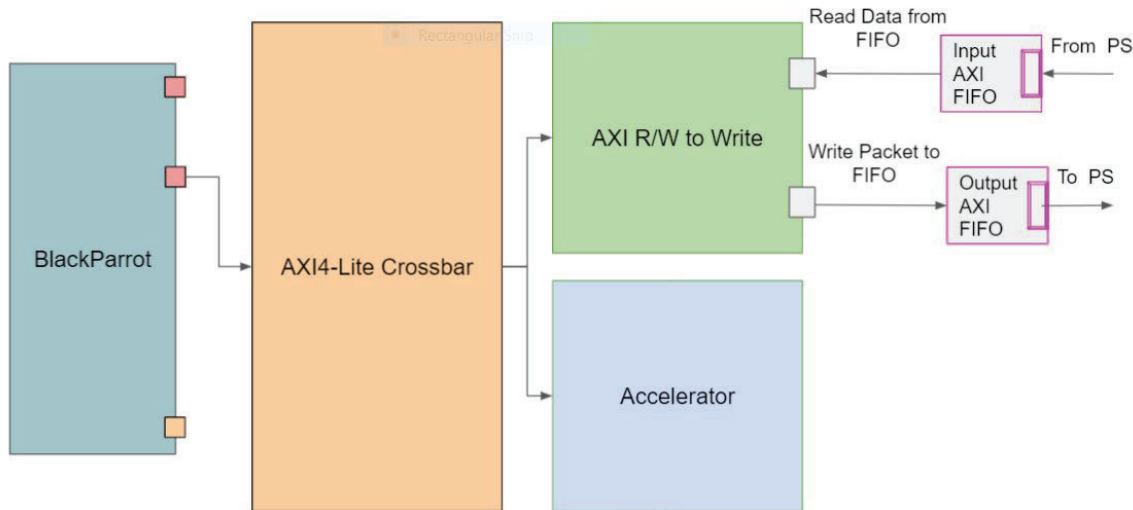


Figure 24: Future Extension to ZynqParrot

- *decode_bp_output*: This function receives 32-bit data and decodes the contents. As mentioned in the hardware section, a packet sent by BlackParrot is a concatenation of <R/W, 23-bit address, 8-bit data>. If the packet is a UART write (R/W = 1 and address = 0x101000), the data is printed onto the screen. If the packet indicates a finish (R/W=0 and Address = 0x102000), the program is terminated. UART reads are currently not supported in this function but can be added by asking for user input and writing the result to the input FIFO.

The testbench also contains multiple diagnostic tests aimed at testing each interface connection in both simulation and actual hardware. The following tests are performed before BlackParrot is initiated:

- Writing and reading the CSRs in the decoder (tests the GP0 port connection).

- Writing to enough addresses in the DRAM through BlackParrot to trigger evictions from the L2 cache and reading these addresses using the virtual addresses in the PS space (on hardware) or through BlackParrot (in simulation) (tests the GP1 port connection for DRAM addresses and HP0 port connection).
- Reading from the *mtime* and *mtimecmp* registers (machine timer registers) inside BlackParrot (tests the GP1 connection for local addresses).

3.2.2.7 Validation and Results

This section provides the numbers and the status as of [this](#) commit. It is important to note that ZynqParrot is a continuously evolving system, so these results are valid only at this point in the history. The testing of this system involved both simulations using Verilator and running the testbench on the PYNQ-Z2 FPGA. Table 10 shows the utilization results on the PYNQ-Z2 for the ZynqParrot system with the default BlackParrot configuration containing 32 KB L1 caches and a 64 KB L2 cache. Table 11 shows the hierarchical utilization of ZynqParrot with some of the major components of the system displayed. The basic “hello_world” program provided the initial confirmation that the system worked as expected. The next step involved running several SPEC2000 and SPEC2006 benchmarks [26] on the FPGA. The *beebz* benchmarks [1] were also run as a regression on the board (all tests run with a single command) with the help of the reset register in the shell module and confirmed its working.

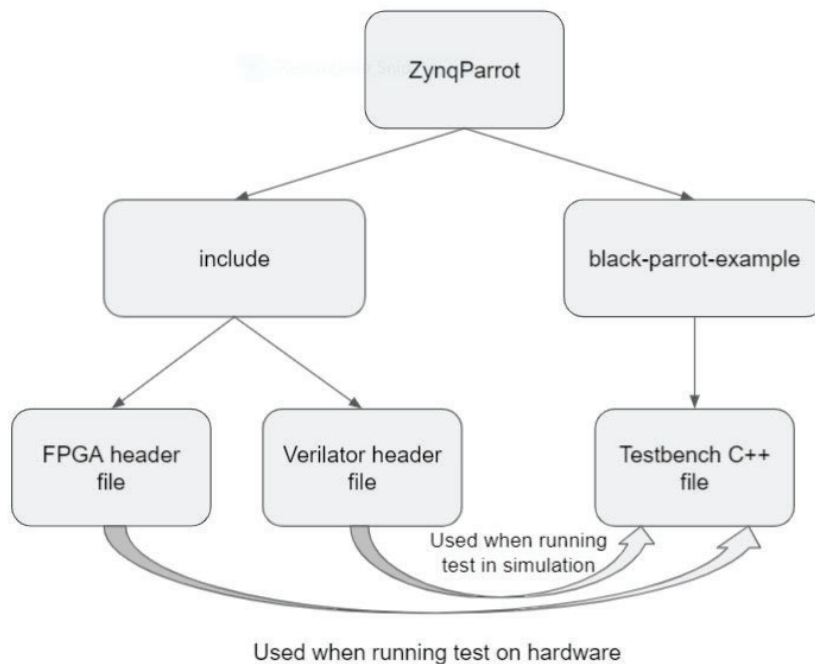


Figure 25: Testing Directory Setup

Table 10: FPGA Utilization for the ZynqParrot System with the Default BlackParrot Configuration

FPGA Component	Utilization	Percent utilization
Logic LUTs	31,917	59.99%
LUTRAMs	1,652	9.49%
SRLs	433	2.49%
FFs	16,586	15.59%
RAMB36	6	4.29%
RAMB18	145	51.79%
DSP48	11	5.00%

Tables 12 and 13 show the instructions, cycles and Instructions per Cycle (IPC) values for the SPEC2000 and SPEC2006 benchmarks running on ZynqParrot. The FPGA emulation speed is around one billion BlackParrot cycles per minute, whereas the Verilator simulation speed is around 100,000 cycles per minute. The numbers in the table show that the longest running benchmark (464.h264ref) would take about 200 minutes to complete on the FPGA, whereas even the smallest benchmark in terms of cycles (175.vpr) would take around 270 minutes to simulate using Verilator. ZynqParrot enables running this new class of tests that is impossible to do in simulation.

Table 11: ZynqParrot Hierarchical Utilization

Component	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP48
ZynqParrot	69,333	3,242	128	50,465	37	30	12
D\$	3,751	191	0	2,117	0	64	0
BP-L2 converter	3,340	384	0	614	0	0	0
SmartConnects	2,933	224	338	3,842	0	0	0
L2 cache	1,797	454	0	1,987	0	64	0
I\$	1,476	183	0	1,373	0	16	0
FMA	1,256	0	71	165	0	0	11
Regfiles	763	0	0	492	5	0	0

Table 12: SPEC2000 Benchmarks Running on ZynqParrot

SPEC2000 Benchmark	Instructions Retired	Cycles	IPC
164.gzip	1,234,135,782	1,968,599,120	0.627
175.vpr	23,452,462	27,906,352	0.840
177.mesa	437,214,657	562,967,056	0.777
183.equake	526,995,458	841,464,792	0.626
186.crafty	4,142,052,045	5,172,257,264	0.801
188.ammmp	38,813,804	72,238,528	0.537
197.parser	227,585,917	289,582,784	0.786
256.bzip2	1,592,049,924	2,205,371,304	0.722
300.twolf	97,342,911	182,005,544	0.535

Table 13: SPEC2006 Benchmarks Running on ZynqParrot

SPEC2006 Benchmark	Instructions Retired	Cycles	IPC
401.bzip2	3,175,874,105	4,210,062,456	0.754
410.bwaves	20,684,252,488	48,997,476,110	0.422
444.namd	64,051,683,912	104,703,424,500	0.612
445.gobmk	416,632,992	1,062,763,112	0.392
462.libquantum	176,152,994	203,135,528	0.867
464.h264ref	142,762,531,949	200,478,638,500	0.712
473.astar	23,673,222,279	42,845,912,910	0.553

3.3 OpenSource Integration

Recent developments in architecture research warrant the need for efficient host cores to interact and manage multiple accelerators in a system-on-chip design. Existing designs suffer from low configurability, rely on non-standard or proprietary tooling, or require sophisticated mechanisms to interact with accelerators. One of the critical requirements to achieve this goal is modifying the core quickly to suit the application’s needs. The work described in this section of the report discusses ways to optimize BlackParrot for integration with diverse architectures. The creation of the BlackParrot unicore and the standard cache interface between the L1 cache and its controller allows BlackParrot to integrate with designs with minimal modification. Its parameterizable cache, along with a multi-cycle fill strategy, has enabled the creation of a tiny core that can find uses in systems with physical limitations. The utility of these modifications, among others, was validated by integrating BlackParrot with HammerBlade [13] and OpenPiton [38].

3.3.1 Introduction

The end of Dennard Scaling [9] and the ‘Utilization Wall’ have forced computer architects to explore domain-specific architectures [79]. Apple’s M1 processor features multiple accelerators for graphics, security, and machine learning on the same chip. Google’s TPU [58] accelerates machine learning workloads and Microsoft uses FPGAs (Project Catapult) to accelerate cloud-

based services and AI. The academic and research communities have many examples showing the growth of hardware specialization [36,49,54,56,63,68,69].

The growing popularity of accelerator SoCs requires host processor cores optimized for interacting with and managing multiple accelerators. Current commercial cores provide interfaces that are not configurable for the needs of the user and require elaborate mechanisms to communicate with an accelerator which adds overhead to the system's performance. Additionally, they are expensive and inaccessible for research and open-source communities.

While [66] shows that BlackParrot provides the best-in-class performance, area, and energy efficiency in comparison to popular open-source and commercial RISC-V cores [34,52,95], its configurability was limited. BlackParrot's multicore system offered the flexibility to add and remove customized tiles for memory, accelerator, and I/O to the network as long as they adhered to the network protocol. This flexibility was, however, only at the system level. The aim was to enable the core itself to be flexible. There are multiple academic, industrial, and open-source research projects that require a core to connect to a pre-existing network. Other projects have restrictions on on-chip (silicon/FPGA) resources for a host core. In both cases, the lack of core-level configurability resulted in resorting to solutions that motivated the development of an accelerator host core in the first place (i.e. using commercial cores with sophisticated mechanisms to interact with the accelerator).

Here the report addresses BlackParrot's configurability at the core level by motivating the need for an efficient uncore and present examples of using these developments for the integration of BlackParrot in other open-source projects. This section of the report covers:

- A size-configurable core, codenamed TinyParrot verified to synthesize using an open-source 45nm process node and a low-cost FPGA. BlackParrot's size configurability creates an ecosystem for a broader base of users to include on-chip accelerator host cores in their designs.
- The integration of BlackParrot into the OpenPiton memory system and its FPGA validation.
- An accelerator host core codenamed HammerParrot for a tiled manycore architecture specialized for machine learning and graph analytics where BlackParrot's tininess and flexibility improves overall system performance.

3.3.2 Background

This section describes the major components used in this section of the report.

3.3.2.1 OpenPiton OpenPiton

[18,39] is an open-source multithreaded manycore processor. It is a tiled design that uses the OpenSPARC T1 core with custom uncore components (such as the caches, cache coherence protocol, 2-D mesh Network-on-Chip NoC). The 2-D mesh design and Coherence Domain

Restriction [50] allow OpenPiton to scale up to 500 million cores. The OpenPiton design is shown in Figure 26 (reproduced from original work [39]).

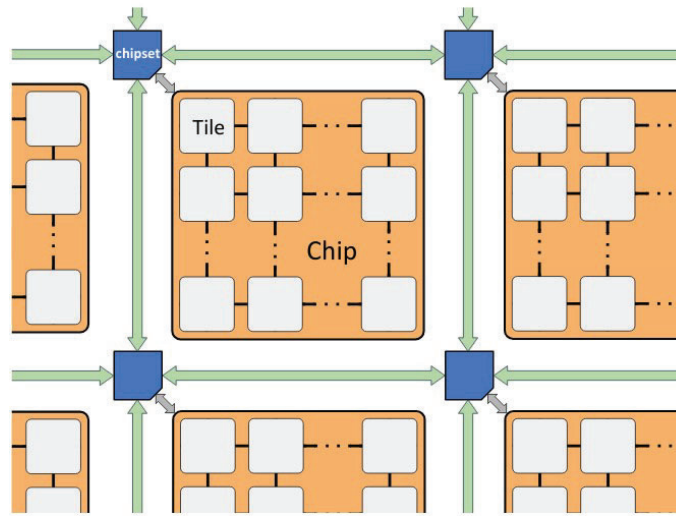


Figure 26: OpenPiton Architecture [39]

Multiple units, each containing a 2-D mesh of tiles, connect to chipset logic through a network. The chipset contains off-chip components such as DDR memory, Ethernet, SD Card, and UART.

Each tile contains an OpenSPARC T1 core, an L1.5 Cache, an L2 Cache, a Floating Point Unit (FPU), arbiters, and routers. The L1.5 cache is a write-back cache between the core's L1 cache and the shared L2 cache of OpenPiton Network. There were two reasons for the development of this cache. Firstly, the OpenSPARC core utilized a write-through L1 cache tightly integrated with its pipeline. Write-through caches are a source of bottlenecks in distributed cache frameworks since they have a very high bandwidth requirement to the next level in the cache hierarchy. Secondly, the cache requests and communication had to be compliant with the OpenPiton cache coherence network interface. Rather than modify the L1 cache in the core to overcome the above two problems, the OpenPiton developers decided to include a private L1.5 cache for each core. This write-back cache avoided overwhelming the memory system with messages and allowed OpenPiton to connect the core easily with the cache coherence network.

The L2 cache is a distributed cache shared across all tiles and is inclusive of both private caches (L1 and L1.5). The L2 is the point of coherence in the memory system, and coherence is maintained using a directory-based MESI protocol.

3.3.2.2 Bring Your Own Core (BYOC)

BYOC [7, 37] is an extension of the OpenPiton framework, which allows different cores to connect to the OpenPiton memory system. This enables heterogeneous ISA research and provides a platform for direct comparison of cores using different ISAs. Figure 27 shows an example BYOC system (reproduced from original work [37]).

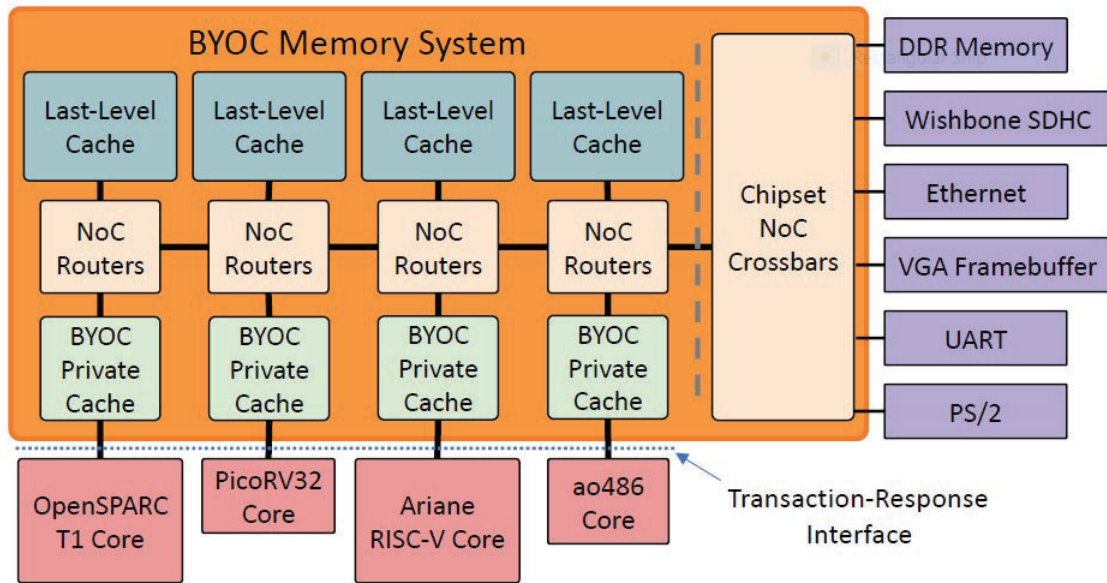


Figure 27: BYOC System Architecture [37]

The standard cache coherence system provided to all the connected cores is called P-Mesh. The memory system contains the BYOC Private Cache (BPC), three NoC networks, and a portion of the distributed LLC. The chipset allows connections to peripherals like DDR Memory, Ethernet, UART, SD card, and accelerators.

Since this system should cater to different cores connecting to it, one question is where to create the interface point. With the assumption that the BYOC framework provides a distributed LLC, the developers compare three different interface points. The first option is to provide the entire cache hierarchy, including the L1 cache, to the cores. This option is impractical since the L1 cache provided would have to account for all the variants, such as virtual or physical tagging and indexing. Further, since L1 caches are usually tightly integrated with the core, requiring the core to interface with another (general) cache could have performance implications. The second option is to provide only the LLC to the cores. This option places a burden on the user connecting a core to the system by requiring them to understand many details of the coherence protocol and incorporate them in their private caches. The third option is to provide a private cache, local to the framework, and expose an interface to the user. In this case, the connecting core does not need to know details about the coherence protocol, and the framework does not need to (itself) support different types of caches. The developers chose to implement this option for its benefits over the other two. Figure 28 illustrates the three options (reproduced from original work [37]).

The Transaction Response Interface (TRI) provides a uniform memory system interface to all the cores. The different cores need to design a transducer that converts the core-specific signals to those understood by the interface to connect to the system. However, the core must have a write-through data cache, resulting from the choice to implement a simple TRI that avoids the extra write-back operation upon invalidation of a dirty line in the core's L1 cache. The L1 cache thus becomes transparent to the BPC.

The BPC is a write-back cache to avoid excessive amounts of data flow to the next level. The BPC stores only data from the L1 D-Cache, with the L1 I-Cache requests bypassed to the next level.

It has to be inclusive of the D-Cache so that all lines in L1 are accessible for invalidation. The BPC tracks the L1 way information using a Way Map Table (WMT). The WMT allows flexibility in the BPC eviction policy while sending invalidations to the correct way in the L1 cache. The BPC currently uses a fixed instruction cacheline size of 32 bytes and a data cacheline size of 16 bytes to support the OpenSparc T1 core. Configurable cacheline sizes for the BPC are a work in progress.

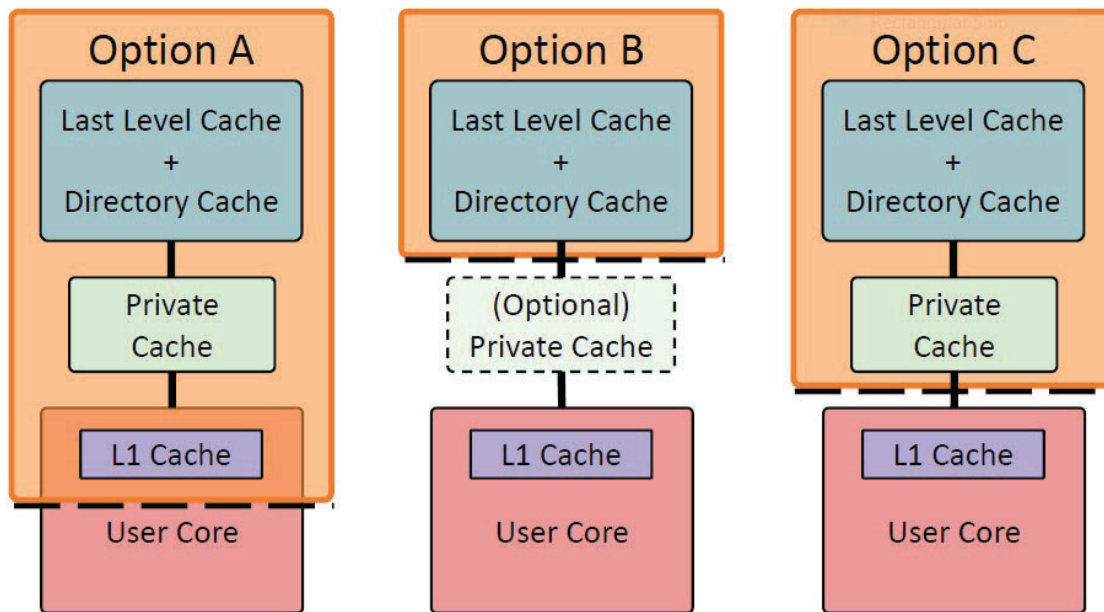


Figure 28: Options for Interfacing the Memory System with the Core [37]

3.3.3 TinyParrot

This section addresses the issue of having an accelerator core for a system despite many pre-existing physical limitations. The efficient uncore(Section 3.3.3.1) strips away unnecessary multicore components and saves area. A study was conducted to investigate the area occupied by the different components of the core. BlackParrot was synthesized for an ASIC process node and an FPGA. The area reports and utilization numbers revealed that the cache SRAMs and the downstream buffering in the default BlackParrot configuration contributed significantly to the area.

This problem was addressed by switching to smaller cache organizations and exploiting the iterative filling/eviction with sub-banking(Sections 3.3.3.2, 3.3.3.3). With these area optimizations in place, BlackParrot can scale to smaller sizes, thereby expanding the range of systems that can use BlackParrot as a host core. The following sections present the synthesis

results for a 45nm ASIC process node and an FPGA, along with any challenges that had to be overcome in this exercise.

3.3.3.1 Motivation for the BlackParrot Unicorn

As stated previously, BlackParrot is an efficient multicore with system-level configurability that supports the integration of coherent and non-coherent, standalone accelerators. Figure 29 shows a single-core tile in the BlackParrot multicore system.

However, not only system-level configurability is enough, but also core-level configurability is necessary to expand the scope of BlackParrot as an accelerator host and cater to more full-featured systems that require host core(s) to connect to a network. The first roadblock in this path was the lack of an efficient single-core implementation of BlackParrot.

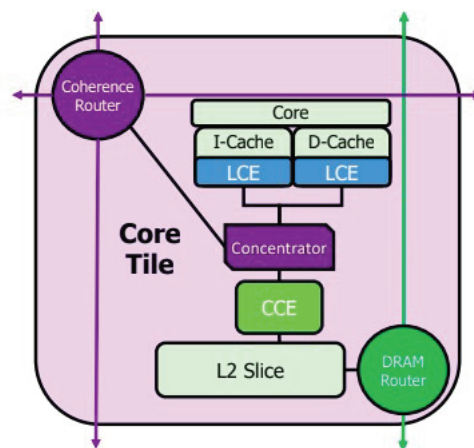


Figure 29: A Core Tile in the BlackParrot Multicore System

LCEs are the Local Cache Engines and a CCE is the directory-based Cache Coherence Engine [66].

A study that measured the area of BlackParrot's core tile (The snapshot of BlackParrot used for this experiment is [commit: c8c67bedb5](#)) exposed the area inefficiency of the core tile for a single core BlackParrot. Figure 30 shows the results of this experiment. The core occupies only 64% of the total area, while the directory controller and the network components consume the rest. These are components that are not required in the single-core implementation. They add to the overall power consumed by the core tile, the cycles incurred during memory operations, and make the system more complex than necessary.

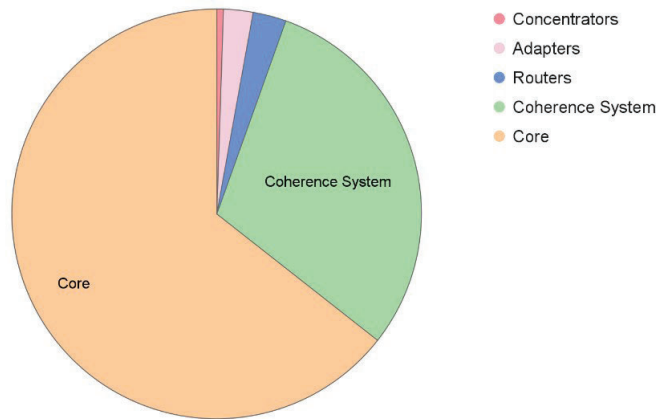


Figure 30: Area Breakdown of a Single BlackParrot Core Tile (no L2 slice)

The area numbers were generated using the Global Foundries 14nm process node. Note: BlackParrot is a **continuously evolving** processor core and the version used here was accessed on Jan 31, 2020.

A uncore system minimally needs the core pipeline, the L1 caches, and their controllers and expose a suitable interface to the rest of the memory system. Additionally, an integration-optimized uncore needs to provide the necessary configurability to communicate with its connected system efficiently. A standard interface between the L1 cache and its controller can achieve the required configurability while satisfying the minimum requirements of the uncore. The following sections describe how to do this in a maximally efficient and modular way.

3.3.3.2 Iterative Filling/Eviction

BlackParrot's default configuration uses an 8-way, 32KB cache with a 64B cache line. Cache line filling and eviction are done at the cache line granularity, meaning that 64B chunks of data are transferred at any point in time. Filling/evicting lines this way was sub-optimal because -

1. It is not practical to expect full cache lines to be available from the lower levels of the memory system because of the network's flit size and wormhole streaming².
2. It occupies silicon die area and requires more energy. The downstream logic in the memory system uses multiple buffers and FIFO queues to handle different data processing rates between modules. Moving 64B chunks of data would therefore require large buffers and FIFO memories.

One of the immediate solutions would be to pick another cache configuration that BlackParrot supports. For example, using a 2-way, 8KB cache can handle data chunks required by systems such as OpenPiton while still moving data around in cache line width. However, this still does not solve the issue with the physical network limitation and decreases the cache's hit rate since it is less associative and smaller in size.

² NoC symbiosis breaks this barrier with some modifications to the system configuration [67]

The solution was to implement an iterative filling strategy. By supporting partial fills in the L1 caches, power consumption and area utilization can be minimized. However, miss latency increases since each fill/eviction incur more cycles. The increase in miss latency can be amortized by implementing critical word first or early restart mechanisms in the cache. BlackParrot supports a rudimentary version of the critical word first mechanism where the controller fills the cache with the requested address first, but the core remains stalled until the miss request is complete.

An additional field, *fill_index*, was introduced into the data memory response packet in the cache interface to implement iterative line filling. This index tells which position in the cache line the fill data is supposed to replace. The fill widths can be a minimum of 8 bytes and a maximum of 64 bytes and can be no smaller than the bank width. The controller can even fill multiple data memory banks at the same time. When a cache request packet is received, the controller first fetches the data for the address requested (i.e., the critical word) and then proceeds to fill all the other sub-blocks within the cache line iteratively. Figure 31 shows the sequence of operations required to fill a single cache line.

3.3.3.3 Sub-bank Filling/Eviction

The iterative filling method described above enforced that the fill width can be a minimum of the bank width. BlackParrot's minimum bank width is 8 bytes since the maximum data width that the core can request is 8 bytes. However, most systems expose interfaces that follow standard protocols such as AXI, AXI-lite, Wishbone, or other pre-existing IP, some of which require 32-bit data buses to transmit and receive data. Additionally, cache designs with no banking shuttle cache lines around which as stated earlier was sub-optimal. To accommodate for the IP requirements/banking strategy while still reaping benefits of iterative filling, sub-banking is introduced.

The iterative filling strategy is extended with minor tweaks to implement sub-bank filling and eviction. The semantics of the *fill_index* field in the data memory packet was modified to indicate a sub-bank index. The cache can then use this index value and the pre-determined fill size to calculate the correct masks to write into the right data memory bank.

BlackParrot's caches currently support the fill widths and bank widths shown in Table 14.

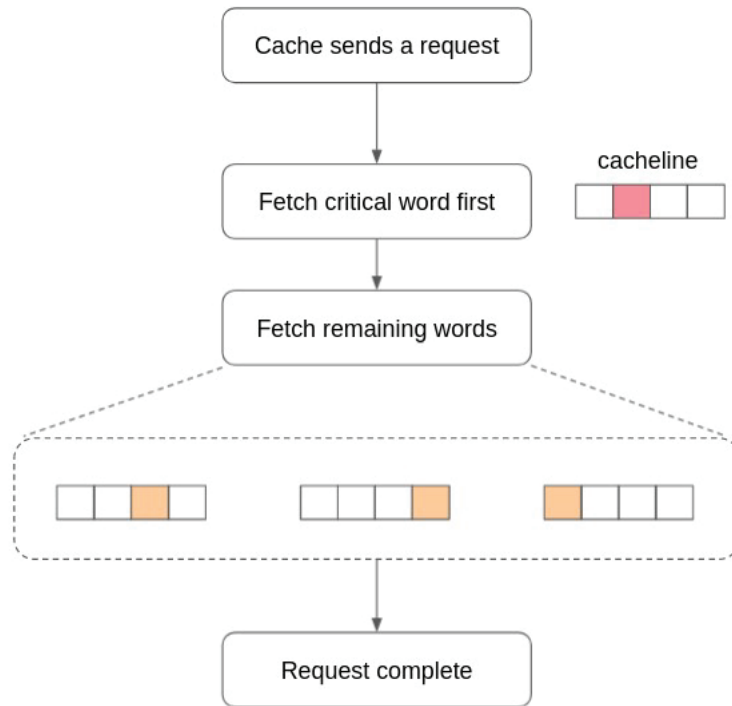


Figure 31: Iterative Filling of a Cacheline

Each colored box in the cacheline indicates the portion of the line that is being filled. The red box indicates the critical word.

Table 14: Bank and Fill Widths Supported

	Widths supported
Bank	64, 128, 256, 512
Fill	64, 128, 256, 512

Supporting a fill width of 32 bits in BlackParrot required some more careful thought. Firstly, BlackParrot’s uncached load and store operations need a maximum of 64 bits. Secondly, many of the peripherals supported by BlackParrot communicate using 64-bit data. Adding Serial-In-Parallel Out (SIPO) modules on the request path and Parallel-In-Serial-Out (PISO) modules on the response path can resolve both the above issues when the fill width is 32-bits. Because almost all of BlackParrot and its peripherals communicate in 64-bits, this would add a reasonably significant overhead to the entire system to support a single bus/accelerator.

Therefore to reconcile the two, BlackParrot will support only a minimum of 64-bit fills, and in order to support 32-bit protocols like AXI-lite, the BP ↔ IP transducer should have PISOs on the request path and SIPOs on the response path. BlackParrot has three Link Protocols - BP Lite

(Similar to AXI-lite), BP Stream (Similar to AXI-Stream), and BP Burst (Similar to AXI-4) to address this. An overview of these protocols can be found in [5].

3.3.3.4 ASIC Synthesis

Bespoke Silicon Group has a well-established ASIC tool flow to convert SystemVerilog/Verilog RTL to GDSII layout. BlackParrot with different cache organizations was synthesized using this tool flow as a backbone and an open-source 45nm library, FreePDK45 [12]. The core BlackParrot RTL along with the necessary changes (a combination of [commit: c945954e7c](#) and [commit: e5723c681a](#)) for the different cache optimizations was merged and synthesized using a pre-existing physical design methodology.

3.3.3.4.1 Challenges

Without any intervention, the synthesis tool converts memories into a 2-D array of flip flops. Flip flops are huge gates requiring almost 20 transistors to realize them. The synthesis process that uses flip flops as memory bit cells would also take a long time to finish with large memories. Therefore, memory macros or black boxes generally replace flip-flop-based memories to enable faster synthesis and occupy lower die area. A RAM generator is a tool that converts a memory specification (depth of memory, width of memory) into a macro, made up of 6 transistor SRAM cells packed together tightly along with any interface logic. The synthesis compiler takes as input these pre-generated memory macros and other RTL to compile the design. The synthesis tool views these memory macros as black boxes, thereby significantly saving compilation time. Additional optimizations from BaseJumpSTL, such as width-banking and depth-banking, helped create smaller, square memories optimized for area. For this case study, a RAM generator from the Bespoke Silicon Group, **bsg_fakeram** was used to generate the different RAMs in the design. However, one limitation of this RAM generator was that it could generate only single-ported memories. BlackParrot's integer register file requires 1 write port and 2 read ports, while its floatingpoint register file requires 1 write port and 3 read ports. BaseJumpSTL modules that realized nread and 1-write memories using 1-read and 1-write memories, partially solve the mismatch in the number of ports required. A new BaseJumpSTL module was created to convert the 1R1W (dual port) memories to a 1RW (single port) memory that this RAM generator could generate. Figure 32 gives an illustration of this design.

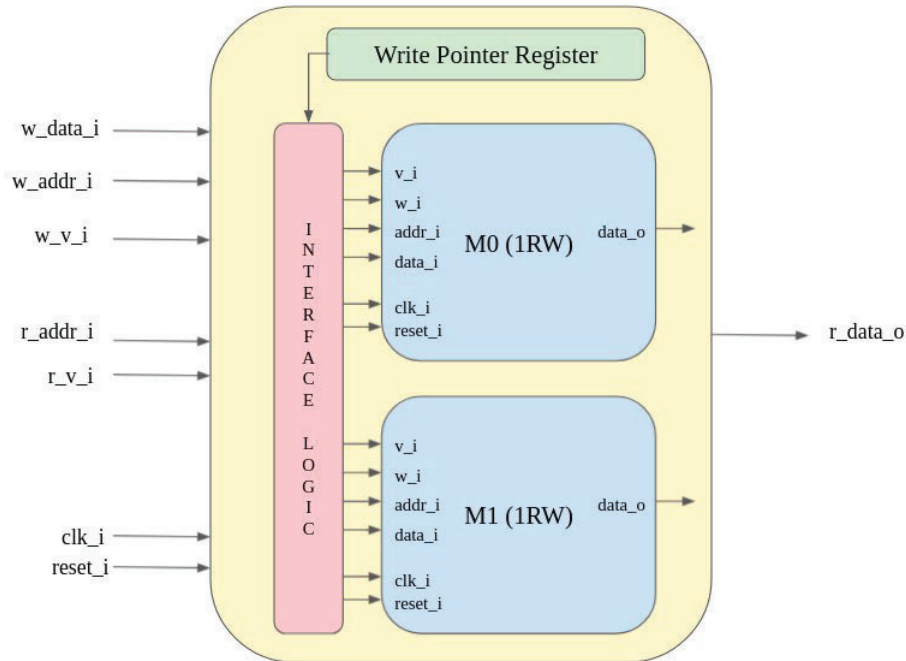


Figure 32: Dual-ported Memories using Single-ported Memories

The key idea here is to use two 1RW memories of the exact specification as the 1R1W memory and maintain an additional register for each element in the memory that tells where the next write should go.

3.3.3.4.2 Synthesis Results

With the necessary design optimizations in place, area runs were conducted, and the results were summarized. Figure 33 shows the area breakdown of a single core BlackParrot, with different cache organizations. For each case, 64-byte, 32-byte, and 16-byte cache lines, respectively, and an 8-byte fill width were used.

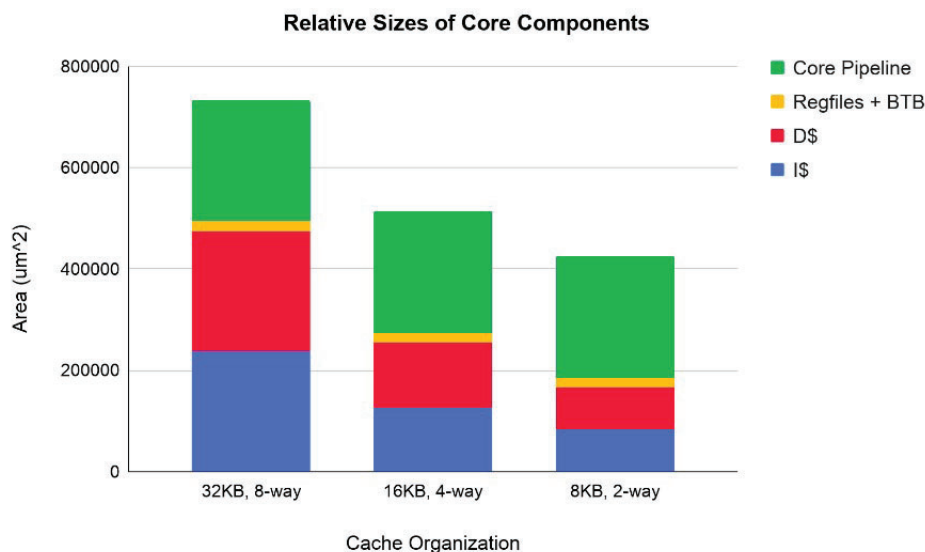


Figure 33: Area Breakdown of BlackParrot with Different cache Organizations in the FreePDK 45nm Process Node

Note: BlackParrot is a **continuously evolving** processor core and the version used for this assessment was accessed on Dec 2, 2020.

3.3.3.5 FPGA Synthesis

A tool flow to synthesize BlackParrot using Xilinx Vivado for a specific FPGA target and simulate the synthesized netlist using Synopsys®VCS was created. Here, the FPGA utilization reports for BlackParrot with different cache organizations are generated using the Vivado synthesis tool flow. The target FPGA was a Xilinx Artix-7 FPGA (XC7A200T) [92]. Table 15 summarises the utilization for BlackParrot with different cache configurations ([commit: 708ae8fb76](#)). For each case, 64-byte, 32-byte, and 16-byte cache lines, respectively, and an 8-byte fill width were used.

Table 15: FPGA Utilization Breakdown of BlackParrot with Different Cache Organizations

Cache Organization	LUTs	Block RAMs (36Kb)	DSP48	Flip Flops
32KB, 8-way	24752 (18.39%)	46.5 (12.74%)	11 (1.49%)	10402 (3.86%)
16KB, 4-way	22335 (16.59%)	26.5 (7.26%)	11 (1.49%)	8594 (3.19%)
8KB, 2-way	21231 (15.77%)	16.5 (4.52%)	11 (1.49%)	7681 (2.85%)

Note: BlackParrot is a **continuously evolving** processor core and the version used for this assessment was accessed on May 12, 2021.

BlackParrot is ASIC-optimized, and the utilization numbers shown in Table 15 is this ASICOptimized version of the core synthesized for an FPGA. By devoting more attention to mapping the core more efficiently onto the FPGA, the utilization can be decreased further.

Moreover, the area utilization reports and the cost of Xilinx FPGAs indicated that users of the core would only have to spend \$100 to buy an Artix-7 (XC7A50T) FPGA of which TinyParrot would occupy approximately 50%, the rest of which could be used to fit in their own accelerator.

3.3.4 ParrotPiton

ParrotPiton is the integration of the BlackParrot core into the BYOC framework. The motivations for this integration were three-fold and mutually beneficial to the two systems. Firstly, this would extend BlackParrot's community reach. The OpenPiton system has a larger community accumulated over the years, which would allow more people to discover the BlackParrot core and use it. Secondly, it would add a validated core with best-in-class performance and energy efficiency to the BYOC framework. Thirdly, the BYOC framework could be leveraged to compare BlackParrot's performance with other cores. This comparison could be with other RISC-V cores or with cores of other ISAs.



Figure 34: ParrotPiton Logo

Some of the TRI specifications for the connecting cores are as follows:

- L1 D-Cache (if any) should have a 16-byte cacheline width and 4-way set associativity and should be smaller or equal in size to the BPC
- L1 I-Cache (if any) should have a 32-byte cacheline width and 4-way set associativity
- L1 D-Cache should be write-through
- The core should handle remote invalidations and interrupts and support L2 atomics Remote invalidations and L2 atomics are explained in Section 3.3.4.1.

3.3.4.1 Setting up for Integration

At this stage, there were two design choices to be made before the actual integration.

3.3.4.1.1 Way mapping the BlackParrot D\$ with the BPC

The BlackParrot I\$ could directly connect to the TRI since the required configuration was available. However, for the D\$, there were two options - 8 KB size, 16-byte cacheline width, and 2-way set associativity, or 16 KB size, 32-byte cacheline width, and 4-way set associativity. While the set associativity matches in the second option, the cacheline width is larger, which would require multiple fills for every request. Further, the cache is twice the required size, resulting in half the cache being unused to maintain inclusivity with the BPC. The first option

offered the same size and cacheline width but smaller set associativity. The way mapping was handled as follows.

The 8 KB BlackParrot D\$ contains 256 indexes, whereas the 8 KB BPC contains 128 indexes. This implies that two ways of the BPC would fit into one way of the D\$. Figure 35 illustrates this design. The D\$ is at the top with an 8-bit index and 4-bit block offset, and the BPC is at the bottom with a 7-bit index and 4-bit block offset. The numbers indicate which way of BPC maps to which section of the D\$. A description of the translation between the BlackParrot way ID and the OpenPiton way ID follows.

3.3.5 Requests

The address and replacement way ID sent to the BPC are given below:

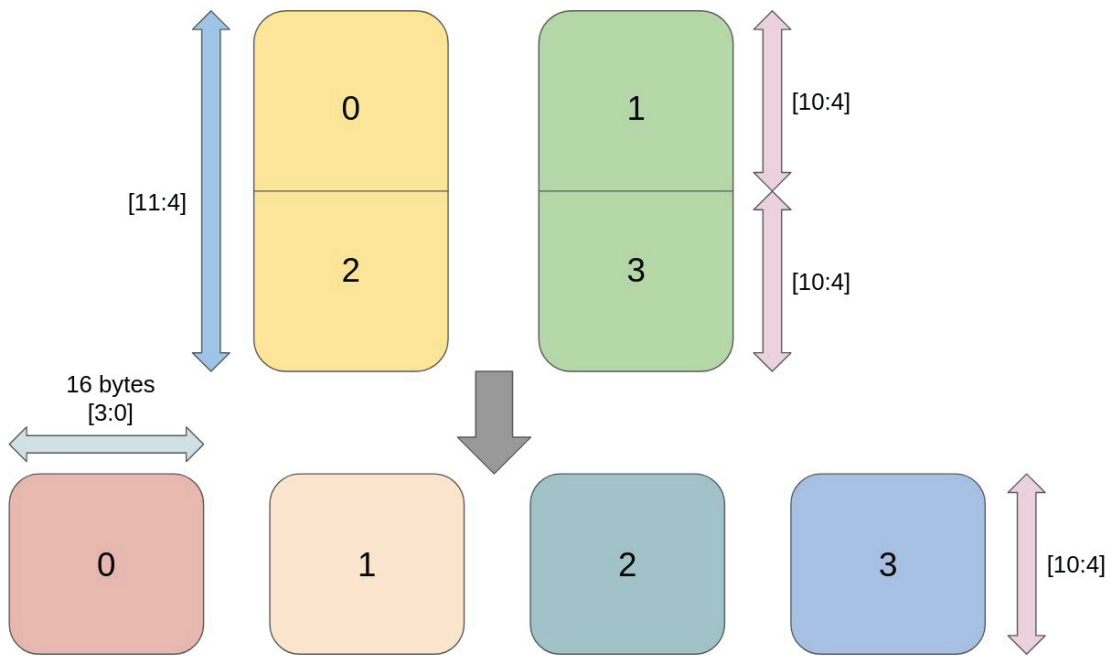


Figure 35: Way Mapping between D\$ and BPC

Address - `cache_req.addr`

Way ID - `{cache_req.addr[11], cache_req_metadata.hit_or_repl_way}`

where `cache_req` and `cache_req_metadata` are the request structures of the BlackParrot caches described earlier. By using the top bit of the cache index, each way of the L1 D\$ is divided into two ways in OpenPiton. This conversion allows the BPC to be agnostic about the actual orientation of the ways in the L1 cache.

3.3.6 Remote Invalidations

The BPC sends a 12-bit index and a 2-bit way ID to the connected core. As the name suggests, the address sent by BPC is bits 15 to 4 of the actual address. A reverse conversion of the above is required to send the invalidation command to the correct cache index and the way. The index and way ID sent to the L1 SRAMs are given below:

Index - {115_transducer_inval_way[1], 115_transducer_inval_address_15_4[10:4]}
Way ID - 115_transducer_inval_way[0]

3.3.6.1.1 L2 Atomics

OpenPiton required the LR/SC instructions to execute at the BPC level and other atomics to execute at the L2 level. Table 16 shows the RISC-V atomic instructions. All instructions are of two types - word (32-bit) and doubleword (64-bit).

At the time of this integration, the BlackParrot core did not support the swap, logic, and arithmetic atomic operations in hardware. These instructions were instead emulated in software. Support was added for these instructions in the instruction decoder, and the requests were bypassed straight through the D\$ to the CE. Further, the LR/SC hardware in the D\$ was disabled to allow execution at the BPC level.

No return L2 atomics are optimizations when the atomic operation's destination register is the zero register. This allows the core to send the request and move on to the next instruction since it does not require a response.

Table 16: RISC-V Atomic Instructions

Atomic type	Operations included
LR/SC	LR
	SC
Swap	swap
Logic	and
	or
	xor
Arithmetic	add
	min
	max
	minu
	maxu

3.3.6.2 P-Mesh Cache Engine (PCE)

The PCE design includes the above design decisions and is the main integration module. It contains an FSM that handles the different cache requests and BYOC-specific commands such as remote invalidations. It is important to note that P-Mesh is big-endian, whereas BlackParrot is little-endian. This requires a conversion in any data sent from BlackParrot to the BPC and vice versa. Further, the PCE logic performs different operations in the same module if the PCE connects to the I\$ or the D\$.

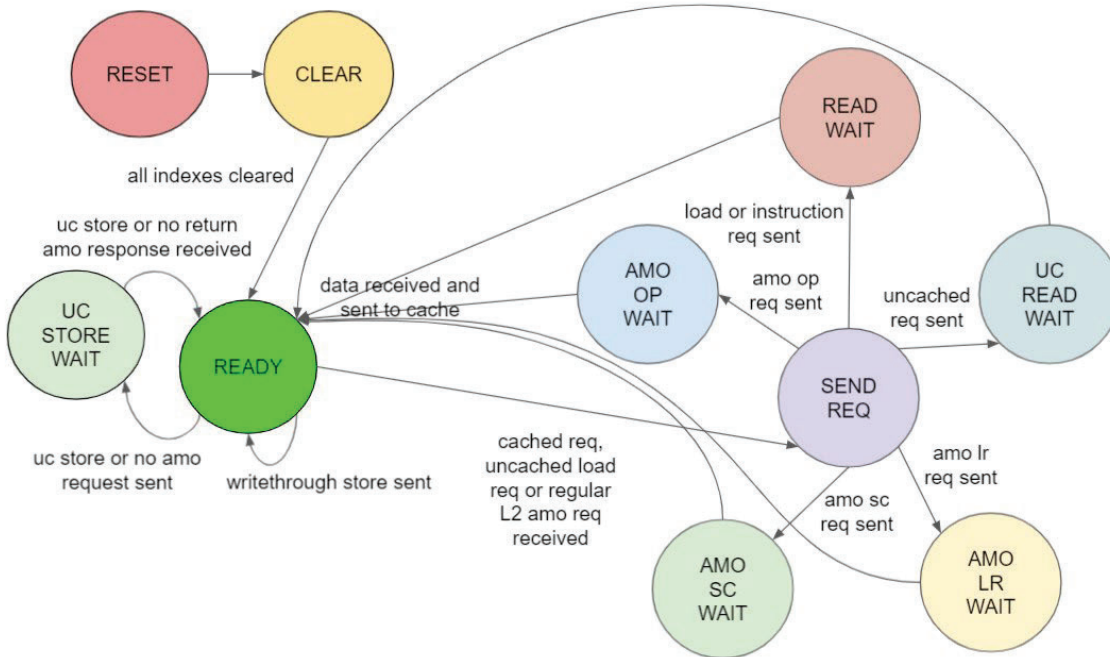


Figure 36: PCE FSM Functionality

The PCE FSM is illustrated in Figure 36 and performs the following operations:

- **reset:** The FSM starts in this state during reset. Unlike the BlackParrot CEs, an *int ret* command from the OpenPiton memory system moves the FSM to the **clear** state. OpenPiton is programmed to send this command to each core connected to it.
- **clear:** The FSM clears the tag and status memory for all indexes before moving to the **ready** state.
- **ready:** This state receives the request from the cache and decides the action required based on the type of request.
 - For a write-through store, the FSM sends the data to the BPC, and the PCE is **ready** to accept a request in the next cycle since the cache is not expecting a response. When the response from the BPC does arrive, the module silently acknowledges it.
 - For uncached stores and no return L2 atomics, the FSM sends the corresponding BYOC request and moves to the **uc_store_wait** state.
 - For all other requests, the FSM moves to the **send_req** state.
- **uc_store_wait:** This state is triggered when an uncached store or no return L2 atomic request is sent to the BPC. OpenPiton expects the core to wait until a non-idempotent operation completes, due to the possibility of request reordering in the memory system. In order to adhere to this, all non-idempotent operations wait for the corresponding response in this state. This modification potentially results in a performance reduction in the system since the L1 cache is ready to send subsequent requests, but the memory system is not ready to accept them.

- **send_req:** This state is triggered when the PCE receives any request other than the ones mentioned above, handled as follows:

For all requests apart from LR (load or store miss, uncached load, SC or atomic logic, or arithmetic operations), the FSM sends the corresponding request to the BPC, with the appropriate changes in the data and way ID. It moves to the respective states as shown in Figure 36 if the next level accepts the request.

For an LR request, a backoff mechanism is implemented. A backoff mechanism is necessary for a multicore system to avoid cacheline bouncing and ensure forward progress. For example, consider a situation where all the cores request a single cacheline to perform an LR operation. This cacheline could be reserved for a core when requested, but the reservation could cease before the corresponding SC request can complete due to an LR request from another core. In this situation, the cacheline reservation bounces from one cache to another, with none of the caches completing an SC request successfully.

The backoff mechanism involves a counter in the PCE that is started upon a failed SC request. This counter prevents the core from sending a subsequent LR request until a specified limit is reached. By having such a mechanism, at least one core will successfully complete an SC request, thereby releasing the other cores to follow suit. There are more sophisticated mechanisms such as exponential backoff, which could be a good future exploration.

- **uc_read_wait:** This state waits for the uncached data to arrive from the BPC, sends the data to the cache, and moves to the **ready** state when the data is accepted.
- **amo_lr_wait:** This state waits for the LR data to arrive from the BPC, sends the data to the cache, and moves to the **ready** state when the data is accepted.
- **amo_sc_wait:** This state waits for the SC data to arrive from the BPC, checks the data to identify if the linear backoff should start (if the data is not 0), sends the data to the cache, and moves to the **ready** state when the data is accepted.
- **amo_op_wait:** This state waits for the logic or arithmetic atomic operation data to arrive from the BPC, sends the data to the cache, and moves to the **ready** state when the data is accepted.

Remote invalidations are handled outside the PCE FSM, since the PCE needs to accept these commands regardless of what operation it is currently performing. OpenPiton provides four types of invalidations: *dcache_inval*, *icache_inval_all_way*, *icache_inval*, and *dcache_inval_all_way*. Of these, the first two are currently used in ParrotPiton.

When an invalidation is received, the PCE sends the corresponding command to the cache SRAMs. It clears all ways for the I\$ and sets the coherence state of the converted index and way ID for the D\$. The transaction is complete when the L1 cache responds with the acknowledgment.

3.3.6.3 Full System Integration

Figure 37 shows a ParrotPiton tile inside the full OpenPiton system. Both I\$ and D\$ connect to their own PCEs. On the request path, a fixed priority arbiter chooses the request to send to the BPC, and a FIFO holds back-to-back requests (in the case of write-through stores). A FIFO holds back-to-back responses routed to the correct PCE based on the response type on the response path.

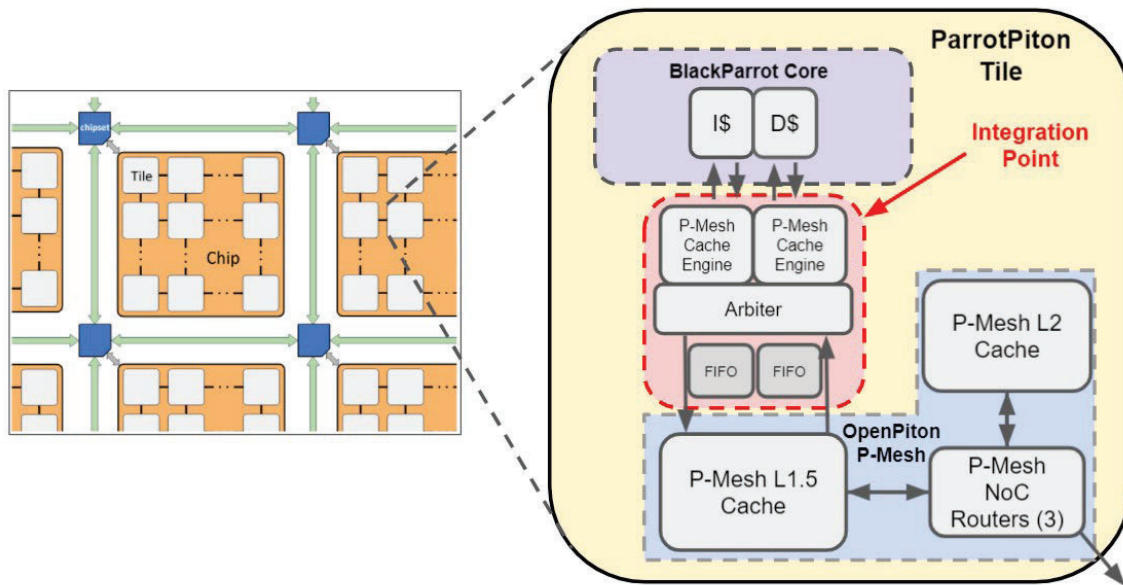


Figure 37: ParrotPiton Tile

OpenPiton is integrated with another RISC-V core, Ariane [95]. The memory system uses the Core Local Interrupt Controller (CLINT) and the Platform Level Interrupt Controller (PLIC) of the Ariane core for generating interrupts. BYOC exposes three different interrupt signals, timer interrupt, software interrupt (which is also called Inter Processor Interrupt (IPI)), and external interrupt. All three signals are connected directly to the BlackParrot interrupt handling logic.

3.3.6.4 Baremetal Testing

ParrotPiton was tested in simulation and FPGA. Initial testing involved the RISC-V test suite (physical, virtual, physical with timer interrupts, and atomic tests) and a suite of C tests provided by OpenPiton that tested operations such as atomics and interrupts in single-core and multi-core configurations.

3.3.6.4.1 Simulation

Simulation testing was the initial strategy to identify any bugs during the integration process. All facets of the PCE were tested: LR/SC, other atomics, invalidations (in multi-core configurations), and regular cache operations. The interrupt test enabled the interrupts by writing to appropriate CSRs, waited for the interrupt and cleared it, and repeated this operation for the specified number of times. All the tests passed in simulation leading to the next phase of testing

on the FPGA since OpenPiton provides an FPGA infrastructure, and the goal was to boot Linux on ParrotPiton using the FPGA.

3.3.6.4.2 FPGA

OpenPiton provides two commands for use with an FPGA. The *protosyn* command builds a bitfile for the specified core, and the *pitonstream* command allows programs to be streamed through UART onto the FPGA. The core, along with the memory system, executes the program and streams the print data through UART onto the screen. The target FPGA for ParrotPiton was Genesys 2.

Running the *protosyn* command with a single BlackParrot core generated a bitfile, and the same programs used in the simulation were used for testing on the FPGA. All the programs used for simulation ran on the board and generated the expected outputs. Table 17 shows the utilization results for single-core, dual-core and three-core ParrotPiton at [this](#) BlackParrot commit on the Genesys 2 FPGA (*xc7k325tffg900-2*).

Table 17: FPGA Utilization for different ParrotPiton Configurations

*BlackParrot is **continuously evolving** so this is a snapshot in time*

Configuration	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP48
Single-core	69,333	3,242	128	50,465	37	30	12
Dual-core	117,787	4,784	223	72,063	68	56	23
Three-core	156,997	6,326	316	93,880	99	82	34
Single-core	34.02%	5.07%	0.2%	12.38%	8.31%	3.37%	1.43%
Dual-core	55.34%	7.48%	0.35%	17.68%	15.28%	6.29%	2.74%
Three-core	77.03%	9.88%	0.49%	23.03%	22.25%	9.21%	4.05%

Table 18 shows the Hierarchical Utilization for Some of the Major Components of the ParrotPiton System

Component	Logic LUTs	LUTRAMs	SRLs	FFs	RAMB36	RAMB18	DSP48
ParrotPiton	69,333	3,242	128	50,465	37	30	12
BlackParrot core	19,823	822	95	7,105	6	25	11
L2 cache	14,516	0	0	7,372	21	1	0
FMA	2,814	0	71	165	0	0	11
D\$	1,801	180	0	980	0	16	0
I\$	861	182	0	1,707	0	8	0
Regfiles	763	0	0	492	5	0	0

An additional step was to run six SPEC2000 benchmarks [2,26](*gzip2*, *vpr*, *parser*, *bzip*, *crafty* and *mcfl*) on this system. These benchmarks worked in BlackParrot as well at this stage, so they were the natural next step.

3.3.6.5 Linux Capability

Running Linux on the ParrotPiton system was the next step in the process. There were a few steps before testing it on the board:

- ParrotPiton had to run with the bootrom option enabled in BlackParrot to use the bootrom provided by OpenPiton.
- OpenPiton-provided Linux image was given to the OpenSBI [61] framework used by OpenPiton in order to generate a payload written to an SD card. This SD card would be used in the Genesys 2 FPGA setup.

To debug the Linux boot, litmus tests were used to isolate bugs and fix them with reasonable simulation times. “A litmus test is a small parallel program designed to exercise the memory model of a parallel, shared-memory computer” [3]. Litmus tests are generated using the Litmus Tool [3] and CHERI-Litmus [8], a lightweight clone of the Litmus Tool that allows litmus tests to run on baremetal configurations. After the debugging process, Linux booted on single-core, dual-core, and three-core ParrotPiton. Figure 38 shows the ParrotPiton Linux boot screen for the three-core version.

```
NFS preparation skipped, OK

Welcome to Buildroot
buildroot login: root

      _ _ _ _ _
     / / / / /
    / / / / /
   / / / / /
  / / / / /
 / / / / /
/ / / / /

  _ _ _ _ _
 / / / / /
/ / / / /
/ / / / /
/ / / / /
/ / / / /
/ / / / /

# cat /proc/cpuinfo
processor       : 0
hart           : 2
isa            : rv64imafd
mmu            : sv39
uarch          : bsg, blackparrot

processor       : 1
hart           : 0
isa            : rv64imafd
mmu            : sv39
uarch          : bsg, blackparrot

processor       : 2
hart           : 1
isa            : rv64imafd
mmu            : sv39
uarch          : bsg, blackparrot

# /tetris
  _ _ _ _ _
 / / / / /
/ / / / /
/ / / / /
/ / / / /
/ / / / /
/ / / / /

1-Player Game   vitetris 0.5
2-Player Game
Netplay
Mode [A-type]
-----
Options
Highscores
```

Figure 38: Three-core ParrotPiton Booting Linux

3.3.7 HammerParrot

Traditionally, CPUs and GPUs (or similar manycore architectures) are separate chips on the motherboard that communicate over sophisticated bus protocols like PCIe. This setup is advantageous for the reasons listed below -

- Integrated CPU and GPU chips usually favor the CPU and use the GPU as a tinier co-processor typically weaker than a dedicated GPU. Dedicated chips have more space for more processor cores or functionality.
- Dedicated GPU cards need high bandwidth channels to DRAM, which is not quite possible on an integrated chip.

However, the problem with this approach is,

- Communication at the board level requires a sophisticated bus protocol which is typically harder to set up and verify. PCIe is a popular choice for such a bus architecture which warrant dedicated teams in large companies to ensure functionality. The number of PCIe experts is even lower for open source and academic research projects.
- The bus protocol increases the latency of communication between the host CPU and the GPU.
- Dedicated chips for CPUs and GPUs provide higher performance at the expense of higher power consumption.

HammerBlade (referred to as the manycore in short) [13] is a Tiled Manycore designed for machine learning and graph analytics workloads. It builds on the MIT RAW [76] processor and the 511 RISC-V core Celerity architecture [45]. High-level software written in PyTorch/GraphIt maps their operations to the manycore hardware architecture using CUDA-lite and an intermediate representation layer designed for HammerBlade [6]. The programmer uses the SPMD (Single Program Multiple Data) programming model to program the manycore. More information about the system architecture will follow in later sections.

Like any other traditional host-accelerator setup, the HammerBlade manycore relied on an offchip x86 host CPU. Figure 39 shows the current setup that connects the host CPU and the HammerBlade manycore.

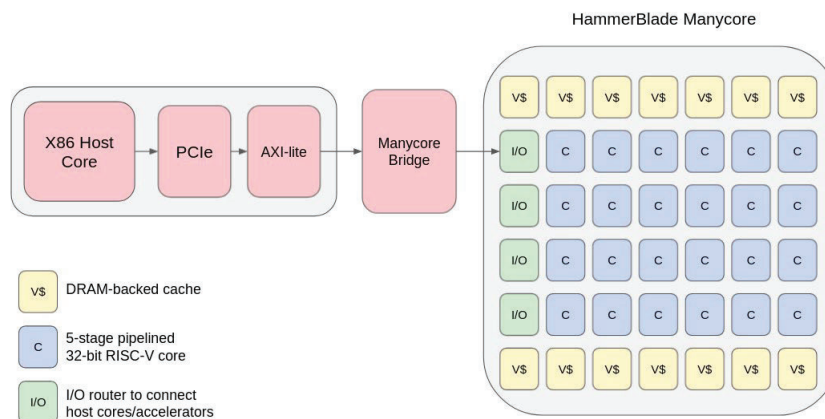


Figure 39: HammerBlade with off-chip x86 Host Core

Previous comments on hosting accelerators and CPU and GPU integration concerns can be alleviated by using an open-source, customizable host core like BlackParrot. It offers flexible integration and benefits from being a tiny core that shares the same die area as the HammerBlade manycore. The proximity provides the flexibility of adding scalable, specialized hardware modules that can achieve peak system performance compared to complicated off-chip buses such as PCIe. This resulting system is codenamed HammerParrot.

3.3.7.1 HammerBlade System Architecture

The HammerBlade manycore is a sea of tiles connected by a 2-D mesh network. The tiles on the network can be of three types - Compute, Accelerator, and Memory. Each tile, in addition, contains an endpoint interface that manages communication with other tiles.

3.3.7.1.1 Compute Tile

The compute tile in the HammerBlade manycore consists of a 5-stage pipelined, 32-bit RISC-V processor. The core supports integer, atomic, and control instructions, among others, and has a floating-point unit. Refer to the HammerBlade GitHub repository [13] for a full list of supported instructions. Figure 40 shows the compute tile's core pipeline.

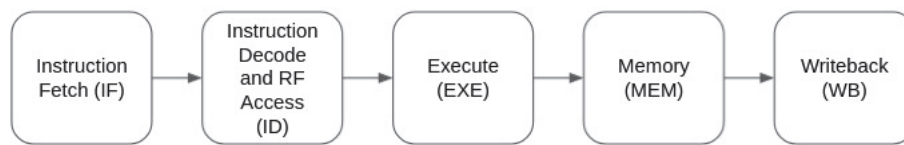


Figure 40: Compute Tile Core Pipeline

Each processor core has an I\$ and data memory. The I\$ is a 1024 entry, direct-mapped cache, and the data memory is a 4KB, single-ported memory that the core can use as a scratchpad. The data memory maps to a portion of the global address space, which other tiles on the network can access. The core pipeline is a standard 5-stage pipeline with some modifications that allow remote loads and stores to memory locations on other compute tiles or the global memory.

3.3.7.1.2 Memory Tile

The memory tile is also called a Victim Cache (V\$). It has a pipelined datapath with configurable associativity and cache size, with each cache line being an integer multiple of 32-bit words. The cache follows a write-back, write-allocate, and fetch on write policy. V\$s are placed at the north, or south sides of the compute array and are backed by high bandwidth DRAM. The DRAM space is striped across V\$s in a cache line granularity. There can be multiple rows of V\$s on the north and south side as well. In this case, the cache line is striped from the inner to the outer rows.

3.3.7.1.3 Accelerator Tile

Accelerator tiles are tiles that can be used as an accelerator. These tiles are expected to occupy nodes to the east and west of the compute array and connect to the compute array via I/O routers. BlackParrot will act as an “accelerator” to the manycore and be connected to the east (and possibly west) of the manycore array. Section 3.3.5.2 gives more details about how this connection will be made.

3.3.7.1.4 Networking and Addressing

Tiles are connected with a 2-D mesh network. There are two physical networks - One network is used to send requests, and the other network is used to receive responses with routing algorithms that avoid deadlock. HammerBlade uses a 32bit Partitioned Global Address Space (PGAS) with an efficient addressing scheme that clearly defines nodes on the network. The HammerBlade manycore uses a packet mechanism to communicate between tiles and global memory on the network. Each packet contains the necessary information to remotely carry out the desired operation, the address, and the tile coordinates. Response packets contain sufficient information to identify the request that originally generated it. More information on the network and addressing schemes used in HammerBlade can be found in [13].

3.3.7.2 HammerParrot Hardware

Figure 39 shows the hardware modules required by an x86 core to host HammerBlade. Since BlackParrot will share the same die as the manycore, complicated bus protocols such as PCIe are no longer required, and the manycore bridge is the only hardware module left to be designed. The x86-HammerBlade bridge consists of FIFOs that the host core uses to send and receive data to/from the manycore. As mentioned earlier, the x86-HammerBlade system is PCIe based, and the PCIe protocol imposes strict response latency requirements, which, if not adhered to, causes the entire system to hang. The FIFOs in the bridge help to adhere to these requirements. The x86-HammerBlade bridge inspired the design of the HammerParrot manycore bridge.

3.3.7.2.1 Bridge FIFO Interface

The HammerParrot manycore bridge consists of the traditional host FIFO interface (like the x86-HammerBlade bridge) that can be written to and read from, using memory-mapped stores and loads, respectively. A request packet constructed in software is written to the host request FIFO as a series of 32-bit stores. The host request FIFO is modeled as a SIPO on the BlackParrot side while the HammerBlade response and request FIFOs are PISOs. 4 32-bit stores create a 128-bit packet that can be sent over the network as a manycore request packet. The manycore enqueues the responses to these requests in the host response FIFO and BlackParrot retrieves them using four 32-bit loads. A similar operation is performed when the host chooses to accept manycore requests. The host, however, does not generate responses to requests from the manycore. A manycore packet is sent/retrieved using four 32-bit stores/loads instead of 2 64-bit stores/loads or a single 128-bit store/load because the host software API exposes a packet interface that renders manycore packets as a 4-element array of 32-bit unsigned integers.

3.3.7.2.2 Bridge MMIO Interface

One of the downsides of the x86 host interface was the inability to map the manycore into its own address space, thereby not being able to access any location on the manycore directly. The x86 host core, therefore, had to rely on software address translation and packet creation which incurs an overhead of 100s of cycles and an additional overhead of 1000s of cycles for transmitting the packet over PCIe.

BlackParrot overcomes these shortcomings by having the flexibility to parameterize its physical address space to map the whole of the manycore's 32-bit address space, allowing BlackParrot to access everything inside the manycore like any other compute or accelerator tile. Any BlackParrot address (with the high bit set to denote it is a manycore MMIO request) is viewed as an endpoint address and can be translated to an address and tile coordinates that uniquely identify the desired resource. BlackParrot's single-core uses a Bedrock interface to the manycore bridge. The manycore bridge readily converts Bedrock messages into HammerBlade packets. Furthermore, since, BlackParrot is on the same network as the manycore, the overhead of the PCIe bus is eliminated, thereby creating an interface that achieves peak performance. The software only has to execute a single instruction (i.e., load/store) to communicate with the manycore.

Direct access into the manycore address space requires extra care, however. BlackParrot's configurable address space allows it to access any address in the manycore address space. However, performing loads and stores to these addresses can receive responses in a different order because of network latency. Requests to tiles farther away from BlackParrot on the network will take longer to return with a response, while requests to closer tiles might return with a response immediately. A reorder FIFO from BaseJumpSTL was used to handle out-of-order responses from the manycore.

Any outgoing request reserves an ID in the reorder FIFO when it is sent to the manycore network. Responses can return in any order and populate the corresponding entry in the FIFO. The reorder FIFO signals that it has a valid packet at its output only when the earliest ID's request returns with a response. BlackParrot currently supports a configurable number of such requests to be in flight at the same time.

3.3.7.2.3 System-level Address Maps

Documentation in the HammerBlade Github repository [13] defines in detail the addresses that map to different compute and memory tiles and how to construct them. Since BlackParrot would like to embed the entire manycore address space into its own, BlackParrot's address space is configured to use 42-bits. The manycore is designated as an off-chip ASIC that can communicate with BlackParrot by setting the high bits in BlackParrot's address space. Table 19 shows the full address map.

Table 19: HammerParrot Address Map

Address	Use
11_1000_0000_0PPP_PPPP_PPPP_PPPP_PPPP_PPPP_PPPP	Manycore Bridge FIFO
00_0000_yyxx_1PPP_PPPP_PPPP_PPPP_PPPP_PPPP_PPPP_PPPP	Global DRAM banks. yyxx denotes DRAM coordinates.
10_yyy_XXXX_XXXPP_PPPP_PPPP_PPPP_PPPP_PPPP_PP00	Victim cache (V\$) address + tags. yyy denotes the V\$'s Y coordinate (north/south). XXXX_XXX denotes the X coordinate
11_0000_00YY_YYYY_YXXX_XXXX_PPPP_PPPP_PPPP_PP00	Compute Tiles YY_YYYY denotes the Y coordinate XXX_XXXX denotes the X coordinate

BlackParrot fulfills the traditional host interface by performing memory-mapped stores and loads to the manycore bridge. Table 20 defines the different addresses in the manycore bridge that map to different FIFOs.

Table 20: HammerParrot Bridge FIFO Address Map

Address	Use
0x1000	Write to the host request FIFO
0x2000	Check for available host request credits
0x3000	Read from the manycore response FIFO
0x4000	Check for available entries in the manycore response FIFO
0x5000	Read from the manycore request FIFO
0x6000	Check for available entries in the manycore request FIFO

3.3.7.2.4 Top-level Integration

Each BlackParrot tile occupies three nodes on the manycore network. All three nodes together form the manycore bridge. One node contains the host FIFO and MMIO interfaces and used as a means for the BlackParrot core to make/receive requests and receive responses. In the HammerParrot system, BlackParrot uses a 32KB, 8-way cache with 64-byte cache lines with a fill width of 8 bytes which means every request to DRAM will be a 64-bit load/store. However, the manycore can only support 32-bit fills. Therefore, every request from BlackParrot is split into two requests to the manycore and injected into the network at two different nodes to handle this mismatch. When both the split requests return with their responses, the splitter combines them and sends a single 64-bit data packet back to BlackParrot. The additional two nodes are occupied to accommodate this setup. Figure 41 shows the HammerParrot manycore bridge.

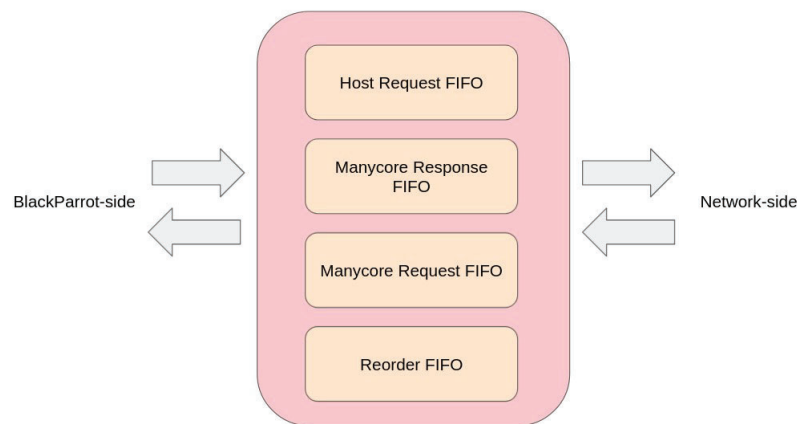


Figure 41: HammerParrot Manycore Bridge

3.3.7.2.5 HammerBlade-x86 Simulation Infrastructure

The HammerBlade software stack written in C/C++ is based on CUDA-lite hosted at [6]. The library handles many things, including software-level address translation, sending and receiving packets to the manycore hardware, writing and reading from tiles, creating tile groups and synchronizing between tiles within a tile group, and allocating kernels to tile groups, and more. A system would only have to implement a platform-specific API exposed by the library to run the CUDA-lite host code. The API dictates how the platform communicates with the manycore and can be thought of as a hardware abstraction layer that hides the underlying communication mechanism from higher levels of software. However, this complicated software stack has only been compiled and tested with an x86 host core running Linux and an Amazon EC2 F1 instance. Figure 42 shows how the existing system was simulated using Synopsys VCS.

The x86 host core executes CUDA-lite code until it needs to send/receive data to/from the manycore. To send/receive data, the platform API uses a SystemVerilog Direct Programming Interface (DPI) to communicate with hardware FIFOs. These FIFOs simulate the x86 manycore bridge that connects and communicates with the manycore.

3.3.7.2.6 Porting CUDA-lite to RISC-V

Replacing the x86 core with BlackParrot can proceed either by booting up Linux on BlackParrot in simulation and running the CUDA-lite binary on top of Linux or running the binary in bare metal mode. The Linux route is not a good starting point because of the high simulation time and longer debugging process. To run in bare metal mode, the CUDA-lite library needed to be compiled using a lightweight implementation of the standard C library called PanicRoom [21]. PanicRoom is a port of Newlib, a lightweight C library for embedded systems, and some system calls for file I/O and a DRAM-based file system called LittleFS. The CUDA-lite library was easy to port to PanicRoom save for a few missing libraries, which were quickly resolved with help from experts. The next step was to then replace the x86 core in the simulation infrastructure with BlackParrot.

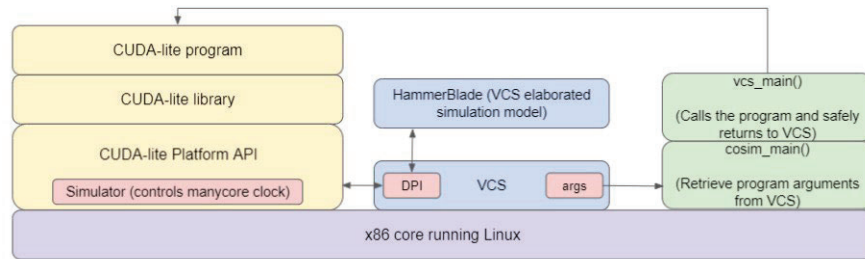


Figure 42: HammerBlade Simulation Infrastructure in VCS

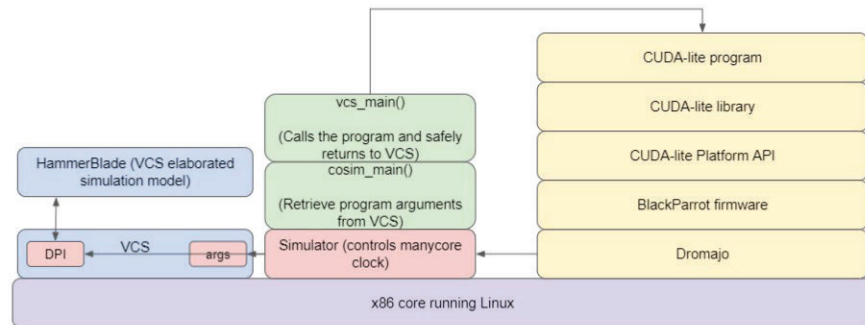


Figure 43: Dromajo+HammerBlade Simulation Infrastructure in VCS

3.3.7.2.7 Software Validation using Dromajo HammerBlade

BlackParrot and HammerBlade are both continuously evolving machines. A direct integration meant that if something goes wrong, one had to debug either the BlackParrot RTL, HammerBlade RTL, or the software. To reduce this initial verification burden as well as to accelerate the software porting effort, Dromajo [10], an open-source C++ reference model for a 64-bit RISC-V core from Esperanto Technologies, was used. Dromajo is originally a RISC-V reference model used to verify RTL correctness through RTL co-simulation. The BlackParrot project uses Dromajo extensively to verify RTL correctness after every feature change. It works by executing each instruction when the RTL commits an instruction and comparing results with the RTL. However, Dromajo can also be used in a standalone mode which is exploited here.

Dromajo's source code was modified to create the HammerParrot bridge in software to emulate the HammerParrot set up as closely as possible. The Dromajo-HammerBlade bridge supports only the traditional FIFO interface like the x86 system, and the MMIO support is a work in progress.

3.3.7.2.8 Dromajo HammerBlade Simulation Infrastructure

Since Dromajo is still only a RISC-V software model, it cannot directly communicate with the hardware running in the VCS simulation environment. Therefore, the packets from the Dromajo FIFOs are communicated to the manycore hardware via the Direct Programming Interface like in the x86 case, which requires that the Dromajo source code be compiled with the VCS executable and another x86 binary that interfaces between VCS and Dromajo through DPI. The x86 binary, in this case, is called the "Simulator" (note: this is not VCS. Refer to Figure 42) that controls the manycore clock. In this case, the Simulator will also execute N instructions in the Dromajo platform at every manycore clock tick and poll the DPI interface for any request and response packets from the manycore while transmitting any host request packets from Dromajo to the manycore. The platform API also requires implementations for other functions that require polling the hardware directly for data, such as the manycore configuration stored in a ROM (not mapped to the manycore address space) or checking if the manycore has finished cycling through a reset or retrieving arguments before calling main. Such requests are handled by sending and receiving packets to the Simulator, which emulates the BlackParrot host that performs these operations. Figure 43 summarises this whole setup.

This cross-platform integration effort revealed many inconveniences in the HammerBlade simulation infrastructure. Therefore the Dromajo+HammerBlade integration was incredibly beneficial for both projects and has set the stage for HammerParrot's simulation infrastructure.

3.4 Application

Although BlackParrot has competitive performance and good power efficiency as compared to other RISC-V cores, the lack of I/O devices significantly limits its use cases, for example running standard Linux distributions. In order to make BlackParrot a fully functional SoC, the team brought a complete yet open source network solution, including both hardware and software, to BlackParrot. For the hardware, first the RISC-V PLIC module from OpenTitan [19] was tested with BlackParrot. Second, the 1G Ethernet MAC module, *eth_mac_1g_rgmii_fifo*, from Alex Forencich [47] has been used to create our own BlackParrot Ethernet controller. Third, the S-mode external interrupt line, which is one of the key components for BlackParrot to run RISC-V Linux kernel, has been added to BlackParrot. Extra efforts have been made to ensure all the hardware designs are largely technology-independent, which means developers only need to replace a small, technology-dependent part of the designs to port these designs to different technology nodes or different FPGA boards.

For the software, the TCP/IP library, lwIP [17], and the SSL library, wolfSSL [31], were ported to BlackParrot SDK [27]. These lightweight libraries provide all the necessary functionalities to enable secure network connections in BlackParrot. Three examples were created to demonstrate the use of these libraries. The first example is a collection of TCP/SSL server/client programs, while the second and third examples are the boot loaders for the BlackParrot System on Module.

Besides those libraries, both Linux and OpenSBI were upgraded in the BlackParrot SDK to a higher version (from Linux 5.8 and OpenSBI 0.7, to Linux 5.15 and OpenSBI 1.0). The Liteeth driver [15] and the PLIC interrupt controller driver [24] from Linux kernel 5.15 have also been tested on BlackParrot with the Ethernet controller and the PLIC module. Both the Linux drivers and the Linux kernel can now work out-of-box on BlackParrot with no custom changes. Finally, a successful NFS boot of a default Linux distribution from Yocto [28] on BlackParrot was demonstrated. Both the hardware and software components are fully free and open-source, and they have all been validated on an FPGA.

3.4.1.1 Introduction

According to the OSI model (figure 44) [20], there are 7 abstraction layers in computer networking, from the physical layer (layer 1) to the application layer (layer 7). The wellknown HTTP protocol is at the application layer, while the TCP, IP protocols are at the transport layer and the network layer, respectively. The Ethernet protocol defined by IEEE 802.3 involves both the physical layer and the data link layer. The Ethernet physical transceiver (PHY) operates at the physical layer. It sends and receives the analog signals on the wired cable. With different types of physical media (for example, copper in 1Gbps, optical fiber in 10Gbps speed case), different types of PHYs are used. The Ethernet MAC operates at the data link layer. It sends/receives the raw Ethernet frames to/from the PHY.

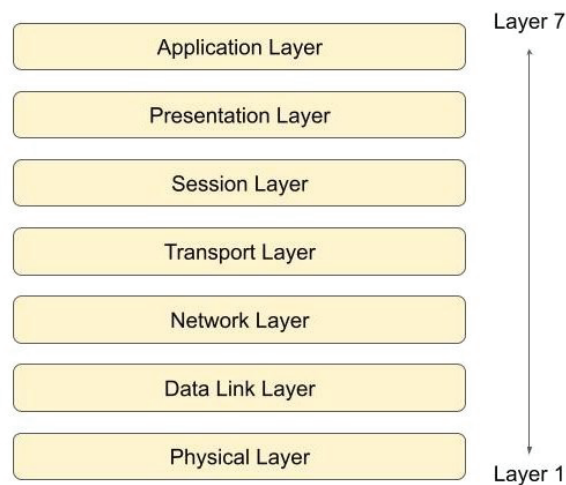


Figure 44: The OSI Model

Figure 45 shows the overview of the typical Ethernet hardware. The PHY device is a dedicated external module that is not within the digital design boundary. The original standard interface between MAC and PHY is called the Media-Independent Interface (MII). It allows the MAC module to connect to different types of media, and hence different types of PHYs, without the need to redesign the MAC hardware. Besides MII, there are other variants as well, including GMII/RGMII, which are commonly seen in 1G Ethernet, and XGMII, which appears in 10G Ethernet.

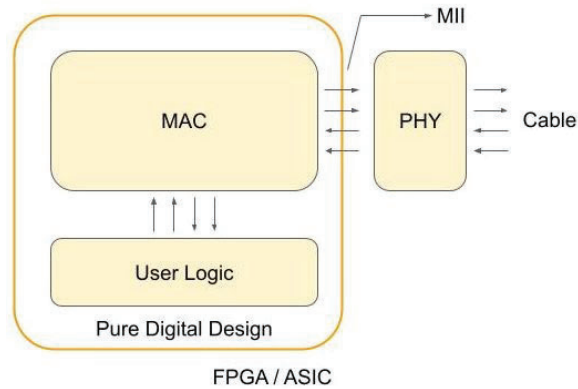


Figure 45: Ethernet Hardware Overview

The MAC module receives the layer 2 Ethernet frame (shown in figure 46) from the user logic. It then adds the preamble, start frame delimiter (SFD) and frame check sequence (FCS) to the frame before sending it to PHY. Frame check sequence is used to detect errors in the Ethernet frame. If there is an error, the receiver will just drop the Ethernet frame.

The user logic can be anything from simple Ethernet control logic to a complex TCP offload engine, depending on the specific need. The Ethernet control logic could include a hardware module that generates interrupts, a DMA engine that transfers the packets, and some control/status registers that provide the MAC access to the users. Together the Ethernet control logic and the MAC module form an Ethernet controller.

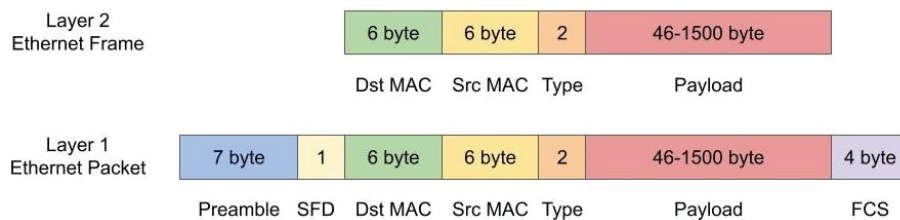


Figure 46: Ethernet Frame

3.4.1.2 Goals

The Ethernet controller is designed with the following four goals: 1. It should be fully open-source and leverage as many existing high quality, off-the-shelf projects as possible in order to minimize the hardware development cost as much as possible. 2. It should be fully compatible with the existing Linux Ethernet device driver so that the entire software stack would work seamlessly with our Ethernet controller without the need of any custom changes. 3. It is intended to be used in both FPGA and ASIC, i.e., it should be synthesized in both ASIC and FPGA with only minimal technology-dependent primitives for I/O. 4. It should support both 100Mbps and 1Gbps as they are two commonly used speeds.

3.4.1.3 Hardware Selection Ethernet MAC Selection.

In our work, several MAC modules have been considered: Liteeth [16], LeWiz [14] and Alex [47]. As pointed out in figure 47, all the three MAC cores can achieve 100M and 1G Ethernet since they all support tri-mode, which includes 10M, 100M and 1G. However, only Liteeth comes with mainline Linux (starting from Linux 5.15) support. Although Liteeth could offer such a big benefit, the cost of learning the Litex framework and new HDL language might offset the benefit. Since Liteeth is designed for FPGA and is written in a language that is not as popular as Verilog, changing the code to make it work for both ASIC and FPGA would likely require developers to learn a new language and be familiar with the Litex framework. Although the framework can convert Migen into Verilog, the auto-generated code is not human-readable and hence is not good for porting and development. Given that the target of interest was Liteeth alone, from both the development and maintenance perspectives, the overhead of learning a new language and platform for just a single module from Litex would go beyond its own advantages.

Another option is to leverage the Linux device driver from Liteeth and use the MAC module from either Alex or Lewiz, which are all written in (System)Verilog. Although Lewiz claimed to support both FPGA and ASIC, from the history of the repository on github and their responses to our previous filed issues, the project seems to have been inactive for some time and only few projects have adopted it. On the other hand, Alex's MAC module has been actively maintained so far and has been widely used. This leads to the decision that the Liteeth Linux device driver in mainline Linux should be used for the software; the MAC module should be used to create a MAC module suitable for both ASIC and FPGA; and a new Ethernet controller should be created.

	Alex Forencich	Lewiz	Liteeth
Speed (Unit: bit/sec, or bps)	Tri-mode (10M/100M/1G), 10G, 25G	Tri-mode (10M/100M/1G)	Tri-mode (10M/100M/1G)
MII interfaces	GMII, RGMII,...	GMII	GMII, RGMII,...
HDL	Verilog	Verilog	Migen
FPGA / ASIC	FPGA	Both	FPGA
Linux Device Driver in mainline kernel	X	X	O
Ethernet Controller	X	X	O

Figure 47: Comparison between Different MAC Modules

As for the MII interface, figure 48 and figure 49 show the GMII and RGMII interface respectively. Both GMII and RGMII are source-synchronous, which means the data signals are sent along with a clock signal generated from a transmitting node, rather than a global clock signal. In GMII, there are GMII_TXD and GMII_RXD for data transmission, both of which are 8 bit wide; there are also control signals for RX_DV (data valid), RX_ER (error) and TX_EN (enable), TX_ER (error). Whereas in RGMII, the data transmission lines for both RX and TX are

cut down from 8 to 4 bits, which is achieved by transmitting the data on both the rising and falling edge of the clock signal, i.e., the first 4 bits on the rising edge and the last 4 bits on the falling edge. Besides, the number of control signals are also reduced by half: The RX_CTL/TX_CTL sends RX_DV/TX_EN on the rising edge and the ‘exclusive or’ of RX_DV/TX_EN and RX_ER/TX_ER on the falling edge. Since the RGMII interface significantly reduces the pin counts, it is the chosen MII interface in our implementation.

3.4.1.4 FPGA Board and PHY Module Selection.

In order to run 1G Ethernet with RGMII interface on an FPGA, the board Zedboard [32], and the PHY module Avnet Network FMC Module [4] are selected. Zedboard has an FMC LPC interface which can be connected to the network FMC module. With some proper pin assignment on Zedboard, the RGMII signals on the FPGA can be connected to the FMC interface, and then routed to the PHY module. The PHY module will then send the signals out to an Ethernet cable.

3.4.1.5 BlackParrot Ethernet Controller

The overview of the BlackParrot Ethernet controller module is shown in figure 52. All the modules that are used in both ASIC and FPGA are placed in this module. For those platform-dependent primitives, they are placed outside *ethernet_controller* and bundled together under the wrapper module, *ethernet_controller_wrapper* (see figure 53). For example, in the case of Zedboard, IDELAYCTRL and IDELAYE2 primitives are needed to meet the timing constraints for RGMII, therefore they are put under the wrapper module. *eth_mac_1g_rgmii_fifo* is the Ethernet MAC module from Alex. The original version of the MAC module supports tri-mode and uses two 125 MHZ clocks (one with 90 degree shift) and some FPGA primitives (IDDR and ODDR) to deal with the RGMII signals. The original version also has a helpful status interface that users can leverage to detect packet corruption at the Ethernet packet level. Our main modifications to the MAC module are as follows: 1. Replace the IDDR, ODDR FPGA primitives with their counterparts in Basejump STL [80]. 2. Replace the two 125 MHZ clocks with one single 250 MHZ clock plus some generated clock logic. 3. Add IDELAYE2 primitives and use IOB packing to meet the timing constraints on Zedboard.

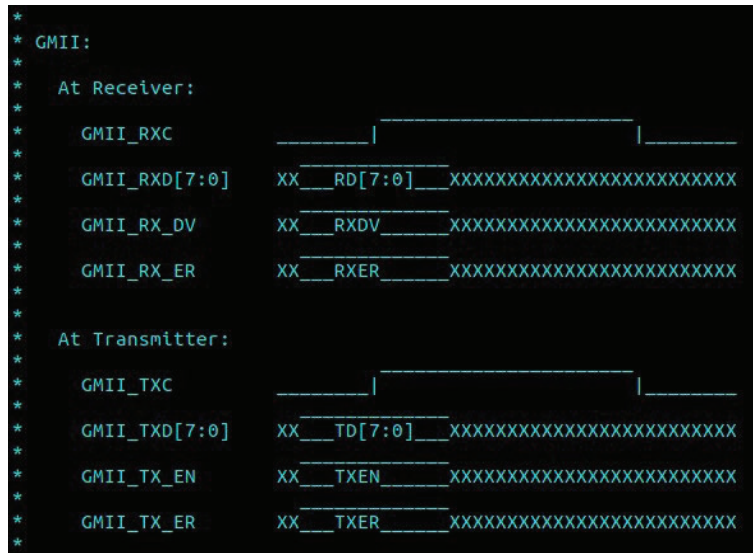


Figure 48: GMII Signals

The MAC module supports tri-mode by detecting the speed of the external PHY, which can be 10Mbps, 100Mbps or 1Gbps, and runs at the exact same speed. If the speed of the external PHY changes during runtime, the speed of the Ethernet controller will also be adapted accordingly. For example, if the speed of PHY is changed from 100Mbps to 1Gbps, the Ethernet controller will detect the speed change (by looking at the clock cycles of the RGMII RX clock from PHY) and update its speed from 100Mbps to 1Gbps.

3.4.1.5.1 ethernet_memory_map

This module presents the memory map of the Ethernet controller (figure 54). The memory map is based on the memory map from the existing Linux Liteeth device driver.

For the transmission path: First, "TX Ready" bit is checked. If it is set, the TX buffer has space and can accept new packets. Second, write the packet to the "TX buffer" and the size of the packet to the "Length of the Transmitting Packet" register. Third, do a write (any value) to the "TX Send" bit to send the packet.

For the receive path: First, "RX Event Pending" bit is checked. If it is set, the RX buffer has packets to receive. Second, read the packet from the "RX buffer" and the size from the "Length of the Received Packet" register. Third, write 1 to the "RX Event Pending" bit to acknowledge the received packet.

The "TX Event Pending" bit is set when the status of the TX buffer goes from full to empty. This is used by the Liteeth Linux driver to notify Linux to resume the Network traffic. Writing 1 to it clears the bit.

When the enable bit is set, the corresponding interrupt is enabled. Otherwise it is disabled.

The "Index of the Received Packet" is unused and is always zero. The "Index of the transmitting packet" is also unused and all the values written to it are ignored. These fields are used as the FIFO pointers to the current RX/TX slots in the Liteeth Linux driver, but since in our design the main RX/TX buffers are in *eth_mac_1g_rgmii_fifo* and does not require the driver to maintain where the current slots are, these fields are deliberately tied to zero to hide those details from the software.

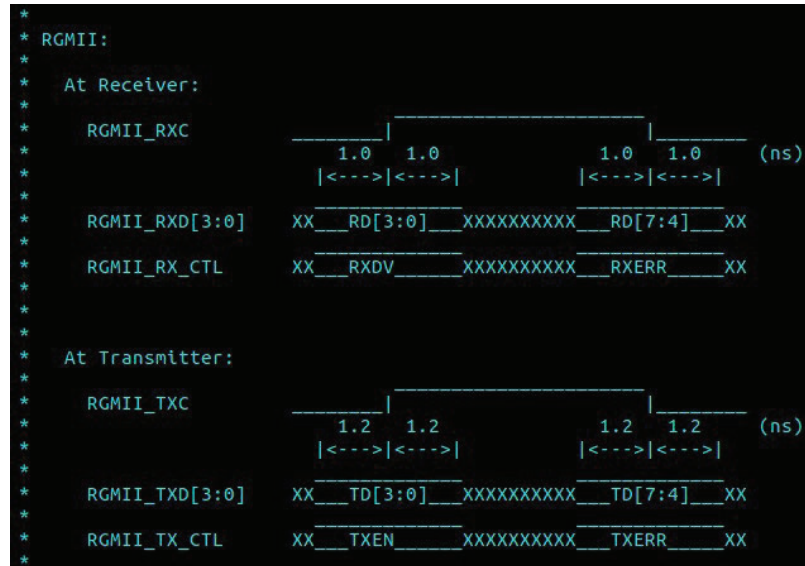


Figure 49: RGMII Signals

3.4.1.5.2 ethernet_sender and ethernet_receiver

Since the TX and RX buffers in Liteeth Linux driver are presented as a random-access memory buffer, these modules are used to do the necessary conversion from the AXI-Stream interfaces to memory arrays. Both *ethernet_sender* and *ethernet_receiver* have a FIFO with 2 slots. Each slot can contain only 1 Ethernet packet, regardless of the size of the packet. The reason to have FIFOs with 2 slots is to allow both the AXI-Stream side and the *ethernet_memory_map* side of the hardware to operate at the same time.

3.4.1.5.3 interrupt_control_unit

This module generates two level-triggered interrupts, one for RX and the other for TX. The RX interrupt is asserted high when the RX buffer is not empty, while the TX interrupt is asserted high when the status of the TX buffer changes from full to non-full.

3.4.1.5.4 eth_mac_1g_rgmii_fifo

This is the core MAC module ported from Alex (figure 50). The Ethernet packets are sent and received through the AXI-Stream interface on the left, which connects to the inner part of the Ethernet controller. Right next to the AXI-Stream interface are the TX and RX FIFOs. They store the packets and do the clock domain crossing between the user logic clock and the Ethernet TX/RX clock. Inside the *eth_mac_1g_rgmii* module, *eth_mac_1g* sends/receives Layer 1 Ethernet frames to/from *rgmii_phy_if* through the GMII interface, while *rgmii_phy_if* does the

conversion between GMII signals and RGMII signals. The RGMII signals then connect to the external PHY module. *rgmii_phy_if* is responsible for maintaining the necessary phase difference between the RGMII_TXD / RGMII_TX_CTL and RGMII_TXC. As shown in the RGMII timing diagram in figure 49, instead of sending together, RGMII_TXC is center-aligned to every beat of RGMII_TXD and RGMII_TX_CTL, that is, the rising edge of RGMII_TXC is aligned with the first 4 bits of data plus TXEN and the falling edge is aligned with the last 4 bits of data plus TXERR. In order to meet such a requirement, from figure 55, *rgmii_phy_if* generates GMII_TXC (which generates

RGMII_TXD, RGMII_TX_CTL), whose rising edges align with the rising edges of clk250_i, and generates RGMII_TXC, whose rising edges align with the falling edges of clk250_i. *rgmii_phy_if* also considers the latency between GMII_TXD and RGMII_TXD, so that the first RGMII data will always be sent along with a rising edge of RGMII_TXC. This can be seen in the same figure: ‘d1d0’ is an 8-bit GMII data and it is transmitted as two 4-bit RGMII data: ‘d0’, and then ‘d1’. ‘d0’ is always followed by a rising edge of RGMII_TXC.

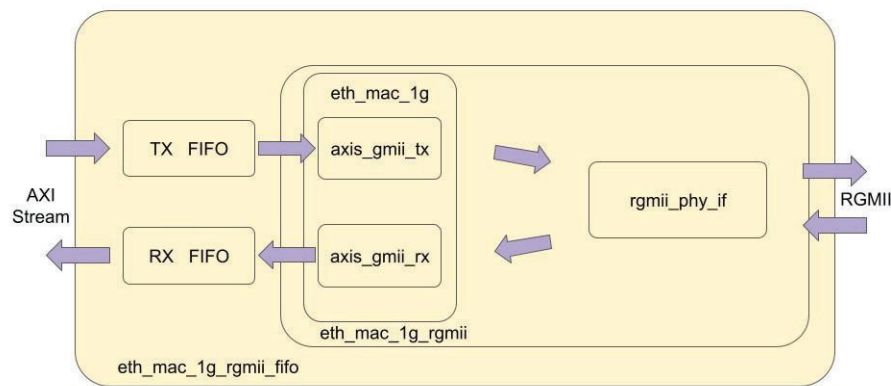


Figure 50: The eth_mac_1g_rgmii_fifo Module

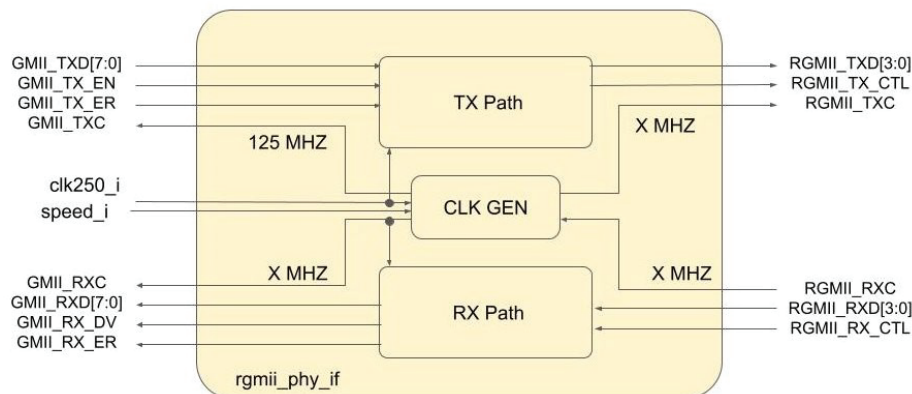


Figure 51: The rgmii_phy_if Module

Figure 51 shows the *rgmii_phy_if* module. The *speed_i* signal in *rgmii_phy_if* comes from the GMII_RXC speed detection logic in *eth_mac_1g_rgmii*. It distinguishes between 2.5 MHz, 25 MHz and 125 MHz and makes the speed of RGMII_TXC equal to GMII_RXC.

The original version of *rgmii_phy_if* uses IDDR and ODDR FPGA primitives and the 90-degree shifted clock to receive and send DDR signals. In order to make the design suitable for both FPGA and ASIC, those primitives and the clock signal have been replaced with the *clk250_i* signal (and its generated clock signals) and synthesized versions of IDDR and ODDR.

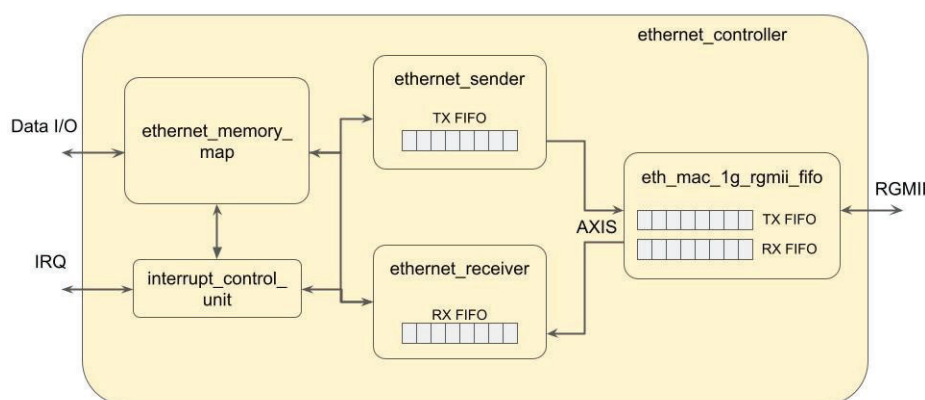


Figure 52: The *ethernet_controller* Module

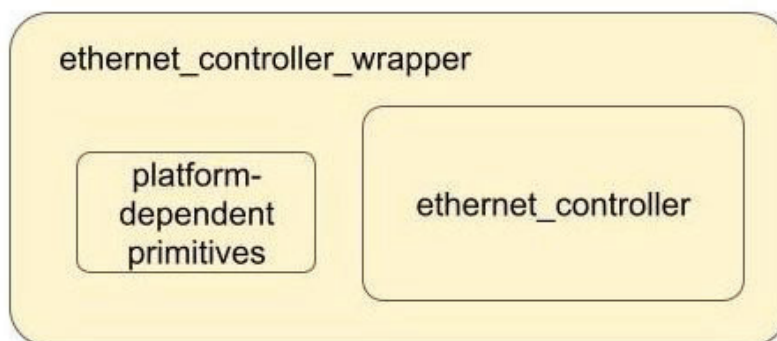


Figure 53: The *ethernet_controller_wrapper* Module

3.4.1.5.5 Clock Domains

The clock domains of the MAC module are shown in figure 56.

There are 3 clock domains in the design: *logic_clk*, *clk250_i*, and *phy_rx_clk* from the external PHY. *clk250_i* generates two additional clocks: *tx_clk* and *phy_tx_clk*. *tx_clk* is the clock used in the TX path of the MAC module, while *phy_tx_clk* is the RGMII TX clock sent to PHY.

rx_clk is the output of a clock buffer for phy_rx_clk. It is used in the RX path of the MAC module.

3.4.1.5.6 Meeting the Timing Constraints

This paragraph describes how the RGMII timing constraints are met on Zedboard. First, the IOB packing has been used to help meet the TX RGMII timing. An IOB (Input Output Block) can hold a register and it is the interface between the FPGA and the outside world. What IOB packing does is it moves an external register into an IOB. This can minimize the skews between the RGMII TX clock and the RGMII TX data signals as both the clock signal and the data signals do not have to propagate for a long distance before reaching the I/O ports. Second, the IDELAYE2 primitives have been used to help meet the RX RGMII timing. An IDELAYE2 primitive simply delays input signals. Since the RX clock signal has a longer input path than that of the RX data signals (because the RX clock signal has to go through the clock buffer before reaching the registers), IDELAYE2 primitives are applied to the RX data signals to meet the RX timing.

```
/*
 * Memory map:
 * 1. RX/TX Buffers:
 *
 *   RX Buffer:
 *   0x0000-0x0800
 *   TX Buffer:
 *   0x0800-0x1000
 *
 * 2. Register Map:
 *
 *   Readable Register:
 *   0x1000: Index of the Received Packet
 *   0x1004: Length of the Received Packet
 *   0x1010: RX Event Pending Bit
 *   0x101C: TX Ready Bit
 *   0x1030: TX Event Pending Bit
 *
 *   Writable Register:
 *   0x1010: RX Event Pending Bit
 *   0x1014: RX Event Enable Bit
 *   0x1018: TX Send Bit
 *   0x1024: Index of the Transmitting Packet
 *   0x1028: Length of the Transmitting Packet
 *   0x1030: TX Event Pending Bit
 *   0x1034: TX Event Enable Bit
 */
```

Figure 54: The Memory Map of the BlackParrot Ethernet Controller

3.4.2 RISC-V PLIC

RISC-V Platform-Level Interrupt Controller (PLIC) [25] multiplexes various interrupt sources onto interrupt lines of targets. Figure 57 shows the four main parts of the PLIC architecture: they are “interrupt signals”, “PLIC gateways”, “PLIC core” and “targets”.

“Interrupt signals” are the interrupt sources of PLIC, for example, they could come from an Ethernet device or a keyboard. A “PLIC gateway” converts its associated interrupt signal to an interrupt request, which then be forwarded to “PLIC core”, and the gateway sets the corresponding interrupt pending bit. One thing to note here is that a “PLIC gateway” only

forwards one interrupt request at a time. It would only resume the forwarding after it gets an interrupt completion message for its associated interrupt source. “PLIC core” computes the EIP (External Interrupt Pending) bit and handles the accesses of the Interrupt Claim/Complete register for each target. Finally, a “target” would be signaled an interrupt notification if its corresponding EIP in “PLIC core” goes high. A “target” is usually a RISC-V hart context (figure 58). For example, a 4-core RISC-V that supports M, S-mode could have up to 8 targets. However, in practice not all the hart contexts are connected to the PLIC. For instance, a typical RISC-V Linux system running in S-mode would only have the S-mode external interrupt lines of the RISC-V cores connected to the PLIC.

Besides the four main parts, there are also memory-mapped registers for controlling the PLIC operations. The “Interrupt Priorities” registers store the priority for each interrupt source, with 0 being disabled and higher value being higher priorities. The “Interrupt Pending Bits” registers show the pending status for each source. The “Interrupt Enables” registers store the enablement of every interrupt source of each target. The “Priority Thresholds” registers mask out all the interrupts with priorities less than or equal to it for each target. Lastly, the “claim/complete” registers are used to acquire the interrupt source ID of each target and send the interrupt completion message to the associated gateway.

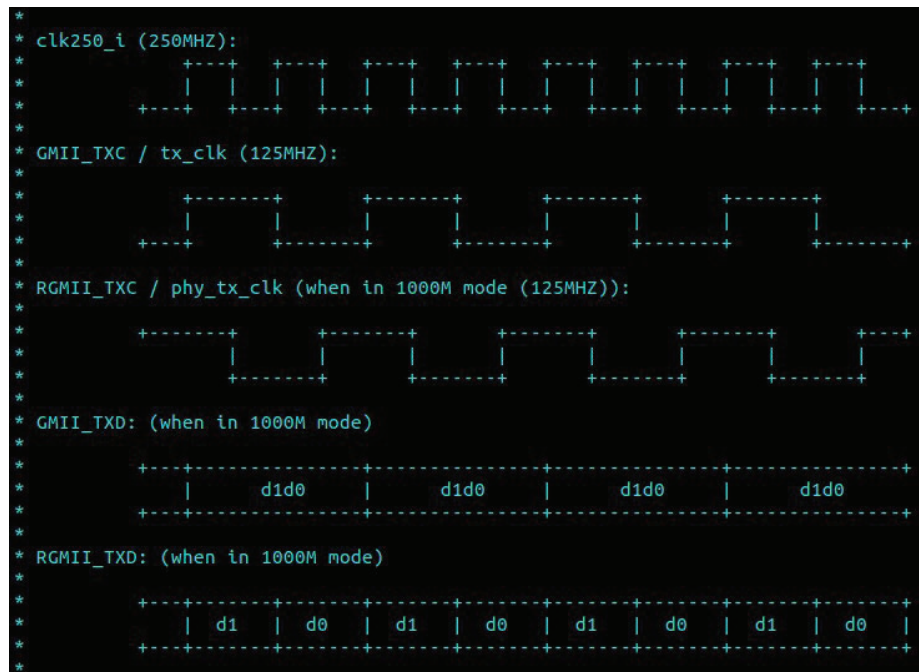


Figure 55: Waveforms of *rgmii_phy_if*

3.4.2.1 RISC-V PLIC from OpenTitan

A RISC-V PLIC module from lowRISC has been used in BlackParrot. The PLIC module almost works out-of-box on BlackParrot. Only a few custom changes have been applied to the module, for example, to workaround a synthesis bug appearing in an older Vivado version. The PLIC module was generated from the PLIC template in OpenTitan. Although the PLIC template can support up to 255 interrupt sources, for the system of BlackParrot plus Ethernet controller, only

two interrupt sources (Note that one of the two interrupt sources is reserved and not used, that is, source 0) and one target are generated.

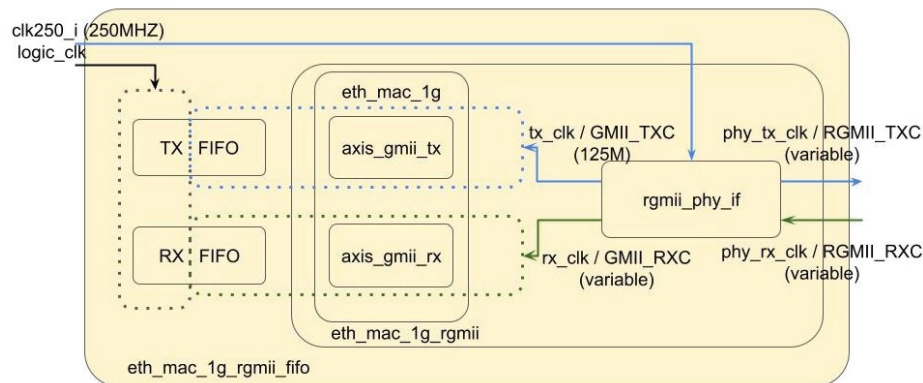


Figure 56: The Clock Domains of the MAC Module

3.4.2.2 Linux Liteeth Driver Memory Map

Figure 59 shows the memory map of a Liteeth device. The map consists of two parts, the TX/RX buffer region (starting from buf_base) and the register region (starting from reg_base). Note that in Liteeth driver, WRITER refers to the RX side of the Liteeth hardware; while READER refers to the TX side. Also, buf_base is not necessarily lower than reg_base. They can be any arbitrary locations.

3.4.2.2.1 The TX/RX Buffer Region

This region consists of a TX FIFO and an RX FIFO. They contain num_tx_slots and num_rx_slots slots respectively. Each slot has the same size and is able to store 1 packet. The current RX slot pointer is maintained in the hardware and can be read out through LITEETH_WRITER_SLOT. On the contrary, the current TX slot pointer is maintained in the driver, and needs to be written to ITEETH_READER_SLOT to inform the hardware the right location to send the packet.

3.4.2.2.2 Memory-Mapped Registers

Below lists only part of the memory-mapped registers that are used in the driver.

- LITEETH_WRITER_SLOT
Index of the current received packet.
- LITEETH_WRITER_LENGTH
Length of the current received packet.
- LITEETH_WRITER_EV_PENDING
Pending bit for RX. If set, the RX buffer is non-empty. Writing 1 to it clears the bit.
- LITEETH_WRITER_EV_ENABLE

Enable bit for RX. If set, the interrupt line of RX is asserted when LITEETH_WRITER_EV_PENDING is set.

- **LITEETH_READER_START**
If written to 1, the next packet in the TX buffer is transmitted.
- **LITEETH_READER_READY** If set, the TX buffer is full.

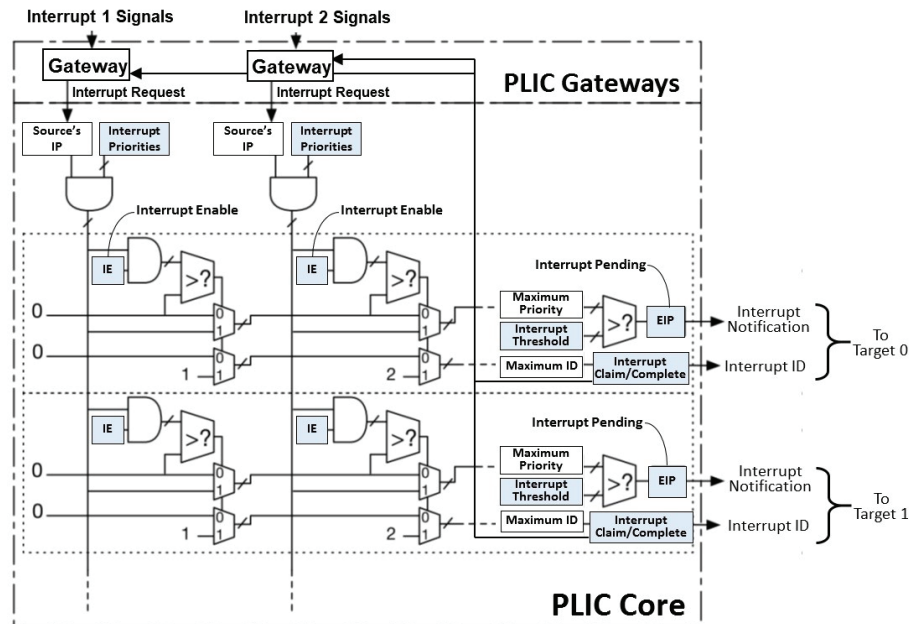


Figure 57: The PLIC Architecture [25]

- `LITEETH_READER_SLOT`
Index of the next transmitting packet.
- `LITEETH_READER_LENGTH`
Length of the next transmitting packet.
- `LITEETH_READER_EV_PENDING`
Pending bit for TX. If set, the TX buffer becomes free. Writing 1 to it clears the bit.
- `LITEETH_READER_EV_ENABLE`
Enable bit for TX. If set, the interrupt line of TX is asserted when `LITEETH_READER_EV_PENDING` is set.

3.4.2.3 Linux Interrupt Controller Driver

There are two RISC-V interrupt controller drivers in Linux kernel 5.15: the RISC-V Hart-Level Interrupt Controller [23] and the SiFive Platform-Level Interrupt Controller [24]. **RISC-V Hart-Level Interrupt Controller (HLIC)**

This controller handles the interrupts defined by the RISC-V specification: they are the software interrupt, timer interrupt and external interrupt. The use of this interrupt controller is specified in the device tree. The device node for HLIC should have the ‘compatible’ property set to “riscv,cpuintc”. As for connecting the interrupt lines of a device to the interrupt controller, “interruptsextended” can be used to specify those connections. For example, the example device tree in figure 60 shows a clint device connecting its M-mode software interrupt line (Number: 3) and M-mode timer interrupt line (Number 7) to a Hart-Level Interrupt Controller, called CPU0_intc.

When Linux registers an HLIC, the callback function, `handle_arch_irq`, within `handle_exception` (which is the entry point of the S-mode interrupt handling in Linux), is set to `riscv_intc_irq`, which is the interrupt handler in this controller, as demonstrated in figure 61. Besides, Linux will create an `irq_domain` data structure to keep the mapping between the hardware irq numbers (like 9 is S-mode external interrupt in a RISC-V hart) and their associated interrupt handlers during the HLIC registration. Linux will add entries to that data structure whenever it discovers a device connecting to the HLIC in the device tree. For instance, there is an HLIC, `cpu0_intc` in figure 60, therefore an `irq_domain` for the HLIC is created. There is also a PLIC device connected to `cpu0_intc`, so an interrupt handler provided by the PLIC device and the hardware irq number 9, are put into the `irq_domain` of `cpu0_intc`, as shown in figure 61. With this mapping, HLIC is able to find out the correct interrupt handler to call based on the passed hardware irq number.

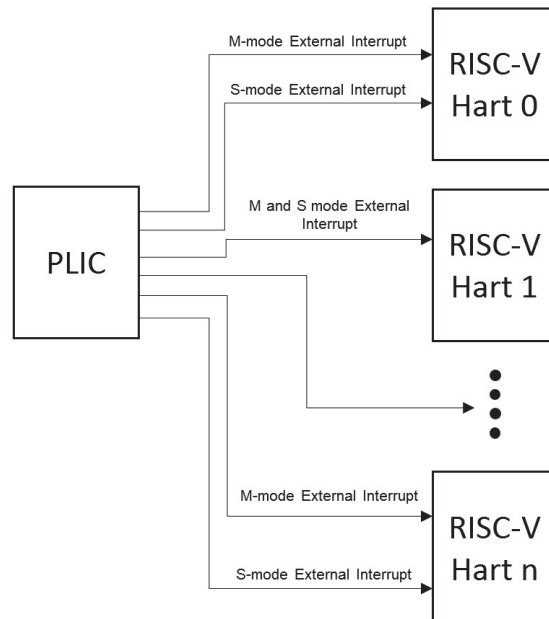


Figure 58: PLIC and RISC-V Harts [25]

3.4.2.4 SiFive Platform-Level Interrupt Controller (PLIC)

While HLIC handles the three standard RISC-V interrupts, this controller deals with the interrupts connected to a PLIC device. The use of this interrupt controller is specified in the device tree. The device node for PLIC should have the ‘compatible’ property set to “sifive,plic-1.0.0”. The ‘extendedinterrupt’ in figure 60 shows a PLIC interrupt controller, `plic`, which connects to the S-mode external interrupt line (number 9) of an HLIC interrupt controller,

CPU0_intc. The ‘reg’ property defines the location and the size of the memory-mapped IO of the PLIC device, in this case they are 0x20000000 and 0x4000000. Finally, the ‘riscv,ndev’ specifies the total number of interrupts supported in the PLIC device.

When Linux registers a PLIC, it not only creates an irq_domain for it, but also adds the mapping between the interrupt handler of PLIC, plic_handle_irq, and the hardware irq number to the irq_domain of its interrupt parent domain, in this case the irq_domain for CPU0_intc. What plic_handle_irq does is that it reads the claim register in the PLIC device to get the interrupt source number. Then it looks up the number in the plic’s irq_domain to find out the right interrupt handler to call.

Going Back to the device tree in figure 60, there is a Liteeth Ethernet device, ethernet, connected to the interrupt source 1 of plic, so the mapping between the Liteeth interrupt handler, liteeth_interrupt, and 1 is added to plic’s irq_domain. Note, however, instead of being called directly by plic_handle_irq, liteeth_interrupt is called by handle_fast_eoi, which sends an End of Interrupt signal to PIC (essentially a write to the PLIC completion register) after the liteeth_interrupt call. This completes the PLIC interrupt handling.

3.4.3 Why not connecting Liteeth Driver to HLIC

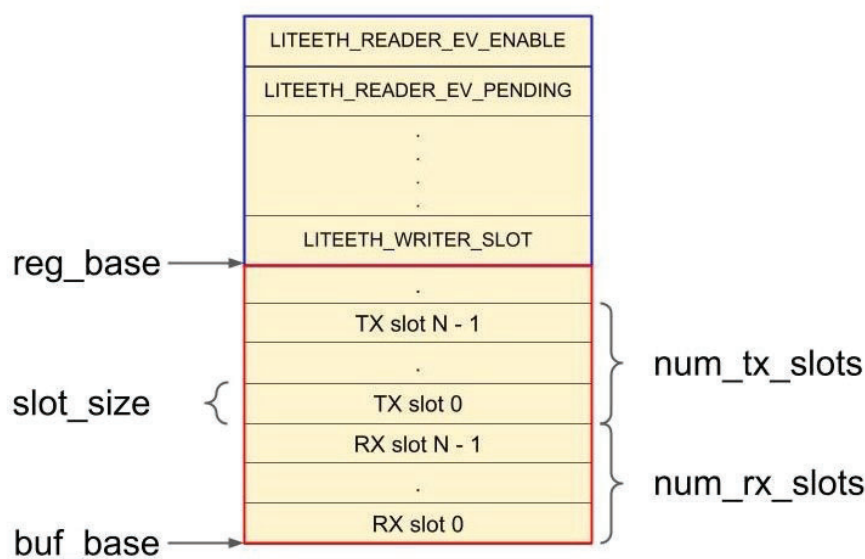


Figure 59: The Liteeth Memory Map

Possibly the simplest way of combining a Liteeth Ethernet controller with a RISC-V core is to just bind the interrupt line of the Ethernet controller directly to an external interrupt line of the core. From the hardware perspective, this is doable if the Ethernet controller is the only external device in the system. However, in Linux, the Liteeth driver cannot work well with the HLIC driver.

When the HLIC driver gets registered, an irq_domain is created for it. One of the domain operations (to be specific, the domain map operation) of HLIC’s irq_domain will only allow interrupt handlers that are marked as ‘percpu’ to be mapped and added to its irq_domain. Since

the interrupt handler from Liteeth (liteeth_interrupt) is not 'percpu', the registration of the Liteeth device will fail. The PLIC driver, on the other hand, does not have such an issue because its interrupt handler is 'percpu' and its irq_domain allows interrupt handlers without 'percpu' to be added to its irq_domain. Therefore, from the software perspective, the Liteeth driver needs to connect to the PLIC driver first, and only with that the PLIC driver can then connect to the HLIC driver. This is one of the reasons why a PLIC device is necessary in our work if one of the goals is to use everything as is.

3.4.4 Percpu Interrupt Handler

A 'percpu' interrupt handler can be called on different CPUs at the same time, without the need of any locking mechanism. Unlike the typical interrupt handler, 'percpu' interrupt handler has per-cpu storage. In other words, each CPU core will have its own copy of data. When any of the CPUs executes a 'percpu' interrupt handler, it will first figure out what its CPU ID is and it will use that ID to get the location of its per-cpu storage, on which the rest of the handler will then operate.

Therefore no race condition will occur in a 'percpu' interrupt handler.

'percpu' interrupt handlers handle CPU-related tasks. For example, the local timer interrupt handler increments the tick counts of the running core; the PLIC interrupt handler accesses the PLIC memory-mapped registers of the running core. In contrast, 'percpu' interrupt handlers are not suitable for tasks such as receiving Ethernet packets, as it is CPU-agnostic and does not behave differently when running on different CPUs.

3.4.4.1 The RX/TX Path in Linux The RX Path

After the Ethernet controller receives an Ethernet packet, it asserts the interrupt line to PLIC, which then sends an interrupt notification onto the S-mode external interrupt line of BlackParrot.

```

cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    timebase-frequency = <2500000>;
    CPU0: cpu@0 {
        device_type = "cpu";
        reg = <0>;
        status = "okay";
        compatible = "riscv";
        riscv,isa = "rv64imafdc";
        mmu-type = "riscv,sv39";
        clock-frequency = <20000000>;
        CPU0_intc: interrupt-controller {
            #interrupt-cells = <1>;
            interrupt-controller;
            compatible = "riscv,cpu-intc";
        };
    };
};
soc {
    #address-cells = <2>;
    #size-cells = <2>;
    compatible = "ucbbar,spike-bare-soc", "simple-bus";
    ranges;
    clint@300000 {
        compatible = "riscv,clint0";
        interrupts-extended = <&CPU0_intc 3 &CPU0_intc 7>;
        reg = <0x0 0x300000 0x0 0xc0000>;
    };
    plic: interrupt-controller@20000000 {
        #address-cells = <0>;
        #interrupt-cells = <1>;
        compatible = "sifive,plc-1.0.0";
        interrupt-controller;
        interrupts-extended = <&CPU0_intc 9>;
        reg = <0x0 0x20000000 0x0 0x4000000>;
        riscv,ndev = <1>;
    };
    ethernet@10001000 {
        compatible = "liteeth,liteeth";
        local-mac-address = [12 34 56 78 9a bc];
        reg=<0x0 0x10001000 0x0 0x100>,
            <0x0 0x10000000 0x0 0x1000>;
        reg-names="mac", "buffer";
        liteeth,rx-slots = <1>;
        liteeth,tx-slots = <1>;
        liteeth,slot-size = <0x800>;
        interrupts-extended = <&plc 1>;
    };
};

```

Figure 60: A Device Tree Example for the RISC-V Interrupt Controllers

After getting the notification, BlackParrot is trapped into the S-mode external interrupt handler, `handle_exception`, in Linux (because opensbi has delegated the S-mode external interrupt to Linux), which then calls `riscv_intc_irq` with interrupt number 9. `riscv_intc_irq` looks up the number in its `irq_domain` structure and finds out the corresponding handler, `plc_handle_irq`. `plc_handle_irq` gets called and reads the claim register in PLIC and gets the interrupt source number, which is 1 in this case. The handler then again looks up the number in its `irq_domain` and finds out the corresponding handler, which is `handle_fasteoi_irq`. This function gets called and then it calls `liteeth_interrupt`, which is the interrupt handling process for the Ethernet. After `liteeth_interrupt` is done, `handle_fasteoi_irq` sends an EOI (essentially a write to the completion register) to PLIC and ends the entire receive flow.

3.4.5 The TX Path

Whenever the Linux kernel gets a packet to transmit for the Liteeth driver, it calls `liteeth_start_xmit`. Note the transmission can be stopped temporarily if the kernel previously found that the TX buffer from the Ethernet controller was full during its last transmission. The kernel will resume the transmission after getting an interrupt indicating that the TX buffer has become available.

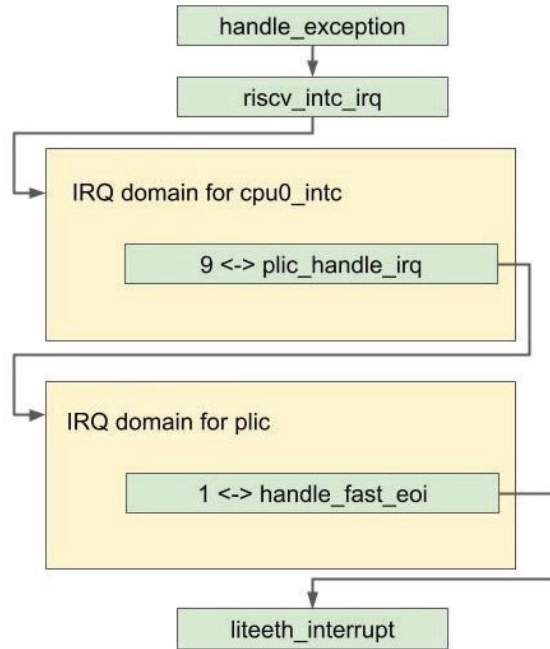


Figure 61: The Linux IRQ Domains for HLIC and PLIC

3.4.6 lwIP

3.4.4.1 Scope The target is embedded applications with small memory/code footprint without operating systems. The final work should be able to run on the BlackParrot Ethernet controller and the RISC-V PLIC module, and support TCP, UDP and IPv4.

3.4.6.1 Introduction

lwIP is a lightweight implementation of many well-known network protocols in layer 3 and 4, such as TCP, UDP, IP(both version 4 and 6) and DHCP. lwIP also provides device drivers for some common layer 2 protocols, for example, the Ethernet protocol. Figure 62 shows the overview of an lwIP application running on the Ethernet: lwIP is located in layer 3, 4 of the OSI model, and the device driver serves as an interface between lwIP and the underlying physical media.

Depending on the use cases, lwIP can either run in the OS mode (with an operating system) or in the mainloop mode (without an OS). The OS mode supports sequential API that is similar to the Linux socket API, while the mainloop mode supports raw API in which the program execution is driven by callback functions. Since our main target is embedded applications without operating systems, the mainloop mode is the main focus.

One of the major differences between the mainloop mode and the OS mode is the preemption of the task. In the OS mode, the tasks spawned by the user program do not have to yield themselves to the other tasks because the OS will take timer interrupts to decide if a task should be preempted and replaced with another task. On the other hand, in the mainloop mode, the tasks have to yield themselves from time to time in order not to block the entire system. This would make programming in the mainloop mode more challenging. Figure 63 shows the typical

programming strategy for the mainloop mode. After the system initialization, the program enters an infinite loop and executes each task in order. Tasks are implemented as state machines so that all tasks are able to resume their previous work in every iteration.

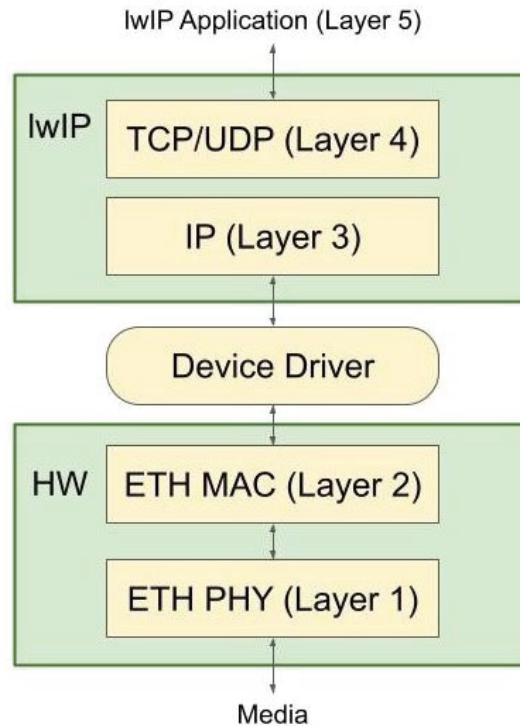


Figure 62: lwIP Overview

3.4.6.2 Components

Figure 64 shows the components of lwIP. TCP and UDP connections are described by `tcp_pcb` structs and `udp_pcb` structs, respectively. Each lwIP network interface, which is described by a `netif` struct, has its own IP address. Based on the destination IP address, the TCP / UDP traffic is routed to an interface by the IP module. Each interface also has its own configurations, for example, the MTU size, the hardware address of the underlying hardware, and functions used by its associated device driver to send / receive the packets to / from lwIP. The boundary between lwIP and the device driver is `netif->input` and `netif->output` (in the case of IPv4). `netif->input` is called by the driver to send packets in lwIP, while `netif->output` is called by the interface to send packets out lwIP. Although not shown in the figure, lwIP uses `pbuf` structs to pass the packet/payload between different components. Each `pbuf` can point to one chunk of memory. Multiple `pbuf`s can form a singly-linked list to store a piece of information that is not contiguous in the memory.

3.4.6.3 Flow

lwIP uses callback-style API in the mainloop mode to notify the application of the happening events. The below pseudo code gives the basic idea of how the API works. Before entering the main loop, the program registers callback functions for different events, such as the TCP data received event or the error notification event. In the main loop, the program does the following

three things: 1. Check if there is any received packet. If yes, send it to lwIP through a netif. 2. Call the lwIP housekeeping function, *sys_check_timeouts*, to take care of all the background protocol operations 3. Call the user task. *sys_check_timeouts* checks the current time and calls a timeout handler if any timer for a network protocol expires, for example, resending a TCP packet when lwIP does not receive an ACK in time. *sys_check_timeouts* should be called from time to time from the main loop, otherwise lwIP will not be able to handle the expired timeout handler properly.

```
void
callback1(void);
void
callback2(void);
```

```
int main () {

    sys_init(); // System Initialization

    while(1) {
        task1(); // state machine 1
        task2(); // state machine 2
        task3(); // state machine 3
    }

}
```

Figure 63: Mainloop

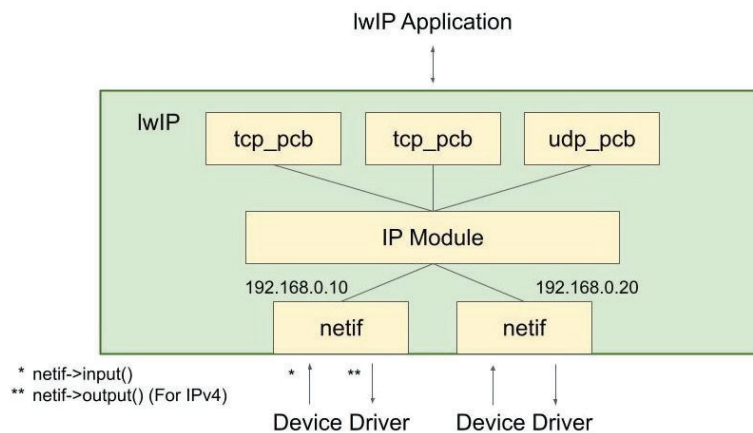


Figure 64: lwIP Components

```
void

callback3(void);

struct netif

netif;

int main() {
    lwip_register_event1_callback(callback1);
    lwip_register_event2_callback(callback2);
    lwip_register_event3_callback(callback3);
```

```

while(1) { receive_packet(&netif);
    sys_check_timeouts(); // the lwIP housekeeping
    function user_task();
}
}

```

3.4.6.4 Dynamic Memory Allocation

lwIP has two dynamic memory allocators, which are the heap memory allocator and the memory-pool allocator. Their corresponding functions are *mem_malloc/mem_free* and *memp_malloc/memp_free*. Both the heap memory and the memory pools are simply statically-allocated memory and their total sizes can be specified in the lwIP configuration.

An lwIP memory pool is a chunk of memory that consists of several memory blocks, each of which has the same size. Figure 65 shows an example of memory pools in lwIP. The yellow bars are the memory pools, and the green boxes are the descriptors for the memory pools. Using the lwIP memory pool allocator has two advantages: 1. The allocation and free operation takes only constant time, since each free memory block in a memory pool is put into the head of a singly-linked list. 2. It avoids memory fragmentation, because each memory pool only holds objects with the same size.

For *mem_malloc/mem_free*, lwIP provides three implementations. The first one (the default one) allocates various sizes of memory chunks from the lwIP heap. The second one simply calls an external memory allocator provided by the user, which can reduce code size if there is already a memory allocator in the user's library (like *malloc* from Newlib). The third one uses the *memp_malloc/memp_free* as its backend. It will search from the memory pool with the smallest block size and return the allocated memory once it finds a block that is just big enough for the user's memory request.

For *memp_malloc/memp_free*, they can be configured to allocate from the lwIP memory pools or allocate memory with *mem_malloc/mem_free*. The latter is useful when *mem_malloc/mem_free* are also configured to the second implementation mentioned in the previous paragraph, as the entire program will then only have the user-provided memory allocator, which could save some code footprint.

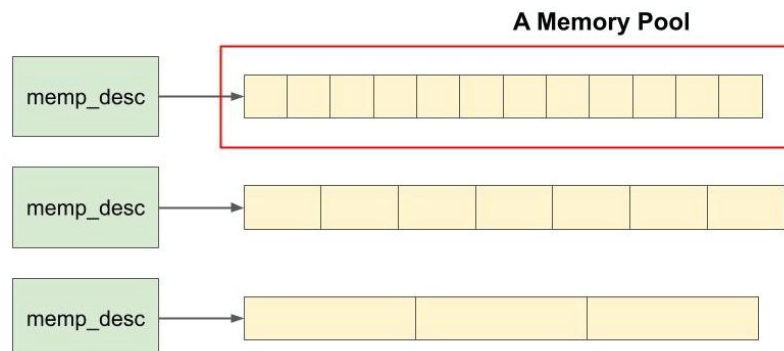


Figure 65: lwIP Memory Pools

3.4.6.5 Porting

The actual porting process for lwIP depends on several factors, including the toolchain being used, the underlying networking device, and the features that users would like to have. For the toolchain and the networking device, the GNU C compiler, the Newlib C library, and an Ethernet controller have already been ported to BlackParrot. For the lwIP features, there will be no OS in our applications so only the mainloop mode is considered. Below gives the list of what has been done for the porting to the BlackParrot platform:

3.4.6.5.1 Implementing `sys_now`

This function is necessary for the timeout features in lwIP to work. In the case of BlackParrot, *gettimeofday* from BlackParrot Newlib is used to implement this function.

3.4.6.5.2 Creating `arch/cc.h`

If users would like to overwrite the default settings in `lwip/arch.h`, which defines macros for different processors and compilers, they should provide their own macros in this user provided file, `arch/cc.h`, which is included at the very beginning of `lwip/arch.h`. Below list some examples of the macros defined in `lwip/arch.h`: macro that specifies the byte order of the processor (little endian), macro that defines the platform specific diagnostic output function (`printf`), and macros that specify the mappings between the lwIP's data types to the compiler's data types.

In the case of BlackParrot, the macro `LWIP_DECLARE_MEMORY_ALIGNED` has been redefined in this file to leverage the alignment attribute from the GNU C compiler to declare an aligned byte array.

3.4.6.5.3 Creating `lwip/lwipopts.h`

If users would like to overwrite the default settings in `lwip/opt.h`, which defines various kinds of features options, such as options for TCP, UDP, IP protocols, options for memory allocation, and options for debugging and statistics, they should provide their own options in this file.

`lwip/lwipopts.h` plays a crucial role in lwIP tuning, and the actual settings in this file highly depend on the actual program to be run. For example, if the program needs high performance, options that tell lwIP to use memory pool allocation in *mem_malloc* can be put in this file; on the other hand, if the program needs ultra low memory footprint, options that disable unused features/modules and options that allocate fewer memory can also be put in this file. For tuning the memory footprint of the lwIP memory allocators, `MEM_STATS` can be set to 1 in `lwipopts.h` and the function *stats_display* can be used to dump the memory usage at any specific moment during runtime.

3.4.6.5.4 Implementing Ethernet device drivers

Each network interface in lwIP is described by a `netif` structure, which includes the following three callbacks:

- `input`: send the received packet to lwIP.

- output: resolve the hardware address of the IPv4 packet and send the packet out to the network device.
- linkoutput: send out the layer-2 packet to the network device. This callback only needs to be set when output requires it.

The above callbacks are the boundary between lwIP and the device driver. In the case of Ethernet, lwIP already has built-in functions, *netif_input* and *etharp_output*, for input and output respectively, so that the user does not need to re-implement all the callbacks. The only callback the user needs to implement is linkoutput since this will be called within *etharp_output*. Besides these callbacks, the user also has to implement their device driver which will communicate with the hardware and interact with the lwIP interface with the above callbacks. For BlackParrot, the built-in Ethernet functions mentioned above are used and linkoutput is set to one of the BlackParrot Ethernet driver functions, *ethernet_send*.

Note that in the mainloop mode, the porting of synchronization primitives such as semaphores and mutexes is not required as all the tasks will not be preempted by each other, and there can only be one task active at a time. Although the interrupt handler that receives the packet can preempt a task, a simple lock-free mechanism has been used to allow safe concurrent data access.

3.4.6.6 Introduction

wolfSSL is a lightweight TLS library suitable for embedded systems. It supports up to TLS 1.3, which is the latest TLS version. Although nowadays TLS and SSL are often used interchangeably to mean the TLS protocol, they are actually two different protocols: For TLS, there are TLS 1.0 to TLS 1.3, whereas for SSL, there are SSL 1.0 to SSL 3.0, which have already been replaced by TLS. In this report, these two terms will continue to be used interchangeably to mean the TLS protocol.

Like TCP, an SSL client will initiate a handshake at the beginning of an SSL connection to an SSL server. One of the steps happening in the SSL handshake is authenticating the server's identity. The main purpose of having the server authentication is to make sure that the client is really connecting to the server that says who it is. Without the authentication, the server can claim to be anyone on the Internet without being noticed by the client.

The authentication process involves an important piece of data called, certificate. Figure 66 shows how a certificate is generated and used in an SSL connection. A certificate, issued by a Certificate Authority (CA), contains the owner's public key and the owner's identity. The issue of a certificate can be described with two steps: First, CA receives a certificate signing request (CSR) from the organization that plans to have the certificate. A CSR includes information like the legal name, email address and the physical address (city, state, country...) of the organization, and a public key. Second, after CA verifies the information on the CSR, it creates a certificate and signs it with its private key. The above steps are how the server and CA certificate are made.

With the server certificate, the server can now show its identity by sending its certificate to its client during the SSL handshaking. After receiving the server certificate, the client starts the

server authentication: First, it figures out which CA signed the server certificate. Then it searches its trusted CA certificate list (pre-installed in the client) to find the CA's public key. Finally, the client uses the CA's public key to verify if the server certificate has really been verified and signed by the CA mentioned in the server certificate. If yes, then the server certificate certifies the mapping between the owner's public key and the owner's identity.

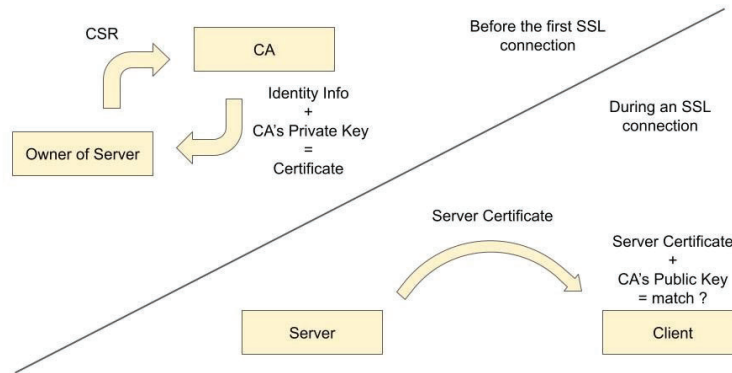


Figure 66: How an SSL Certificate Works

3.4.6.7 Porting

The actual porting of wolfSSL depends on the use case and the available software/hardware resources. For BlackParrot, BlackParrot Newlib and the lwIP library are available. This section describes the actions taken to port wolfSSL to BlackParrot with those libraries.

- **WRITEV**
By default, wolfSSL includes `<sys/uio.h>` and provides `wolfSSL_writev` to the program. Since the header is not supported in BlackParrot Newlib, this feature is disabled by defining `NO_WRITEV`.
- **INPUT / OUTPUT**
wolfSSL uses the callbacks `CBIORcv` and `CBIOSend` to interact with the underlying transport layer. By default, those two callbacks are set to `EmbedReceive` and `EmbedSend`, which defaults to using the `recv` and `send` from a BSD socket interface. In order to integrate lwIP into wolfSSL, `WOLFSSL_USER_IO` is defined and the callbacks `CBIORcv`, `CBIOSend` are set to `ssl_media_send` and `ssl_media_recv`, which interact with lwIP.
- **FILESYSTEM**
Although there is lightweight filesystem support in BlackParrot Newlib, the filesystem support in wolfSSL is disabled by passing `-disable-filesystem` to `./configure` to achieve a smaller code size.
- **TIME**
Although BlackParrot Newlib already has a time function, `gettimeofday`, some modification to the time function is still needed for wolfSSL. In SSL, current time is used to validate certificates: Each certificate has `notBefore` and `notAfter` fields, and the current time has to

fall within this time window to pass the validation. By the time the wolfSSL was porting to BlackParrot, there was no real time clock in BlackParrot that could keep the current time. Therefore, in order to work around this timing issue, the *gettimeofday* implementation in BlackParrot Newlib was changed to return an Epoch time that starts from a fixed time base, for example, June 1, 2021 12:00:00 AM. The expiration date of the demo certificates (which is located in the folder `tcp_ssl/ssl_cert_build/`. This can be found in section 4.4.1) were set accordingly.

- **RANDOM SEED**
By default, wolfSSL uses `/dev/urandom` or `/dev/random` to retrieve random numbers. Since no such device files exist in our environment, `NO_DEV_RANDOM` is defined and the *rand* function from BlackParrot Newlib is used instead.
- **THREADING**
Since there will only be one task active in the system, and tasks will not preempt with each other, `-enable-singlethreaded` is passed to `./configure` to negate the need to port the wolfSSL mutex layer.

3.4.6.8 Boot Loaders for BlackParrot System on Module

A BlackParrot System on Module (BP SOM) consists of only a BlackParrot core, an Ethernet controller, a piece of tiny ROM and RAM. This lightweight design leads to low cost and little power consumption and is suitable for embedded applications. However, if users would like to run a Linux distribution on a BP SOM, the software booting process would be challenging as there would be no hard drive storing the Linux kernel and all the other system data. What makes things even more difficult is that the tiny ROM in the BP SOM should be made as small as possible, possibly no larger than 64 KiB, which is too small for holding a regular boot loader. In order to address the above issues, two boot loaders along with the Linux Network File System (NFS) have been introduced to the booting process.

Figure 67 shows the booting process of a Linux distro on the BP SOM. The middle row shows the current program running in the BP SOM at a specific stage, while the bottom row shows the three different servers. Since the first-stage boot loader is stored in ROM, the booting process starts by relocating the first-stage boot loader from ROM to RAM. Then the first-stage boot loader downloads the second-stage boot loader through a protocol called Trivial File Transfer Protocol (TFTP). TFTP is a client server protocol: the client sends read or write requests to the server that hosts the files. TFTP is also a lightweight protocol running on UDP. It has been frequently used in network booting as it can be implemented with a very small memory footprint. After downloading the second-stage boot loader, the second-stage boot loader downloads the Linux kernel from the SSL server and executes it. At the end of execution, the Linux kernel does the NFS mounting: it mounts the root directory of the Linux distro from the NFS server as its root. Now every access to the root file system happens on the network, as if the root file system were stored on a local hard drive. Finally, the kernel runs an init script in the root to boot the Linux Distro.

The purpose of passing the Linux kernel through an SSL connection is to ensure the kernel image is not corrupted by any malicious attack. The reason why the SSL client is not used as the

first-stage boot loader is that the code size for both the SSL and TCP (SSL typically runs on TCP) layers is too large to fit into a 64-KiB ROM. Therefore a more lightweight protocol, TFTP, is chosen for the first stage instead. In order to make sure that the second-stage loader passed by the TFTP server will not be corrupted, a SHA256 checksum of the loader is written in ROM beforehand with the TFTP client program. This allows the TFTP client program to discover any errors by comparing the checksum of the received loader against the pre-calculated checksum.

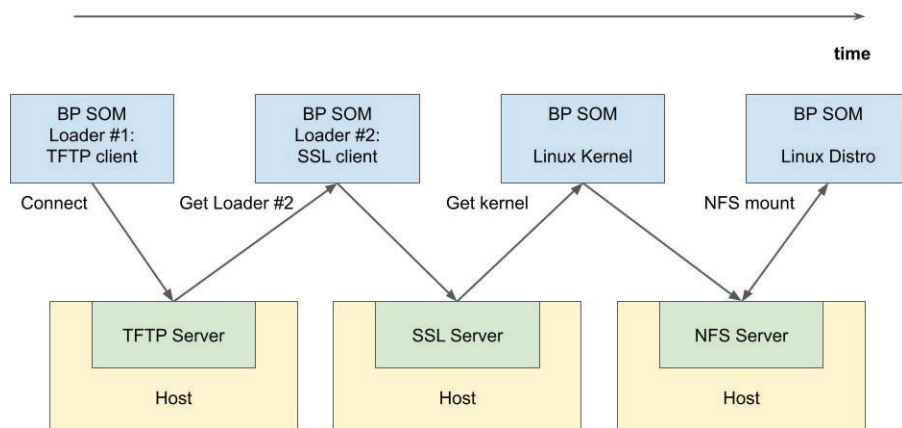


Figure 67: The Booting Process of a Linux Distro on the BlackParrot System on Module

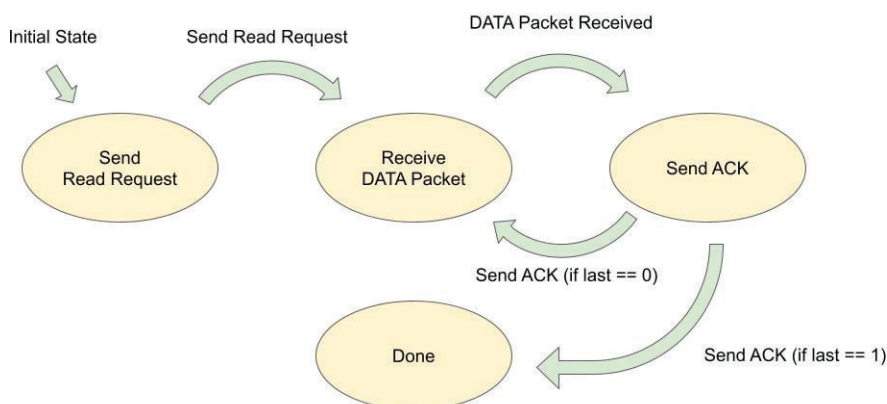


Figure 68: The FSM within the TFTP Loader

In order to minimize the code size of the TFTP loader as much as possible, the following four steps have been taken for the TFTP loader:

1. Eliminate the use of BlackParrot Newlib.
2. Remove unnecessary features, for example, TCP protocol and IP fragmentation, by specifying options in lwipopts.h. IP fragmentation is unnecessary because the TFTP blocksize used in the loader is smaller than the maximum size a UDP packet can have without fragmentation.

3. Remove the code of the internal lwIP heap memory allocator by reusing the existing lwIP memory pool allocator. As described in the previous section, there are several implementations for the lwIP heap memory allocator. One of the implementations is to use the lwIP memory pool allocator for the lwIP heap memory allocation. This can be done by setting MEM_USE_POOLS to 1 in lwipopts.h, and defining a configuration file lwippools.h that specifies the schema of the memory pools. For example, define two memory pools, one with 50 blocks, 64 bytes for each, and the other with 25 blocks, 88 bytes for each. By reusing the code, the total code size is further reduced.
4. Implement only necessary features of the TFTP protocol. In our case, support only the read requests, as the TFTP loader is only required to read a file from the server.

3.4.7 The SSL Loader

Similar to the first-stage boot loader, the second-stage loader also consists of two parts: the SSL client that loads the Linux binary and the program loader that loads the Linux binary to where it should be. The SSL client part of the boot loader uses the SSL client in the tcp_ssl example as the template. Therefore the description is omitted here.

Table 21: ZynqParrot Utilization

Resource	Utilization	Available	Utilization %
LUT	38921	53200	73.16
LUTRAM	3386	17400	19.46
FF	20207	106400	18.99
BRAM	123.5	140	88.21

3.4.8 Validation and Results

The combination of BlackParrot, BlackParrot Ethernet Controller and the RISC-V PLIC module has been verified both in software simulation and on Zedboard, under the ZynqParrot framework. Figure 69 shows the final block diagram with all the modules integrated. The ETH block is the BlackParrot Ethernet controller with an AXI client (slave) adapter, whereas the PLIC block is the RISC-V PLIC module from OpenTitan with an AXI master adapter and an AXI client adapter. Although not shown in the figure, the RGMII interface of the Ethernet controller is connected to the FMC interface on Zedboard, and the TX and RX IRQ line of the Ethernet controller are ORed together before connecting to the interrupt source 1 of the PLIC module. The clock frequencies for the BlackParrot, the Ethernet Controller (except the RGMII part of the circuits), the PLIC module and the AXI connections are all 20 MHZ.

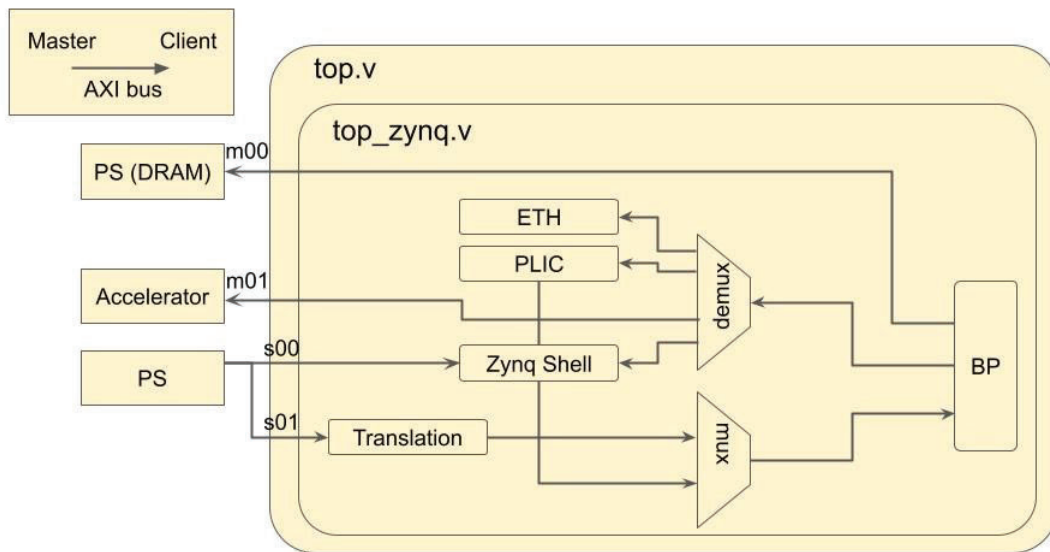


Figure 69: ZynqParrot with Ethernet Controller and PLIC

3.4.8.1 FPGA Synthesis Reports Zedboard Utilization

Table 21 shows the total utilization of BlackParrot, Ethernet controller and PLIC module and other modules under ZynqParrot. Table 22 and table 23 show the relative utilization compared to the total of ZynqParrot.

Table 22: The BlackParrot Ethernet Utilization

Resource	Utilization	ZynqParrot %
LUT	849	2.18
LUTRAM	0	0.00
FF	981	4.85
BRAM	9.5	7.69

Table 23: The RISC-V PLIC Module Utilization

Resource	Utilization	ZynqParrot %
LUT	3	0.01
LUTRAM	0	0.00
FF	16	0.08
BRAM	0	0.00

3.4.9 Quality of the RX RGMII Signals

Even if the BlackParrot Ethernet controller supports 1G Ethernet, without having good quality of the RGMII signals, the speed would be severely degraded. The Ethernet controller does Cyclic Redundancy Check (CRC) whenever it receives a packet. If there is data corruption in the packet, possibly due to some noises in the Ethernet cable or improper latch of some RX RGMII signals, the hardware will simply discard the packet and result in a packet loss.

The original MAC module from Alex already provides signals that would indicate whether such a packet loss is happening on the RX side. Those signals have been used during the software tests (which will be described in the next section) to evaluate the quality of the RX RGMII signals. The end results showed that there was no such packet loss on the RX side. For the quality of TX RGMII signals, it is hard to show if there is data corruption happening there, since there is no feedback signal from the receiver that would indicate the data has been corrupted.

3.4.10 Linux Kernel 5.15 (with Liteeth driver and the RISC-V PLIC driver)

After booting up the Linux kernel on Zedboard, the Ethernet connection was tested by running a TCP server written in C that sends 4 MiB of data to a host. The Ethernet connection was also tested by doing a Linux NFS boot. First, an NFS server that would provide the root directory of a Linux distro from Yocto was launched on a host computer. Then the Linux kernel began to boot on Zedboard.

3.4.10.1.1 TCP/SSL Examples

During the test for TCP/SSL server, multiple TCP/SSL client programs written in Python connected to the server on Zedboard one by one and then they sent/received messages to/from the server at the same time. The clients also connected and disconnected multiple times to ensure the server could handle the disconnection properly.

During the test for the TCP/SSL client, a TCP/SSL server written in Python launched on a host first and then the TCP/SSL client on Zedboard sent data to the server.

3.4.10.1.2 The BlackParrot System on Module

The complete boot flow was tested on Zedboard: The first-stage boot loader loaded the second-stage boot loader from a busybox TFTP server running on a host and verified its SHA256 checksum. The second-stage boot loader then ran and loaded the Linux binary from a python SSL server running on a host. After booting up the Linux kernel, the kernel started NFS booting from an NFS server on a host to boot the Linux Distro from Yocto.

3.4.10.2 Software Footprint lwIP Library Footprint

Figure 70 shows the breakdown of the lwIP library footprint. The data in the figure was generated by the following command ‘size -t’. Each column, from left to right, shows the text section, the data section, the bss section and the total size of the previous three sections in decimal and hexadecimal. Each row shows the size of each component in the lwIP library, and the total sizes can be seen at the bottom row. The library has been compiled by “gcc” with “-Os”

and without RISC-V compressed instruction set (as BlackParrot currently does not support the RISC-V C Extension). It has also been compiled with “-fno-common”, so that the uninitialized data (for example, the static memory managed by the two lwIP memory allocators) is put in the bss section.

This library has been tuned for the TCP/SSL programs from the first example (and assuming that the server program can only make up to 4 TCP connections at the same time). The total text size of the library is roughly 79.5KiB, and the total size of data and bss is roughly 3.9KiB. The two main sources for the bss section are the static memory from the lwIP heap allocator (mem.o) and the static memory from the lwIP memory pool allocator (memp.o). The MEM_SIZE macro that specifies the maximum size of the lwIP heap, and the MEMP_NUM_* macros that specify the maximum numbers of elements in each of the lwIP memory pool, have been specified in lwipopts.h to minimize the sizes of the two bss sections. Those memory limits should be set properly based on the actual applications. They can be found by setting MEM_STATS to 1 in lwipopts.h and dumping the memory stats with *stats_display*.

In our specific setting, the dynamic memory footprint increases 216 bytes per TCP connection, as each TCP connection is represented by a TCP control block, and the control block is 216 bytes.

wolfSSL Library Footprint Figure 71 shows the breakdown of the wolfSSL library footprint. Similar to the lwIP library, the data from the figure was generated by the following command ‘size-t’. The library has been compiled by “gcc” with “-Os” and without RISC-V compressed instruction set. It has also been compiled with “-fno-common”. The total text size of the library is roughly 313.9 KiB, and the total size of data and bss is roughly 5.7 KiB.

With one SSL connection, the dynamic memory footprint is roughly 14.8 KiB. With more than one SSL connection, the dynamic memory footprint increases 3.1 KiB/s per SSL connection.

The TFTP Loader Footprint Figure 72 shows the footprint of the TFTP loader. The size of the text section is roughly 30.8 KiB (this amount of code will be put in the ROM). The bss section is roughly 4.1MiB. 4MiB is allocated for storing the second-stage boot loader.

3.4.10.3 Network Throughput

The network throughputs were measured on Zedboard with BlackParrot running at 20 MHZ. The Ethernet speed ran at 1Gbps and was connected to a 1G switch. There was also a Linux host computer connected to the switch.

3.4.10.4 lwIP UDP Throughput

With continuous calls to *udp_send* with 1472 bytes of data (This is the maximum size one can get to send out UDP packets without fragmentation) and *PERIODIC_TASK*, the TX UDP throughput is roughly 1707 KiB/s.

When the host sends UDP packets with size 1472 bytes continuously, the RX UDP throughput is roughly 1791 KiB/s.

3.4.10.4.1 lwIP TCP Throughput

When transmitting files between the host, the RX TCP throughput is roughly 918.0 KiB/s, while the TX TCP throughput is roughly 1024 KiB/s.

text	data	bss	dec	hex	filename
1024	0	240	1264	4f0	stats.o (ex liblwip.a)
60	0	0	60	3c	time.o (ex liblwip.a)
72	0	0	72	48	init.o (ex liblwip.a)
1544	0	1928	3472	d90	memp.o (ex liblwip.a)
11422	6	44	11472	2cd0	tcp.o (ex liblwip.a)
0	0	0	0	0	alttcp_alloc.o (ex liblwip.a)
0	0	0	0	0	dns.o (ex liblwip.a)
2291	0	1064	3355	d1b	mem.o (ex liblwip.a)
1442	0	16	1458	5b2	timeouts.o (ex liblwip.a)
0	0	40	40	28	ip.o (ex liblwip.a)
532	0	0	532	214	def.o (ex liblwip.a)
0	0	0	0	0	raw.o (ex liblwip.a)
11342	0	96	11438	2cae	tcp_in.o (ex liblwip.a)
6409	0	1	6410	190a	pbuf.o (ex liblwip.a)
2739	0	24	2763	acb	netif.o (ex liblwip.a)
0	0	0	0	0	sys.o (ex liblwip.a)
0	0	0	0	0	alttcp.o (ex liblwip.a)
1372	0	0	1372	55c	inet_chksum.o (ex liblwip.a)
0	0	0	0	0	alttcp_tcp.o (ex liblwip.a)
3652	2	8	3662	e4e	udp.o (ex liblwip.a)
12007	0	0	12007	2ee7	tcp_out.o (ex liblwip.a)
5231	0	16	5247	147f	ip4_frag.o (ex liblwip.a)
9568	4	56	9628	259c	dhcp.o (ex liblwip.a)
4758	0	401	5159	1427	etharp.o (ex liblwip.a)
1176	0	16	1192	4a8	ip4_addr.o (ex liblwip.a)
0	0	0	0	0	igmp.o (ex liblwip.a)
2385	0	2	2387	953	ip4.o (ex liblwip.a)
0	0	0	0	0	autoip.o (ex liblwip.a)
1606	0	0	1606	646	icmp.o (ex liblwip.a)
826	0	0	826	33a	ethernet.o (ex liblwip.a)
81458	12	3952	85422	14dae	(TOTALS)

Figure 70: lwIP Library Footprint

3.4.10.4.2 wolfSSL Throughput

When transmitting files between the host, the RX SSL throughput is roughly 78.3 KiB/s, while the TX SSL throughput is roughly 50.8 KiB/s.

3.4.10.4.3 Linux Throughput

When sending data to the host through the Linux TCP socket, the TX throughput is roughly 331.1KiB/s.

3.4.10.4.4 Linux NFS Throughput

When running “dd if=/dev/zero of=test_file bs=X count=Y”, where, $X * Y$ equals to 2MiB (that is, the total size is fixed to 2MiB), the throughput are as follows:

- X=512: 59.9 KiB/s
- X=1024: 74.7 KiB/s

- X=2048: 82.4 KiB/s
- X=4096: 86 KiB/s
- X=8192: 90 KiB/s
- X=16384: 97.6 KiB/s
- X=32768: 95.4 KiB/s

The ideal throughput of 1G Ethernet should be 125 MB/s (1 Gbps / 8 bit). However, since ZynqParrot is running at 20 MHz and the AXI bus to the Ethernet controller is 32-bit, the ideal throughput under this hardware setting can only be 80 MB. Other factors could also further reduce the throughput: The first one is the DEMUX module. The module currently does not support multiple outstanding AXI requests, that is, it can only handle requests one by one. Therefore, the cycles needed to handle one single request is equal to the roundtrip latency between the DEMUX and the Ethernet controller, which is 4 cycles. Therefore the throughput reduces from 80 MB to 20 MB. The second one is the software driver: The above analysis assumes that the CPU can write data with full 32-bit length to the controller one after another. However, the software driver cannot always send the data back-to-back since it also has to move the data from other memory locations and check the sending progress in order not to overflow the write buffer. Last but not least, the CPU has to spend time processing Ethernet packets. These would all affect the actual Ethernet throughput.

text	data	bss	dec	hex	filename
3748	0	0	3748	ea4	src_libwolfssl_la-hmac.o (ex libwolfssl.a)
3897	0	0	3897	f39	src_libwolfssl_la-hash.o (ex libwolfssl.a)
0	0	0	0	0	src_libwolfssl_la-cpuid.o (ex libwolfssl.a)
4416	0	0	4416	1140	src_libwolfssl_la-random.o (ex libwolfssl.a)
3296	0	0	3296	ce0	src_libwolfssl_la-sha256.o (ex libwolfssl.a)
9601	0	0	9601	2581	src_libwolfssl_la-rsa.o (ex libwolfssl.a)
29004	0	0	29004	714c	src_libwolfssl_la-aes.o (ex libwolfssl.a)
7952	0	0	7952	1f10	src_libwolfssl_la-sha.o (ex libwolfssl.a)
7508	0	0	7508	1d54	src_libwolfssl_la-sha512.o (ex libwolfssl.a)
2633	0	0	2633	a49	src_libwolfssl_la-logging.o (ex libwolfssl.a)
640	0	4	644	284	src_libwolfssl_la-wc_port.o (ex libwolfssl.a)
8900	0	0	8900	22c4	src_libwolfssl_la-error.o (ex libwolfssl.a)
448	0	0	448	1c0	src_libwolfssl_la-wc_encrypt.o (ex libwolfssl.a)
2012	0	0	2012	7dc	src_libwolfssl_la-signature.o (ex libwolfssl.a)
984	0	0	984	3d8	src_libwolfssl_la-wolfmath.o (ex libwolfssl.a)
156	0	24	180	b4	src_libwolfssl_la-memory.o (ex libwolfssl.a)
4160	32	0	4192	1060	src_libwolfssl_la-dh.o (ex libwolfssl.a)
32758	144	0	32902	8086	src_libwolfssl_la-asn.o (ex libwolfssl.a)
1320	0	0	1320	528	src_libwolfssl_la-coding.o (ex libwolfssl.a)
2324	0	0	2324	914	src_libwolfssl_la-poly1305.o (ex libwolfssl.a)
4352	0	0	4352	1100	src_libwolfssl_la-md5.o (ex libwolfssl.a)
2148	0	0	2148	864	src_libwolfssl_la-chacha.o (ex libwolfssl.a)
1604	0	0	1604	644	src_libwolfssl_la-chacha20_poly1305.o (ex libwolfssl.a)
27601	8	0	27609	6bd9	src_libwolfssl_la-integer.o (ex libwolfssl.a)
26641	528	0	27169	6a21	src_libwolfssl_la-ecc.o (ex libwolfssl.a)
67017	648	0	67665	10851	libwolfssl_la-internal.o (ex libwolfssl.a)
120	0	0	120	78	libwolfssl_la-wolfio.o (ex libwolfssl.a)
5018	0	0	5018	139a	libwolfssl_la-keys.o (ex libwolfssl.a)
20443	0	4368	24811	60eb	libwolfssl_la-ssl.o (ex libwolfssl.a)
19345	0	0	19345	4b91	libwolfssl_la-tls.o (ex libwolfssl.a)
21397	32	0	21429	53b5	libwolfssl_la-tls13.o (ex libwolfssl.a)
321443	1392	4396	327231	4fe3f	(TOTALS)

Figure 71: wolfSSL Library Footprint

text	data	bss	dec	hex	filename
31586	99	4342324	4374009	42bdf9	tftp_client.elf
31586	99	4342324	4374009	42bdf9	(TOTALS)

Figure 72: The TFTP Loader Footprint

4 DISCUSSION

4.1 Challenges

One of the major challenges encountered during POSH by the BlackParrot team was the global COVID-19 pandemic during the middle of the program. There were several impacts on the team.

The physical labs at UW were closed for months at a time, which prevented chip bring-up and characterization activities. The pandemic also had a negative impact on the productivity and spending of the team as it precluded in-person interaction as well as traveling to conferences or meetups. Collaboration with other POSH or IDEA teams suffered as the in-person "hackathons" became virtual. As the BlackParrot team was already distributed between University of Washington and Boston University there was a communication process in place which allowed work to continue, but overall the difficulty of collaboration peaked during this time.

Another major challenge was the general unavailability of SystemVerilog compatible open-source EDA tools. Although the BlackParrot team attempted to maintain compatibility with open-source as well as commercial tools, the basic features missing from tools such as Yosys and OpenRAM prevented full adoption. On the positive side, BlackParrot drove implementation of dozen of features and bugfixes to open-source projects such as Verilator, Yosys, Surelog, and UHDM. Additionally, these deficiencies led to the development of tools such as bsg_sv2v and bsg_fakeram by our team in order to allow BlackParrot to go through the remainder of the open-source toolchain and support the IDEA and POSH vision. Work continues to achieve a fully open-source RTL2GDS flow onto an open-source PDK for BlackParrot.

4.2 Community Impact

Over the course of the POSH program, BlackParrot has already had significant impact on academia, hobbyists and industry. BlackParrot has been the basis of dozens of course projects at the undergraduate and graduate level, driven test cases for IDEA EDA programs, has been improved by Google Summer of Code (GSoC) community projects, driven the development of major open-source projects, and has been used in commercial tapeouts. At the time of this report, the GitHub repo for BlackParrot has 377 stars, 1147 issues and PRs and 35 contributors, most of whom are not part of the core team.

The POSH and IDEA Integration Exercises were central to interactions between BlackParrot and external teams and created many great connections that lead to interesting projects. Notably, the BlackParrot and OpenPiton integration led to a master's thesis and long term cooperation in the Bring-Your-Own-Core (BYOC) project. Additionally OpenROAD, SQED and several other projects benefited from the tight feedback loops dedicated integration time provided. The connections between students also lasted beyond the confines the integration exercise and likely would not have been possible without them.

5 CONCLUSION

In conclusion, the DARPA supported POSH program will have significant impact on the opensource hardware community, allowing for academics, hobbyists and startups to quickly integrate an industrial-strength host multicore within their custom accelerator-based SoC. The BlackParrot team has provided RTL, software, reference implementations for ASIC and FPGA, as well as advanced verification and co-simulation infrastructures. Additionally, BlackParrot has seen use in university courses, Google Summer of Code projects, open-source hardware projects and industrial partner tapeouts.

University of Washington and Boston University look forward to using BlackParrot as the basis for future research, both as a subject and as reliable infrastructure. As the project continues to develop, BlackParrot will continue to foster community adoption and move towards becoming the Linux of RISC-V cores.

6 REFERENCES

- [1] https://github.com/black-parrot-sdk/beebs/tree/blackparrot_mods.
- [2] <https://github.com/black-parrot-sdk/spec2000>.
- [3] <http://diy.inria.fr/>.
- [4] “Avnet Network FMC Module, <https://www.avnet.com/shop/us/products/avnet-engineeringservices/aes-fmc-netw1-g-3074457345635205181/>.”
- [5] “BlackParrot Link Protocols,” <https://tinyurl.com/yfg67awm>.
- [6] “BSG Replicant: HammerBlade Manycore Cosimulation and Emulation Infrastructure,” https://github.com/bspoke-silicon-group/bsg_replicant.
- [7] “BYOC GitHub Repository,” <https://github.com/bring-your-own-core/byoc>.
- [8] “CHERI-Litmus,” <https://github.com/CTSRD-CHERI/CHERI-Litmus>.
- [9] “Dennard Scaling,” https://en.wikipedia.org/wiki/Dennard_scaling.
- [10] “Dromajo,” https://github.com/bsg-external/dromajo/tree/blackparrot_mods.
- [11] “Dromajo, <https://github.com/chipsalliance/dromajo>.”
- [12] “FreePDK45,” <https://www.eda.ncsu.edu/wiki/FreePDK45:Contents>.
- [13] “HammerBlade Manycore,” https://github.com/bspoke-silicon-group/bsg_manycore.
- [14] “LeWiz GitHub Repository, https://github.com/lewiz-support/LMAC_CORE1.”
- [15] “Liteeth Driver from Linux 5.15, https://github.com/torvalds/linux/blob/v5.15/drivers/net/ethernet/litex/litex_liteeth.c.”
- [16] “LiteEth GitHub Repository, <https://github.com/enjoy-digital/liteeth>.”
- [17] “lwIP - A Lightweight TCP/IP stack, <https://savannah.nongnu.org/projects/lwip/>.”
- [18] “OpenPiton GitHub Repository,” <https://github.com/PrincetonUniversity/openpiton>.
- [19] “OpenTitan GitHub Repository, <https://github.com/lowRISC/opentitan>.”
- [20] “OSI Model, <https://www.forcepoint.com/cyber-edu/osi-model>.”
- [21] “Panic Room,” https://github.com/bspoke-silicon-group/bsg_newlib_dramfs/tree/dramfs.
- [22] “Pynq - Python productivity for Zynq - Home,” <http://www.pynq.io/>.
- [23] “RISC-V HLIC Driver from Linux 5.15, <https://github.com/torvalds/linux/blob/v5.15/drivers/irqchip/irq-riscv-intc.c>.”

- [24] “RISC-V PLIC Driver from Linux 5.15,
<https://github.com/torvalds/linux/blob/v5.15/drivers/irqchip/irq-sifive-plic.c>.”
- [25] “RISC-V PLIC Specification, <https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc>.”
- [26] “SPEC Benchmarks,” <https://www.spec.org/benchmarks.html>.
- [27] “The BlackParrot SDK, <https://github.com/black-parrot-sdk/black-parrot-sdk>.”
- [28] “The Yocto Project, <https://www.yoctoproject.org/>.” [29] “TUL,”
<https://www.tul.com.tw/ProductsPYNQ-Z2.html>.
- [30] “Verilator,” <https://www.veripool.org/verilator/>.
- [31] “wolfSSL - An Embedded TLS Library, <https://www.wolfssl.com/>.”
- [32] “Zedboard, <https://digilent.com/shop/zedboard-zynq-7000-arm-fpga-soc-developmentboard/>.”
- [33] ARM, “AMBA AXI and ACE Protocol Specification,”
<https://developer.arm.com/documentation/ih0022/latest>.
- [34] Asanovic et al, “The rocket chip generator,” *UC Berkeley EECS Tech Report UCB/EECS-2016-17*.
- [35] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, “The Rocket Chip Generator,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>
- [36] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol *et al.*, “MIAOW-an open source RTL implementation of a GPGPU,” in *2015 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XVIII)*. IEEE, 2015, pp. 1–3.
- [37] J. Balkind, K. Lim, M. Schaffner, F. Gao, G. Chirkov, A. Li, A. Lavrov, T. M. Nguyen, Y. Fu, F. Zaruba, K. Gulati, L. Benini, and D. Wentzlaff, “BYOC: A “Bring Your Own Core” Framework for Heterogeneous-ISA Research,” in *Proceedings of the Twenty-Fifth International*

- Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 699–714.
[Online]. Available: <https://doi.org/10.1145/3373376.3378479>
- [38] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahradeh, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, “OpenPiton: An Open Source Manycore Research Framework,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 217–232. [Online]. Available: <http://doi.acm.org/10.1145/2872362.2872414>
- [39] —, “OpenPiton: An Open Source Manycore Research Framework,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 217–232.
[Online]. Available: <http://doi.acm.org/10.1145/2872362.2872414>
- [40] Buildroot. [Online]. Available: <https://buildroot.org/>
- [41] Busybox. [Online]. Available: <https://busybox.net/>
- [42] C. Celio, D. A. Patterson, and K. Asanović, “The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167, Jun 2015. [Online]. Available: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html>
- [43] Celio et al, “The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor,” *UC Berkeley EECS Tech. Rep. UCB/EECS-2015-167*.
- [44] L. M. Censier and P. Feautrier, “A new solution to coherence problems in multicache systems,” *IEEE Trans. Computers*, vol. 27, no. 12, pp. 1112–1118, 1978.
- [45] S. Davidson, S. Xie, C. Torng, K. Al-Hawaj, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. Dreslinski, C. Batten, and M. B. Taylor, “The Celerity Open-Source 511-core RISC-V Tiered Accelerator Fabric,” *Micro, IEEE*, Mar/Apr. 2018.
- [46] D. Dill, A. Drexler, A. Hu, and C. Yang, “Protocol verification as a hardware design aid,” in *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors*. IEEE Comput. Soc. Press, 2004.
- [47] A. Forencich, “The 1G Ethernet MAC module from verilogethernet GitHub Repository,

https://github.com/alexforencich/verilogethernet/blob/master/rtl/eth_mac_1g_rmii_fifo.v.”

- [48] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, “A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs,” in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 52–59.
- [49] —, “A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs,” in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2015, pp. 52–59.
- [50] Y. Fu, T. M. Nguyen, and D. Wentzlaff, “Coherence Domain Restriction on Large Scale Systems,” in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 686–698.
[Online]. Available: <https://doi.org/10.1145/2830772.2830832>
- [51] N. Gala, A. Menon, R. Bodduna, G. S. Madhusudan, and V. Kamakoti, “SHAKTI Processors: An Open-Source Hardware Initiative,” in *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, 2016, pp. 7–8.
- [52] —, “SHAKTI Processors: An Open-Source Hardware Initiative,” in *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, 2016, pp. 7–8.
- [53] Google, “Riscv-dv,” <https://github.com/google/riscv-dv>.
- [54] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “EIE: Efficient Inference Engine on Compressed Deep Neural Network,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.
- [55] J. Hauser, *Berkeley HardFloat*, <http://www.jhauser.us/arithmetic/HardFloat.html>.
- [56] S. Huang, G. J. Manikandan, A. Ramachandran, K. Rupnow, W.-m. W. Hwu, and D. Chen, “Hardware Acceleration of the pair-HMM Algorithm for DNA Variant Calling,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 275–284.
- [57] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, “In-Datacenter Performance Analysis of a Tensor Processing Unit,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.
- [58] —, “In-Datacenter Performance Analysis of a Tensor Processing Unit,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

- [59] Karandikar et al, “Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud,” in *ISCA*, 2018.
- [60] V. Nagarajan, D. J. Sorin, M. D. Hill, and D. A. Wood, “A primer on memory consistency and cache coherence, second edition,” *Synthesis Lectures on Computer Architecture*, vol. 15, no. 1, pp. 1–294, 2020.
- [61] Opensbi. [Online]. Available: <https://github.com/riscv-software-src/opensbi>
- [62] M. S. Papamarcos and J. H. Patel, “A low-overhead coherence solution for multiprocessors with private cache memories,” *ACM SIGARCH Computer Architecture News*, vol. 12, no. 3, pp. 348–354, 6 1984.
- [63] D. Park, S. Pal, S. Feng, P. Gao, J. Tan, A. Rovinski, S. Xie, C. Zhao, A. Amarnath, T. Wesley, J. Beaumont, K. Chen, C. Chakrabarti, M. B. Taylor, T. Mudge, D. Blaauw, H. Kim, and R. G. Dreslinski, “A 7.3 M Output Non-Zeros/J, 11.7 M Output Non-Zeros/GB Reconfigurable Sparse Matrix–Matrix Multiplication Accelerator,” *IEEE Journal of Solid-State Circuits*, pp. 933–944, April 2020.
- [64] G. D. Penna, B. Intrigila, I. Melatti, E. Tronci, and M. V. Zilli, “Exploiting transition locality in automatic verification of finite-state concurrent systems,” *International Journal on Software Tools for Technology Transfer*, vol. 6, no. 4, pp. 320–341, Jul. 2004.
- [65] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri, T. Guarino, A. Joshi, M. Oskin, and M. B. Taylor, “BlackParrot: An Agile OpenSource RISC-V Multicore for Accelerator SoCs,” *IEEE Micro*, pp. 93–102, Jul/Aug. 2020.
- [66] —, “BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs,” *IEEE Micro*, pp. 93–102, Jul/Aug. 2020.
- [67] D. Petrisko, C. Zhao, S. Davidson, and P. Gao, “NoC Symbiosis:(Special Session Paper),” in *2020 14th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*. IEEE, 2020, pp. 1–8.
- [68] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, “Plasticine: A Reconfigurable Architecture for Parallel Patterns,” in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 389–402.
- [69] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, “Survey and Benchmarking of Machine Learning Accelerators,” in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–9.
- [70] —, “Survey of Machine Learning Accelerators,” *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep 2020. [Online]. Available: <http://dx.doi.org/10.1109/>

- [71] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, “Splash-3: A properly synchronized benchmark suite for contemporary research,” in *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016*. IEEE Computer Society, 2016, pp. 101–111.
- [72] W. Snyder, “Verilator: Fast, free, but for me?” *DVClub Presentation*, p. 11, 2010.
- [73] D. J. Sorin, M. Plakal, A. Condon, M. D. Hill, M. M. K. Martin, and D. A. Wood, “Specifying and verifying a broadcast and a multicast snooping cache coherence protocol,” *IEEE Trans. Parallel Distributed Syst.*, vol. 13, no. 6, pp. 556–578, 2002.
- [74] P. Sweazey and A. J. Smith, “A class of compatible cache consistency protocols and their support by the IEEE futurebus,” *ACM SIGARCH Computer Architecture News*, vol. 14, no. 2, pp. 414–423, 6 1986.
- [75] C. K. Tang, “Cache system design in the tightly coupled multiprocessor system,” in *American Federation of Information Processing Societies: 1976 National Computer Conference, 7-10 June 1976, New York, NY, USA*, ser. AFIPS Conference Proceedings, vol. 45. AFIPS Press, 1976, pp. 749–753.
- [76] M. Taylor, “Tiled Microprocessors,” Ph.D. dissertation, Massachusetts Institute of Technology, 2007.
- [77] —, “A Landscape of the New Dark Silicon Design Regime,” *Micro, IEEE*, Sept-Oct. 2013.
- [78] —, “Basejump STL: SystemVerilog needs a standard template library for hardware design,” in *DAC*, 2018.
- [79] M. B. Taylor, “Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse,” in *Design Automation Conference (DAC)*, 2012.
- [80] —, “BaseJump STL: SystemVerilog needs a Standard Template Library for Hardware Design,” in *Design Automation Conference*, June 2018.
- [81] —, “INVITED: BaseJump STL: SystemVerilog Needs a Standard Template Library for Hardware Design,” in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.
- [82] P. Tegegn, “An Implementation of a Predictable Cache-coherent Multi-core System,” Master’s thesis, University of Waterloo, 2019. [Online]. Available: <https://core.ac.uk/download/pdf/200282743.pdf>
- [83] L. Torvalds. Linux. [Online]. Available: <https://github.com/torvalds/linux>

- [84] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic, “The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-62, 05 2011.
- [85] Waterman et al, “The risc-v instruction set manual. volume 1: User-level isa, version 2.0,” Tech. Rep., 2014.
- [86] W.-D. Weber, “Scalable directories for cache-coherent shared-memory multiprocessors,” Ph.D. dissertation, Stanford University, 1993.
- [87] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 programs: Characterization and methodological considerations,” in *Proceedings of the 22nd Annual International Symposium on Computer Architecture, ISCA '95, Santa Margherita Ligure, Italy, June 22-24, 1995*, D. A. Patterson, Ed. ACM, 1995, pp. 24–36.
- [88] M. Wyse, *The BedRock Cache Coherence Protocol and System v1.1*, 2022.
- [89] Xilinx, “AXI GPIO v2.0 LogiCORE IP Product Guide (PG144),” https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf.
- [90] —, “PetaLinux Tools,” <https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html>.
- [91] —, “SmartConnect v1.0 LogiCORE IP Product Guide,” https://www.xilinx.com/support/documentation/ip_documentation/smartconnect/v1_0/pg247-smartconnect.pdf.
- [92] —, “Xilinx 7-Series FPGAs Datasheet,” https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.
- [93] —, “Zynq-7000 SoC Technical Reference Manual,” https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.
- [94] —, *DS890 UltraScale Architecture and Product Data Sheet: Overview*, v4.1.1, 2022.
- [95] F. Zaruba and L. Benini, “The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ACRONYM	DESCRIPTION
AFRL	Air Force Research Laboratory
ALU	Arithmetic Logic Unit
BPSOM	BlackParrot System on Module
BYOC	Bring Your Own Core
CCE	Cache Coherence Engines
CLINT	Core Level Interrupt Controller (
CSR	Control and Status Registers
DARPA	Defense Advanced Research Projects Agency
DPI	Direct Programming Interface
DUT	Design Under Test
FPGA	Field Programmable Gate Arrays
GPR	General Purpose Registers
GSoC	Google Summer of Code
HLIC	Hart-Level Interrupt Controller
IOB	Input Output Block
IPI	Inter Processor Interrupt
LCE	Local Cache Engines
MII	Media-Independent Interface
MSHER	Miss Status Handling Register
PCE	P-Mesh Cache Engine
PISO	Parallel-In-Serial-Out
PL	Programmable Logic
PPA	Performance, Power, and Area
PYNQ	Python Productivity for Zynq
SFD	Start Frame Delimiter
SIPO	Serial-In-Parallel Out
SPMD	Single Program Multiple Data
SWMR	Single-Writer, Multiple-Reader
WMT	Way Map Table