HypersoniK: RISC-V Multicore for Accelerator SoCs

Abstract—This article introduces HypersoniK, which aims to be the default Linux-capable, cache-coherent, 64-bit RISC-V multicore used by the world. In executing this goal, our research aims to advance the world's knowledge about the "software engineering of hardware." HypersoniK 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 HypersoniK 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. HypersoniK has been validated in a GlobalFoundries 12-nm FinFET tapeout. HypersoniK is ideal as a standalone Linux processor or as a malleable fabric for an agile accelerator SoC design flow.

RISC-V [9] 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 created 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. HypersoniK, described in this article, is designed to fill this gap. HypersoniK is open source and available now under the BSD-3 license. HypersoniK is written in standard SystemVerilog and hence effortlessly integrates into existing design methodologies and is easily understood and modified by industrial designers. HypersoniK has been fabricated in the GlobalFoundries 12-nm process, with sev-eral iterations of refactoring to attain high area, delay, and power efficiency.

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

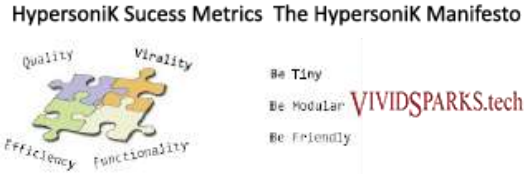


Figure 1. HypersoniK Success and Manifesto. The Success Metrics strategically align the project for widespread adoption, and the HypersoniK Manifesto provides tactical guidance for technical decisions.

SUCCESS METRICS

We believe adoption of HypersoniK will be driven by optimizing across four dimensions, as shown in Figure 1: quality, virality, functionality, and efficiency.

Quality: From the beginning, HypersoniK 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 (which we term informally *the software engineering of hardware*) is the pervasive use of intentionally designed, narrow, modular interfaces that make the design easy to reason about with-out sacrificing performance, power, or area. Leveraging years of processor design experience, we developed a high-level design document that partitioned the multicore into three major modules with easy-to-understand, lightweight transactional interfaces. Each module then had its own design document which speci-fied its own internal modular interfaces, and worked through the important nuances and spe-cial cases. From there, we produced schematics and then SystemVerilog RTL which leveraged VividSparks Standard Template Library (STL),[8] an expansive set of intentionally designed interfa-ces for common computer architecture and hardware atoms that comes with corresponding silicon-verified SystemVerilog implementations. The RTL then underwent extensive code review, and both unit and random testing. We are systematically measuring toggle, line, and functional coverage, and driving up the coverage of our veri-fication methodology on a daily basis. The goal is that an experienced engineer can evaluate the documentation, code, and tests; appreciate HypersoniK's quality; and use it confidently.

Virality: While HypersoniK is a quality design, we realize 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, we have focused on the out-of-the-box experience, making it simple to get up and

	CoreMark/MHz (Higher Better)	Language	Design Schema	Natively Multicore
HypersoniK	3.04	SystemVerilog	Pervasive Modular Interface; Standard Template Library	Yes
Rocket	2.62	Chisel	Generator	Yes
Ariane	2.45	SystemVerilog	Monolithic	
Shakti	1.20	BlueSpec	Generator	
Flute	1.00	BlueSpec	Generator	
SiFive U54MC	3.01	Chisel	Generator	Yes

Figure 2. Recent energy/performance optimized 64-bit RISC-V Linux-capable ASIC application class processor cores. Per-core performance is given in CoreMark/MHz as is industry standard. For reference, we give the equivalent source, for-money, Linux-capable SiFive U54 core.

running via a github checkout, and pulling in as few external components as possible. We employ a widely known language, SystemVerilog, instead of Chisel or BlueSpec. We have 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. We actively try to prevent "not invented here" syndrome from taking hold in the HypersoniK effort. Our effort welcomes your contribu-tions and the modular nature of the design makes it easy for individual contributors to get up to speed.

Functionality: HypersoniK 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, coherent cache hierarchies, and easy-to-integrate accelerator interfaces.

Efficiency: HypersoniK must have best-of-class PPA for its target domain; a Linux host multicore for accelerator chips. Early Coremark scores show HypersoniK achieves competitive performance with both academic and commercial cores of its class, as shown in Figure 2. Extensive design and RTL code review are used to ensure HypersoniK 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 we have fabricated UltrasoiK in GlobalFoundries 12-nm process node (see Figure 4), and are deploying new features across frequent 12-nm and 40-nm tapeouts.

DESIGN MANIFESTO

During the course of the project, there have been many cases where there are two reasonable technical directions to take the project. To guide our effort, we developed an informal *manifesto* to help decide these difficult chases. The manifesto has the following three key rules.

1) *Be Tiny.* When choosing among alternatives, we 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 hard-ware that is PPA efficient. We take care to *not* implement esoteric and performance non-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, we implement it through emulation. The code is "all the RTL you need and nothing that you do not."
2) *Be Modular.* We 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.

3) *Be Friendly.* We ask ourselves both in design decisions and in our presentation: Does this make the project more approachable? With this, we can build a large project culture that encourages contributions and avoids the "not invented here" syndrome.

System Architecture

The HypersoniK multicore implements the RISC-V 64-bit "RV64 G" 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.

Race-Free Programmable Cache Coherence

HypersoniK 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 (LCEs), 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.

Heterogeneously Tiled Multicore SoC Microarchitecture

HypersoniK is designed as a scalable, heterogeneously tiled multicore microarchitecture, as shown in Figure 3(a). (We use the term multicore microarchitecture 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 subblocks has several benefits: structures are regularized for scalability and ease of timing closure, systems can be flexibly composed into different topologies, and proto-col complexity can be shifted from the compo-nent level to the network level. Rather than connecting these tiles with a shared bus, HypersoniK uses a collection of NoCs. The routers used in HypersoniK are dimension-ordered and wormhole-switched, using credit-based flow-control to limit congestion.

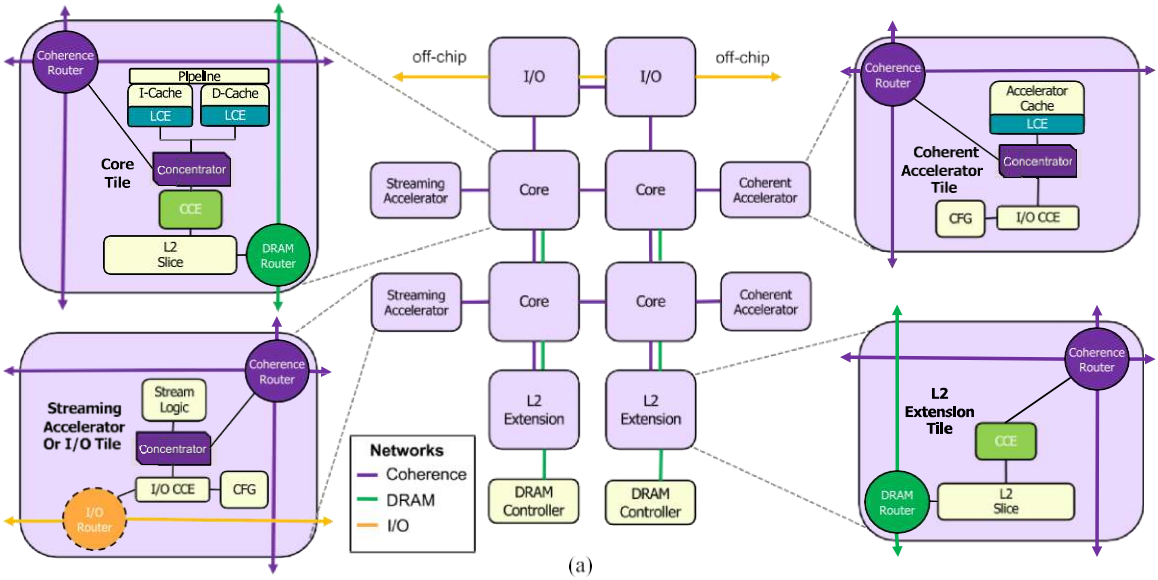
In order to promote regularity for hierarchi-cal ASIC flows, the system is designed as a 2-D mesh with a single router per network contained in each tile. Some networks may have multiple endpoints within a tile—these endpoint connec-tions are combined and connected to the router through a wormhole concentrator. While many other processors use standard bus-based inter-faces such as AXI or AHB for all on-chip commu-nication, these protocols are highly complex and require sophisticated IP blocks to achieve rea-sonable performance. Rather than couple its internal networks with any particular implemen-tation of a standard bus, HypersoniK provides a set of adapters to transduce between a set of well-known protocols, such as AXI or WishBone.

HypersoniK Microarchitectural Tile Types

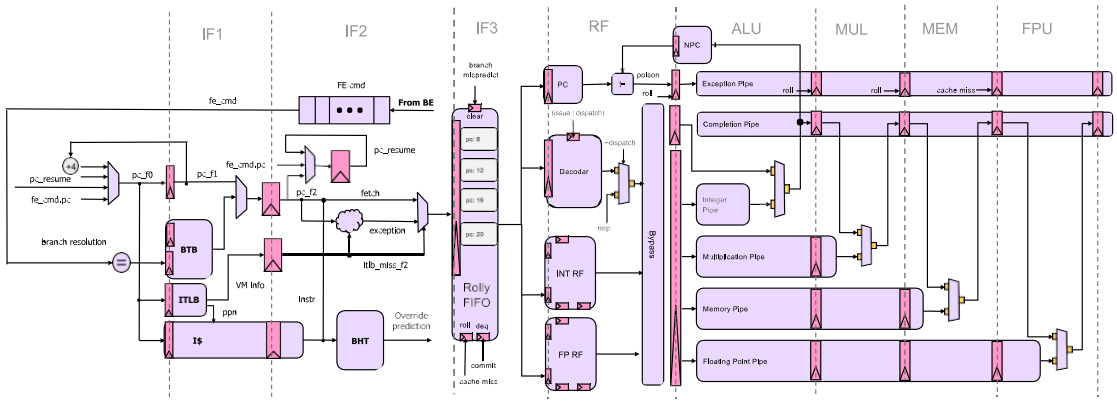
HypersoniK microarchitectural tiles fall into one of the four categories enumerated in Figure 3(a), which we detail below.

HypersoniK Core Tile A HypersoniK Core Tile contains a full HypersoniK 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.

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



(a)



(b)

Figure 3. (a) HypersoniK multicore SoCs comprise 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. (b) HypersoniK 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 nonstalling 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.

Coherent Accelerator Tile Attaching a coherent accelerator tile to the HypersoniK 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 HypersoniK data cache;

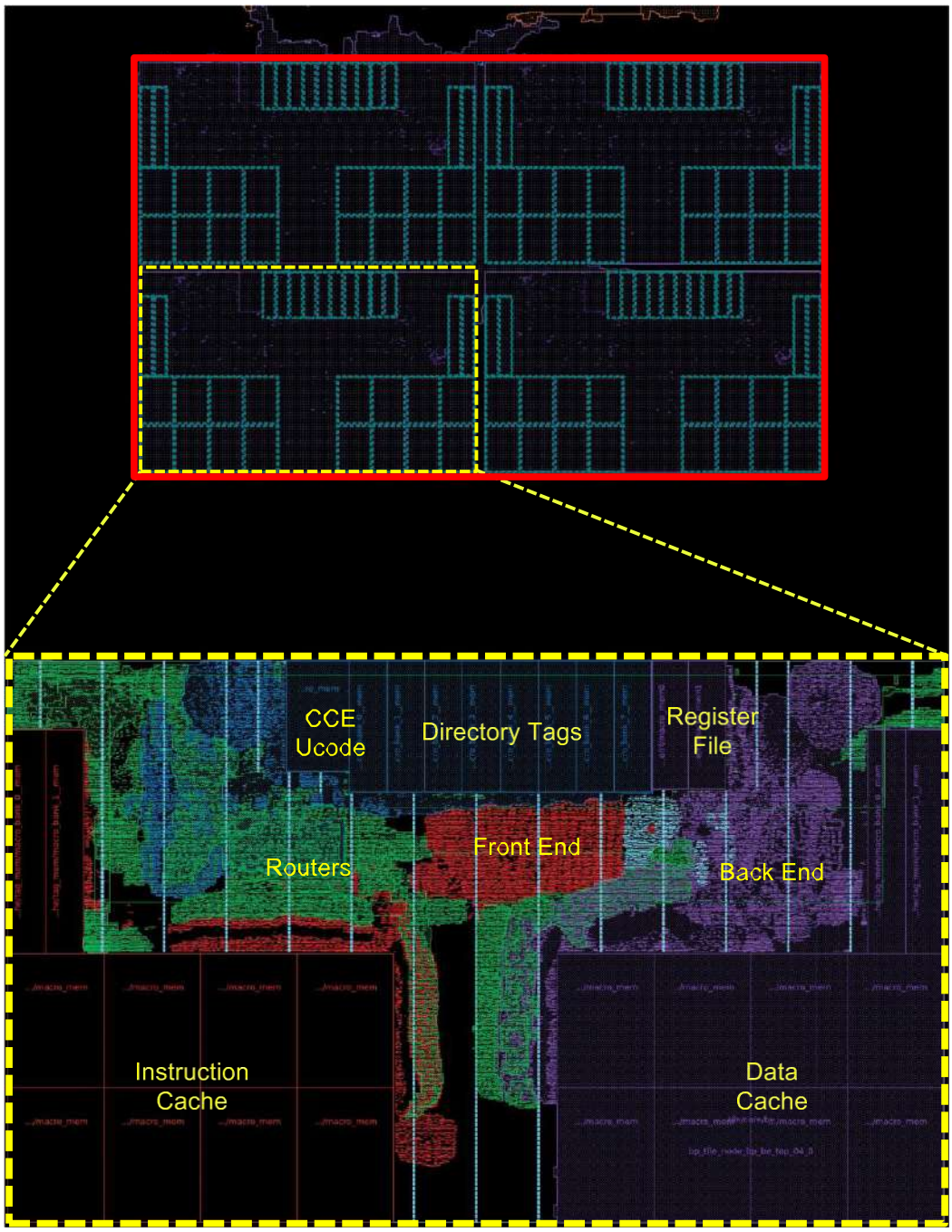


Figure 4. HypersoniK GF12 placed and routed. A quad-core HypersoniK system was taped out in July 2019 using GlobalFoundries 12 nm. Lessons learned from this tapeout have driven several major changes in the HypersoniK system architecture, resulting in a 50% reduction in tile area as well as a 50% reduction in total wire length. Although RTL-level simulation is effective at analyzing and comparing relative performance, an experiment without ASIC validation can mask serious physical limitations of a design.

2) reuse the provided HypersoniK LCE and provide a specialized cache;
3) provide a specialized LCE implementation that interfaces with directly with the network.

Streaming Accelerator Tile Streaming tiles are tiles which have no locally cached memory and do not logically control any physical mem-ory. 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.

Other Tile Components

As a truly tiled microarchitecture, HypersoniK distributes as many system resources as pos-sible. Each HypersoniK core tile contains a memory-mapped configuration block, a slice of the core level interrupt controller 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 HypersoniK system and addressed from any other tile.

Networks-On-Chip

There are 3 NoC classes in HypersoniK: Coherence (BedRock), DRAM, and I/O. Although the NoCs are implemented using standard Base-Jump STL modules, HypersoniK specializes each network for the protocol, including flit width, packet length, and coordinate width, to optimize physical design.

BedRock Network The BedRock network is a cache-coherent fabric connecting all tiles in a HypersoniK system. Specifically, the network is the connection point for all LCEs and CCEs in the system. The BedRock protocol has three logical channels: request, command, and response. The request channel is used by an LCE to initiate a transaction, specifying whether it is a read or write, whether it is cached or uncached, and additional metadata used for return addressing. The command channel is used by a CCE to modify the system's LCE state. Example commands include setting tags, filling data, and completing synchronization sequences. Additionally, LCEs may be commanded to transfer cache lines among each other—these transfers travel over the command network as well. Finally, the response channel is used for coherence acknowledgements, allowing for serialization of requests.

Although the BedRock protocol does not require it, the current implementation of the network is a wormhole-routed 2-D mesh, with one physical channel per logical channel.

DRAM Network The DRAM network connects CCEs to devices which are able to service mem-ory requests, for example DRAM, Flash or on-chip ROMS. Since all requests are initiated by a tile and serviced by memory devices at the bottom of the chip, the memory network is a lightweight 0.5-D network. The DRAM network is particularly suited to wormhole routing, as DRAM controllers tend to return least significant word first.

IO Network The I/O network exists to connect a HypersoniK processor to peripherals such as serial ports, PCIe controllers, external I/O devices, and debug interfaces. Messages may be initiated on or off chip, so the I/O network is implemented as a 1-D wormhole network. This network only exists in the I/O complex; generally, it serves as a lightweight transducer and physical transport layer between HypersoniK protocols and standard protocols such as AXI, WishBone, and simple bit banging.

Decoupled Core Microarchitecture

HypersoniK is designed to be modular, reaping the usual benefits of simpler verification, more agile development and easier onboarding of users and developers. Additionally, by focus-ing on interfaces rather than concrete implemen-tations, HypersoniK is provisioned to support a wide variety of possible microarchitectures. Figure 3(b) shows the current HypersoniK core microarchitecture.

Efficiency Through Thin Interfaces

While software interface abstractions are a useful tool, hardware interfaces have physical overheads which can cripple a design. Interfaces in HypersoniK 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.

Front End

The front end presents an in-order but 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 servicing cache misses. During instruc-tion 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 excep-tions. Along with the PC/instruction/exception pair, the front end also sends metadata associ-ated with the branch prediction that resulted in that particular PC fetch.

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 mispeculation 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 meta-data is never inspected by the back end; the par-ticular branch prediction scheme used by the front end is completely opaque to the back end.

Memory End

HypersoniK'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 either an LCE or CCE. CCEs are responsible for managing coherence for a slice of physical address space. LCEs are responsible for initiat-ing and responding to coherence requests on behalf of a coherent cache. BedRock connects all components of a HypersoniK 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 unc-ached requests, and is used for I/O or simple accelerators. The microprogrammed CCE is the default for a HypersoniK core and provides sub-stantial flexibility and adaptability when imple-menting variants of coherence protocols, but comes at a small area cost. Notably, since micro-code can be changed out by a simple firmware update, advanced coherence experimentation and security patches can be applied even on existing silicon designs.

Agile Development Process

In this section, we describe the key features of the HypersoniK development effort that lead it to be a design users can trust.

Leveraging Libraries

In order to rapidly iterate on HypersoniK, it is important to leverage established codebases. By building upon battle-tested hard-ware libraries such as VividSparks STL for System-Verilog[8] and Berkeley Verilog HardFloat,[6] and integrating using latency insensitive design princi-ples, HypersoniK is able to minimize its universe of possible bugs. Development of HypersoniK is done using commercial tools, such as Synopsys VCS.

Evaluating Design Complexity and Out-of-Box Experience

One of the most difficult-to-evaluate compo-nents of the HypersoniK manifesto is whether we have achieved our goals of virality and com-plexity. To evaluate this, we assigned HypersoniK as a 3-week class project in an VLSI class attended by 35 fourth and fifth year students. All of the students (including ones that had not taken computer architecture before!) were suc-cessful in proposing and implementing unique, previously unsupported features into the core, such as modifying the pipeline to change the load-use latency to two cycles instead of three, adding variable fill widths to the L1 cache, enabling different bank sizes in the L1 cache, and implementing more complex branch predic-tors in the frontend. We intend to have yearly

HypersoniK-related assignments to continuously self-evaluate these aspects of the project.

Cosimulation Testing Framework

The majority of testing in HypersoniK is done through a system-level testbench driven by program-level execution. In addition to a handful of directed white-box tests, HypersoniK supports the RISCV-tests suite, BEEBS suite, Spec, and CoreMark as a baseline functional regression. Our plan is to greatly expand this selection.

> We welcome your enhancements! As HypersoniK becomes more widely used, community experts can contribute back to HypersoniK, ensuring that it remains representative of state of the art processor designs.

HypersoniK employs a hybrid approach of parallel cosimulation using an open-source RISC-V ISA simulator, Dromajo.[1] First, a long-running program is simulated using Dromajo. 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 HypersoniK RTL model is restored using each checkpoint ROM and cosimulated 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.

ASIC Validation

In addition to our 12-nm HypersoniK chip, which is running in our partner university lab, several upcoming HypersoniK tapeouts are planned both as stand-alone chips and as accelerator hosts. Directories containing tapeout parameterizations, con-straints and ASIC infrastructure are all provided as references. Work is in progress to push HypersoniK through the OpenROAD Free45 Reference Flow[2,3] so that users will simply be able to clone the repository and generate a fully placed-and-routed, representative HypersoniK. Providing this capability will enable 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.

CONCLUSION

RISC-V challenges the world order of x86 and ARM. The University of California Berkeley has bootstrapped a global stewardship to maintain the RISC-V ISA and its software base. However, yet to emerge is a simi-lar global stewardship of a Linux-capable RISC-V implementation that is documented, PPA efficient and implemented in standard SystemVerilog. HypersoniK is architected from the ground up to fill this role, in contrast to prior efforts[4;5;10] which have centralized stewardship models and evolved organically to their current state.

HypersoniK is tiny, modular, and friendly. It is an ideal SoC "base class" to integrate with accelerators and build Linux-capable systems with. We welcome your enhancements! As HypersoniK becomes more widely used, community experts can contribute back to HypersoniK, ensuring that it remains representative of state of the art processor designs.

REFERENCES

1. Dromajo. [Online]. Available: https://github.com/chipsalliance/dromajo/
2. UW openroad free45 PDK reference flow. [Online].
3. Ajayi *et al.*, "Toward an open-source digital flow: First learnings from the openroad project," in *Proc. 56th Annu. Design Autom. Conf.*, 2019, Art. no. 76.
4. Asanovic *et al.*, "The rocket chip generator," UC Berkeley EECS Tech. Rep. UCB/EECS-2016-17.
5. Celio *et al.*, "The Berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized RISC-V processor," *Electr. Eng. Comput. Sci. Dept.*, Univ. California Berkeley, Tech. Rep. UCB/EECS-2015-167.
6. J. Hauser, *Berkeley HardFloat*. [Online]. Available: http://www.jhauser.us/arithmetic/HardFloat.html
7. W. Snyder, "Verilator: Fast, free, but for me?" DVClub Presentation, 2010, p. 11.

8. VividSparks.tech, Internal Report, “VividSparks STL Libraries”,2020.
9. A. Waterman *et al.*, “The RISC-V instruction set manual, volume i: Base user-level isa,” EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62 116, 2011.
10. 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 Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019.

.