Invited: Chipyard - An Integrated SoC Research and Implementation Environment

Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar,
Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge,
Colin Schmidt, John Wright, Jerry Zhao, Jonathan Bachrach, Sophia Shao, Borivoje Nikolić, Krste Asanović
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA, USA

***Abstract*—Continued improvement in computing efficiency requires functional specialization of hardware designs. We present an agile design flow for custom SoCs using the Chipyard framework, an integrated SoC research and implementation environment for custom systems. Chipyard includes configurable, composable, open-source, generator-based designs that can be used across multiple stages of the hardware development flow while maintaining design intent and integration consistency. Through cloud FPGA simulation and rapid ASIC implementation, we demonstrate an iterative agile hardware design cycle which enables continuous validation of physically-realizable customized systems.**

***Index Terms*—system-on-chip, agile, hardware, simulation, VLSI, specialization**

I. Introduction

In the face of a slowdown in technology scaling, continued improvement in system efficiency requires increasing use of specialization and customization of chip architectures. This era of specialization is being seen as a new golden age of computer architecture [1], but raises new challenges in maintaining reasonable development costs. Differentiated architectures require efficient digital system design methods for architectural exploration, system integration, verification, validation, and physical design. As a results, various customizable and heterogeneous system design frameworks and methodologies have been proposed [2]–[5]. In particular, agile generator-based design methods provide a solid foundation for further innovation around the SoC design and implementation ecosystem.

Chipyard is a framework which brings together a collection of independently developed open-source tools and RTL generators, allowing concurrent research and development of heterogeneous SoCs through integrated environments. Chipyard helps alleviate many of the challenges posed when using independent and uncoordinated open-source tools and designs, as often experienced in concurrent and non-uniform feature design iterations, typical in the agile design process.

II. Chipyard

Chipyard provides a unified framework and work flow for agile SoC development. Multiple separately developed and highly parameterized IP blocks can be configured and interconnected to form a complete SoC design. The SoC design can be verified and validated through both FPGA-accelerated and standard software simulations, then pushed through portable VLSI design flows to obtain tapeout-ready GDSII data for various target technologies. Chipyard also provides a workload management system to generate software workloads to exercise the design.

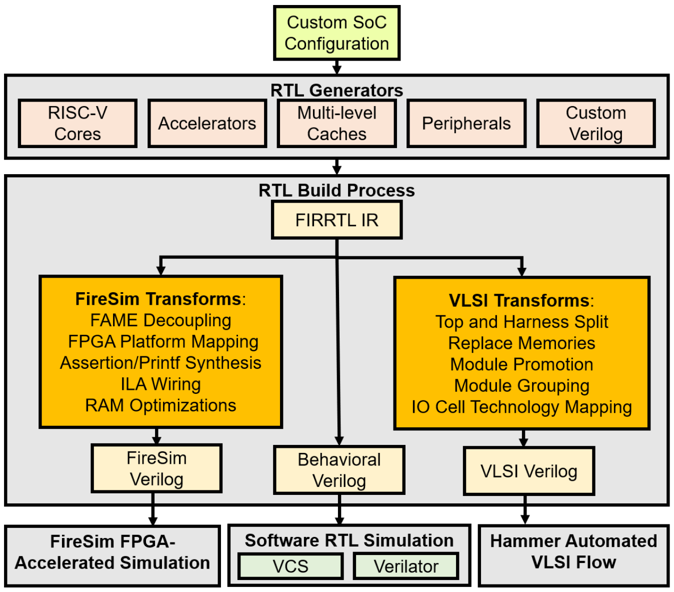

Custom SoC Configuration
RTL Generators
RISC-V Cores
Accelerators
Multi-level Caches
Peripherals
Custom Verilog
RTL Build Process
FIRRTL IR
FireSim Transforms: FAME Decoupling, FPGA Platform Mapping, Assertion/Printf Synthesis, ILA Wiring, RAM Optimizations
VLSI Transforms: Top and Harness Split, Replace Memories, Module Promotion, Module Grouping, IO Cell Technology Mapping
FireSim Verilog
Behavioral Verilog
VLSI Verilog
FireSim FPGA-Accelerated Simulation
Software RTL Simulation
VCS
Verilator
Hammer Automated VLSI Flow

Fig. 1. Multiple disparate design flows supported by the Chipyard framework through generators and transformations. Starting from the same generators and common custom configuration, a series of FIRRTL transformations outputs appropriate Verilog and associated collateral for different design-stage platforms.

A. Chipyard Front-End RTL Generators

The front end of the Chipyard framework is based on the Rocket Chip SoC generator [2], [3]. Chipyard inherits Rocket Chip's Chisel-based parameterized hardware generator methodology [3], including a Scala-based parameter-negotiation framework, Diplomacy [6], that negotiates mutually compatible parameterizations and interconnections across all IP blocks in a design. A unified top-level SoC generator enables the generation of heterogeneous systems based on parameterized configurations. Chipyard allows IP blocks written in other hardware languages, e.g., Verilog, to be included via a Chisel wrapper.

Chipyard adds a large corpus of open-source IP generators to the existing Rocket Chip base library, allowing for the construction of modern digital SoCs. These include the Berkeley Out-of-Order Machine (BOOM) generator [7], the Ariane core [8], the Hwacha vector-unit generator [9], digital signal processing (DSP) modules, domain-specific accelerators (including machine learning and cryptography), memory systems, and peripherals. The majority of these generators have silicon-proven instances in a variety of process technologies.

B. FIRRTL Intermediate Representation

The Chipyard framework currently integrates tools to address the three main activities within the custom SoC design cycle: front-end RTL design, system validation/verification, and back-end chip physical design. These different activities require different levels of design description. For example, while front-end RTL descriptions usually use abstract notions of memory and I/O, back-end RTL requires more precise descriptions mapped to the underlying process technology. Similarly, FPGA emulation requires the digital design to interact with FPGA-specific interfaces, periphery, and internal components. Co-simulation also requires additional hardware clock gating to control simulation progress.

Chipyard elaborates the front-end RTL design into a FIRRTL [10] intermediate representation. Custom FIRRTL transformations convert the generated FIRRTL design to drive the different flows used at different stages of the design cycle. Using FIRRTL transformations to enable multiple disparate design flows from the same shared code repository and source RTL helps to reduce and amortize the environment setup costs incurred with frequent iterations between development stages, as is needed for an agile methodology. This approach is demonstrated in Figure 1.

In contrast to alternative hardware package management systems [11] or integration standards like IP-XACT, which focus on metadata associated with particular IP components to target different EDA flows, FIRRTL transformations can perform wholescale manipulation of complete RTL designs in Chipyard.

C. FPGA-Accelerated Simulation with FireSim

For full-system validation and evaluation, the Chipyard framework harnesses the FireSim [12] open-source FPGA-accelerated simulation platform using the AWS EC2 public cloud. In contrast with FPGA *prototyping*, FPGA-accelerated *simulation* correctly models timing behavior of not only the design under test, but also the I/Os and peripherals of the SoC. Furthermore, FPGA-accelerated simulation in FireSim enables deterministic and reproducible evaluation within the realistic system environment, as opposed to FPGA prototyping where each execution is sensitive to the FPGA environment and timing depends on the performance of peripherals attached to the FPGA (e.g. DRAM performance). FireSim also provides FirePerf [13], a set of powerful on-FPGA out-of-band performance profiling tools that enable high-fidelity cycle-by-cycle introspection into software running on the simulated system, without perturbing the target system.

Originally developed as a platform to enable scale-out simulation for datacenter architecture research on hundreds of cloud FPGAs, FireSim necessarily automates the infrastructure management and simulation mapping necessary to automatically run high-performance simulations. As part of the agile chip-design stack, this automation and integration reduces the level of expertise required to harness cloud FPGAs for emulation purposes and thus increases the accessibility of high performance full-system simulation to a broad spectrum of designers. FireSim has been useful in pre-silicon verification, validation, and software development. From the perspective of small agile teams with limited resources, FireSim provides many of the features available in costly commercial emulation platforms. In contrast with prior FPGA-accelerated simulation tools, the accessibility of FireSim through FPGA instances on the AWS public cloud, as well as the automation of host-target interfaces with the FPGA, have made FireSim a popular tool within Berkeley and other academic hardware development users, as well as emerging startup companies.

FireSim enables co-development of software and hardware simultaneously, allowing for quick software adjustment turnarounds based on hardware modifications. Furthermore, FireSim plays a major role in the performance and functional validation of processors, since it enables the identification of bugs deep into simulation execution time thanks to FPGA-acceleration with appropriate peripheral modeling. Unlike many other open-source hardware development platforms with FPGA support, FireSim's focus on simulation and emulation as opposed to prototyping enables true pre-silicon performance evaluation and validation in a full-system context within the Chipyard framework. While maintaining its stand-alone operation as an architectural research platform, FireSim was transformed into a library which is integrated into the broader Chipyard framework. As such, FireSim can now consume design configurations composed within the Chipyard framework, and transform them into FPGA-accelerated simulations. Furthermore, the FireSim Golden Gate compiler has been integrated into the Chipyard framework, so it can now consume arbitrary FIRRTL as its input, as well as external Verilog components necessary for broader system integration.

D. Back-End Physical Design with Hammer

For back-end physical design, Chipyard includes a modular VLSI flow named Hammer [14]. The Hammer VLSI flow provides an abstraction layer above process-technology- and EDA-tool-specific concerns, with the goal of increasing re-use and modularity of vendor-specific components of the physical design flow. To this end, the Hammer VLSI flow utilizes separate vendor-specific process technology plug-ins and EDA-tool-specific plug-ins, which implement abstracted software APIs to generate design-flow collateral like Tcl scripts, clock constraints, and power specifications based on higher-level design inputs. For example, Hammer will emit process- and vendor-specific macro placement, obstruction, and power-strap placement commands from a high-level process- and vendor-agnostic description of the design. This separation of abstraction layers between design, process technology, and EDA tool vendor enables faster adoption of open-source components.

The Hammer flow aspires to support open-source tools in conjunction with commercial and proprietary tools by using common levels of abstraction. As such, while the first Hammer-based designs were implemented using proprietary process technologies, a plug-in for the ASAP7 [15] open-source predictive PDK was created in only a few weeks and is now included in the core Hammer repository. With this, small teams and academic users can prototype design flows and experiment with RTL designs using predictive or simple physical design kits, while being able to reuse similar Hammer descriptions for chip fabrication using advanced process nodes.

Hammer was designed to support hierarchical physical design flows. Hierarchical physical design flows are of particular importance in highly complex custom SoCs, composed of multiple specialized blocks with a variety of physical design constraints. Decomposing a design into these smaller hierarchical components not only improves the quality of results emitted by EDA tools, but it also allows the distribution of physical design tasks among multiple hardware developers, which is important for agile design. FIRRTL-based grouping and flattening transformations in Chipyard further assist the hierarchical physical design flow in Hammer by enabling users to specify one logical hierarchy in the source RTL, while choosing a different hierarchy for physical boundaries through automated transformations.

E. Input/Output Management

The various implementation and simulation flows in the SoC design process will typically treat the digital IOs on a system in different ways. For example, a software RTL simulation would typically connect digital IOs directly to software models in the *TestHarness*. However, on an FPGA prototype or FPGA-accelerated simulator, these IOs would be require some synthesizable bridge. For physical design, IO cells must be inserted into the module hierarchy.

In order to handle the various methods of providing input and output to the SoC across different simulation and implementation flows, the Chipyard framework uses *IOBinders*. IOBinders define the attachment behavior of IO ports to the digital system being developed. An IOBinder specifies a behavior for driving or interpreting an IO port of the digital system. Thus, each simulation or implementation flow specifies its own set of IOBinders to control how the digital IOs will be interpreted in that flow. This abstracts IO management from the actual implementation of the digital system.

F. Software Management

In order to enable complete SoC design and customization, software testing is treated as a first-class component within the Chipyard framework. As such, Chipyard provides a compatible set of software tools for development and testing. Chipyard provides a versioned set of standard RISC-V software development tools (e.g. GNU toolchain, QEMU, Spike ISA Simulator), as well as a set of equivalent non-standard RISC-V development tools for non-standard extensions of custom IP blocks. The two software development tool sets can be used interchangeably in the framework. Chipyard provides additional support for bare-metal software testing by using a minimalistic port of `libgloss` [16] for RISC-V Machine-mode. This port enables testing of bare-metal systems by implementing system calls through a Chipyard-compatible Host-Target interface for tethered systems.

Chipyard enables shared software development and management of complex software workloads through the FireMarshal software workload generation tool. FireMarshal provides a standard version-controlled format for software workload descriptions and automates the generation of these workloads for various simulation targets (e.g. Spike, QEMU, FireSim). FireMarshal is especially beneficial for Linux-based software workloads, where it makes the complex task of software development and porting easily reproducible and reusable by anyone on the design team without requiring special expertise.

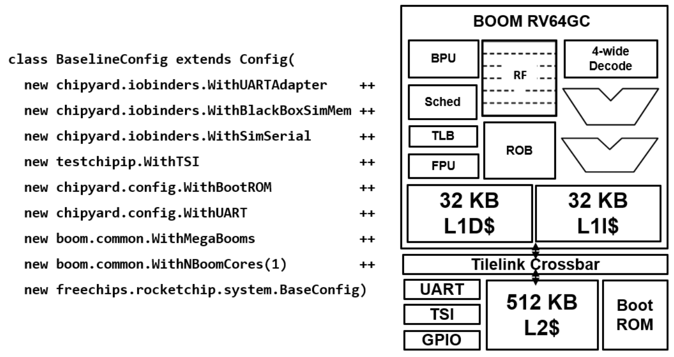
III. Agile Hardware Development Using Chipyard

To demonstrate the agile framework in action, we take an example baseline Chipyard SoC configuration and iteratively apply changes throughout the development and customization process. We will further examine and validate various properties relevant to those changes across the customization process. The example baseline Chipyard SoC includes a single BOOM out-of-order application-class processor, a shared L2 cache, and mix of UART, TSI and GPIO peripheral interfaces. This example baseline SoC and it's associated Chipyard configuration is shown in Figure 2. The SoC configuration is a composition of several other configuration *fragments* of system sub-components. For example a BOOM `WithMegaBooms` configuration fragment specifies a BOOM configuration with a decode pipeline width of 4 instructions, 3-wide integer issue, 2-wide floating-point issue, 2-wide memory issue, 128 ROB entries and 128 physical registers, and many other additional configuration parameters.

A. Adding an Accelerator

With the slowdown of Moore's law, many research endeavours begin with implementing a specialized compute accelerator for particular applications. Chipyard offers multiple methods of accelerator integration through the Rocket Chip generator with varying degrees of coupling to the host processor. These methods include memory-mapped peripheral accelerators and Rocket Custom Co-processors (RoCC).

Memory-mapped peripherals are a common method for custom accelerator integration in SoCs. In a memory-mapped peripheral, the processor communicates with the accelerator through memory-mapped registers using the TileLink bus protocol. Under such a scheme, the Rocket Chip generator will generate a complete memory map that can be used in



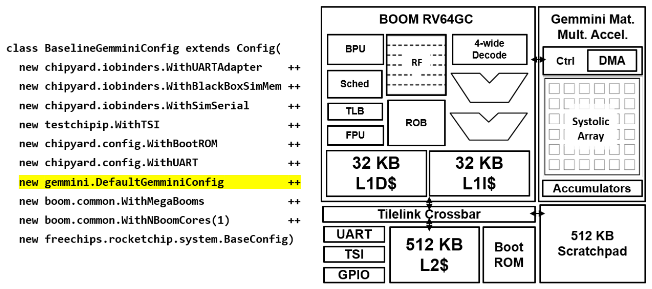
```
class BaselineConfig extends Config(
  new chipyard.iobinders.WithUARTAdapter      ++
  new chipyard.iobinders.WithBlackBoxSimMem ++
  new chipyard.iobinders.WithSimSerial        ++
  new testchipip.WithTSI                     ++
  new chipyard.config.WithBootROM            ++
  new chipyard.config.WithUART               ++
  new boom.common.WithMegaBooms              ++
  new boom.common.WithNBoomCores(1)          ++
  new freechips.rocketchip.system.BaseConfig)
```

Fig. 2. An example baseline SoC configuration consisting of a single 4-wide BOOM out-of-order application-class processor, a shared L2 cache, and mix of UART, TSI and GPIO peripheral interfaces.

software header files for drivers and other low-level software elements which implement the communication between the processor and the accelerator. Memory-mapped accelerators are relatively generally accessible from an SoC perspective in the sense they do not require particular support or interface implementation from the application processor cores. Nevertheless, they require software components in the form of device drivers, and are located further down the memory hierarchy from the processor core. This level of decoupling allows for the flexibility of integrating and sharing these accelerators under a variety of multi-core SoC configurations, but incurs software access costs of interrupts and memory operations when using them. An example of an open-source memory-mapped accelerator is the Nvidia Deep Learning Accelerator (NVDLA) [17]. The NVDLA has been used with the Rocket Chip generator in several projects [18], and is also integrated as part of the Chipyard framework.

Chipyard also supports tighter integration of accelerators through the RoCC (RoCC) interface. Cores that support the RoCC interface communicate with the accelerator through a custom protocol and custom non-standard RISC-V instructions reserved in the RISC-V ISA encoding space. The RoCC protocol enables RoCC accelerators to access the L1 data caches, stall the processor pipeline, and pass values through registers. Both BOOM and Rocket cores support the RoCC protocol. Each core can have up to four RoCC accelerators controlled by custom instructions and sharing resources with the CPU. An example of an open-source RoCC accelerator is the Gemmini machine-learning accelerator [19]. Unlike the NVDLA, Gemmini has direct DMA access to the SoC's shared L2 cache, can stall the host processor pipeline, and can be programmed directly from user-space with custom assembly instructions.

All accelerator integration methods within Chipyard fit within the Chipyard configuration system and allow for easy addition and removal from the SoC. As a result, the Chipyard generator repository includes multiple open-source accelerator IPs that can be added to the SoC using a single line of code within the SoC configuration. For example, Figure 3 demonstrates adding the Gemmini accelerator (which uses the RoCC interface) to the baseline SoC. This is done by adding the `DefaultGemminiConfig` fragment to the SoC configuration. the `DefaultGemminiConfig` fragment class is defined within the Gemmini project repository, and sets the various accelerator parameters such as the systolic array size (16x16), scratchpad size (512 KiB), accumulator sizes (64 KiB), datatypes (Int8), dataflows (WS), etc.



```
class BaselineGemminiConfig extends Config(
  new chipyard.iobinders.WithUARTAdapter      ++
  new chipyard.iobinders.WithBlackBoxSimMem ++
  new chipyard.iobinders.WithSimSerial        ++
  new testchipip.WithTSI                     ++
  new chipyard.config.WithBootROM            ++
  new chipyard.config.WithUART               ++
  new gemmini.DefaultGemminiConfig           ++
  new boom.common.WithMegaBooms              ++
  new boom.common.WithNBoomCores(1)          ++
  new freechips.rocketchip.system.BaseConfig)
```

Fig. 3. An SoC configuration adding a Gemmini machine learning accelerator to the baseline SoC config using a single configuration line (highlighted).

B. Accelerator Software Validation

Pre-silicon validation of software which uses the custom accelerator helps shorten the overall system development cycle, and identify functional bugs and performance pathologies when changes can still be made.

For the SoC configuration from the previous section, we want to evaluate the execution of DNN inference using the accelerator within the SoC. Executing the inference of a batch of 4 images using the standard ResNet-50 DNN model takes 4 billion cycles. Running such a software RTL simulation would take several days. The relevant testing and validation flow within the Chipyard framework uses the Spike functional ISA simulator and the FireSim FPGA-accelerated simulation platform. A functional model of the Gemmini accelerator is integrated into a non-standard Spike functional simulator, which enables initial software development with the Gemmini custom instruction extensions. Once functional results are satisfactory, further performance tuning is performed using the FireSim FPGA-accelerated simulation platform. Executing the 4 billion cycles of ResNet-50 on Gemmini with FireSim take a mere few seconds of wall-clock time. While initial FPGA simulation synthesis and build time is longer than standard RTL software simulation compilation time, the combined synthesis and build time is significantly shorter than software simulation time. The one-off build synthesis time overhead is further amortized over a large number of simulation runs when used for pre-silicon software performance optimization on the simulated custom SoC.

In fact, this flow of complete validation of software using custom accelerators on the Chipyard framework has become sufficiently easy-to-use that it can also be used for accelerator evaluation under educational settings. The aforementioned flow for hardware-software performance optimization of a

```
class QuadRocketGemminiConfig extends Config(
  new chipyard.iobinders.WithUARTAdapter                  ++
  new chipyard.iobinders.WithBlackBoxSimMem               ++
  new chipyard.iobinders.WithSimSerial                    ++
  new testchipip.WithTSI                                  ++
  new chipyard.config.WithBootROM                         ++
  new chipyard.config.WithUART                            ++
  new gemmini.SmallGemminiConfig                          ++
  new freechips.rocketchip.subsystem.WithNBigCores(4)++
  new freechips.rocketchip.system.BaseConfig)
```

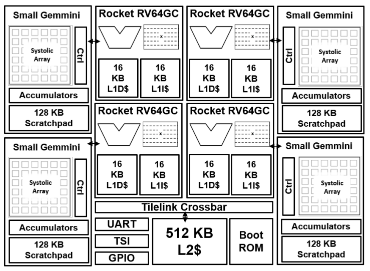


Fig. 4. An SoC configuration replacing the single BOOM out-of-order core with 4 smaller Rocket in-order cores and smaller Gemmini accelerators to evaluate PPA tradeoffs. The highlighted lines represent the configuration lines that were replaced.

Gemmini accelerator was used in a class-wide lab in a recent course offered at UC Berkeley [20].

C. Core Power-Performance-Area Tradeoffs

SoC designers can choose amongst a variety of application cores to anchor an SoC. A common Power-Performance-Area (PPA) consideration when designing a custom SoC is whether to use smaller and more energy efficient parallel cores vs. larger and more power-consuming high-performance single-thread cores. While a particular application amenable to parallel processing might benefit from a multi-core configuration, Amdahl's law reminds us that single-threaded performance limits the potential overall performance gains from parallelism.

Thus, determining the right core configuration requires careful consideration of the expected software workload of the device. A SoC designer might choose to explore multiple core configurations before picking a design point. The Chipyard framework does not "lock-in" any particular core configuration or core count, as the framework can generate reasonable single-core and multi-core designs, with core configurations ranging from small embedded-class in-order cores to large application-class out-of-order cores.

For example, after careful analysis of the software workloads meant to run on the target SoC in Figure 3, the designer may want to explore the PPA tradeoff of a more parallel design-point which trades-off the powerful single-thread performance of the BOOM out-of-order core for multiple smaller, but more efficient Rocket in-order cores with smaller Gemmini accelerators. With just a two-line change, the designer can reconfigure Chipyard to generate this vastly different SoC architecture illustrated in figure 4 with four smaller in-order cores each driving a smaller 8x8 Gemmini accelerator. The designer can then take advantage of Chipyard simulation and VLSI flows to quickly measure the power-performance tradeoffs of either design. Figure 5 demonstrates such an area tradeoff comparison using a commercial FinFET process and the Hammer flow as part of the SoC design process. Similar estimations can be performed using open-source PDKs.

D. Non-invasive Physical Design Optimization

Throughout the implementation process, constraints imposed by the physical and computational realities of VLSI

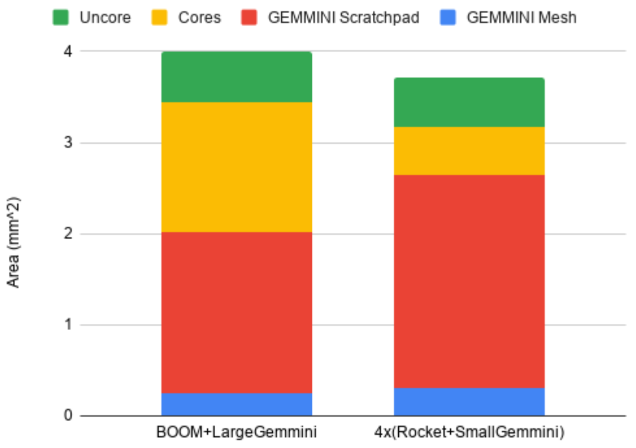


Fig. 5. Post-synthesis area estimates using a commercial FinFET process comparing a powerful single-core system with an alternative equivalent parallel multi-core system.

processes many times result in divergences from the original architectural design. Chipyard attempts to address the design divergences that appear between digital/architectural simulation and silicon implementation, through a "source-of-truth" RTL generator with re-usable and customizable RTL passes that transform the RTL for VLSI flows. It is important that the source hardware description simulated during architectural exploration is as close as possible to the hardware description that goes through physical design and implementation.

Instead of performing a flat layout in which the entire SoC design is ran through the VLSI EDA tools at once, many SoCs instead use a hierarchical flow. In a hierarchical physical design flow, the SoC is split into smaller subcomponents which are later assembled together, so VLSI EDA tools can better optimize the smaller sub-blocks instead of attempting to find a global layout optimum of the entire SoC. When combined with floorplanning to get better timing and reduce wire congestion, the physical module hierarchy can differ significantly from the logical hierarchy described in the source RTL. However, directly modifying the source RTL to move and group modules to fit the new hierarchy is both time-intensive and error-prone. Leveraging the power of FIRRTL, a set of FIRRTL passes can be applied to the design to quickly make large-scale hierarchy changes while maintaining correct functionality. This is illustrated in Figure 6, in which the prior design in Figure 4 was transformed into four module regions: two clusters that both include two Rocket cores and two Gemmini accelerators, an IO region that contains UART, TSI, and GPIO interfaces, and an uncore region that contains items like the L2 cache and bus subsystems. This is done with a native FIRRTL transformation called `GroupAndDedup` which groups modules together, followed by the `InlineInstances` FIRRTL transformation to split up a module into its submodules. This new hierarchy is used within a hierarchical physical design flow which synthesizes, places and routes each of the components individually and

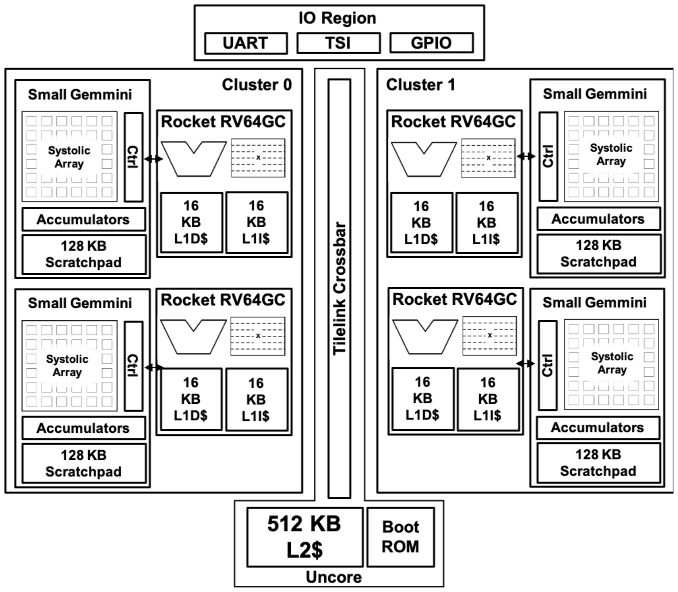


Fig. 6. Physical-design friendly hierarchy generated after FIRRTL transformations, using the same SoC configuration from Figure 4.

then assembles them together. As a result, the overall time of a placement and routing iteration is reduced from 100's of hours to 10's of hours.

By using these mainstream FIRRTL passes, the main Chisel RTL design is unchanged from the one that was originally tested and verified, but the design can now continue iterating through more efficient VLSI flow quality-of-results (QoR) optimizations. This enables shortening of design cycle iterations through overlapping simulation, testing and physical design.

IV. Conclusion

In this article, we presented the Chipyard framework, an integrated SoC research and implementation environment for agile development. Through integration with the Rocket Chip generator ecosystem, Chipyard provides a large number of easily composable and extendable open-source digital designs, enabling the construction and customization of complex SoCs from inception to implementation. We demonstrate an example exploratory SoC design flow using the the multiple simulation and implementation tools integrated within Chipyard, which enable continuous and simultaneous agile development for higher-quality verification, validation, and system integration.

V. Acknowledgments

The information, data, or work presented herein was funded in part by the Defense Advanced Research Projects Agency (DARPA) through the Circuit Realization at Faster Timescales (CRAFT) Program under Grant HR0011-16-C0052, and by the Advanced Research Projects Agency-Energy (ARPA-E), U.S. Department of Energy, under Award Number DE-AR0000849. Research was partially funded by ADEPT Lab industrial sponsors and affiliates. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

References

[1] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019.

[2] Y. Lee, A. Waterman, H. Cook *et al.*, "An agile approach to building risc-v microprocessors," *IEEE Micro*, vol. 36, no. 2, pp. 8–20, Mar 2016.

[3] K. Asanović, R. Avizienis, J. Bachrach *et al.*, "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

[4] J. Balkind, M. McKeown, Y. Fu *et al.*, "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.

[5] L. P. Carloni, "Invited - the case for embedded scalable platforms," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 17:1–17:6.

[6] H. Cook, W. Terpstra, and Y. Lee, "Diplomatic design patterns: A tilelink case study," in *1st Workshop on Computer Architecture Research with RISC-V*, 2017.

[7] C. Celio, D. A. Patterson, and K. Asanovic, "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*, 2015.

[8] 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.

[9] Y. Lee, C. Schmidt, A. Ou *et al.*, "The hwacha vector-fetch architecture manual, version 3.8. 1," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-262*, 2015.

[10] A. Izraelevitz, J. Koenig, P. Li *et al.*, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *Proceedings of the 36th International Conference on Computer-Aided Design*, ser. ICCAD '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 209–216.

[11] O. Kindgren, "Invited paper: A scalable approach to ip management with fusesoc," in *Workshop on Open Source Design Automation*, 2019.

[12] S. Karandikar, H. Mao, D. Kim *et al.*, "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 29–42.

[13] S. Karandikar, A. Ou, A. Amid *et al.*, "FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design," 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. 715–731.

[14] E. Wang, C. Schmidt, A. Izraelevitz *et al.*, "A methodology for reusable physical design," in *Twenty First International Symposium on Quality Electronic Design, 2020. Proceedings.*, March 2020.

[15] L. T. Clark, V. Vashishtha, L. Shifren *et al.*, "Asap7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.

[16] Libgloss, a free board support package (bsp). [Online]. Available: https://www.gnu.org/software/dejagnu/manual/Libgloss.html

[17] F. Sijstermans, "The nvidia deep learning accelerator," in *Hot Chips 30: The Flint Center for the Performing Arts, Cupertino, California, August 19–21, 2018*, 2018. [Online]. Available: https://www.hotchips.org/hc30/2conf/2.08_NVidia_DLA_Nvidia_DLA_HotChips_10Aug18.pdf

[18] F. Farshchi, Q. Huang, and H. Yun, "Integrating NVIDIA deep learning accelerator (NVDLA) with RISC-V SoC on FireSim," in *In proccedings of The 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications, at HPCA 2019*, 2019. [Online]. Available: http://arxiv.org/abs/1903.06495

[19] H. Genc, A. Haj-Ali, V. Iyer *et al.*, "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures," 2019.

[20] A. Amid, H. Genc, and S. Y. Shao. (2020) EE290-2 Hardware for Machine Learning. Lab 3: Tiling and Optimization for Accelerators. [Online]. Available: http://www-inst.eecs.berkeley.edu/~ee290-2/sp20/assets/labs/lab3.pdf